Multiprocessor Architecture Using an Audit Trail for Fault Tolerance

Dwight Sunada[1], David Glasco[2], and Michael Flynn[1]

[1]Computer Systems Laboratory
Stanford University
Stanford, California 94305
dwight@cs.stanford.edu
flynn@umunhum.stanford.edu

[2]Austin Research Laboratory
International Business Machines, Inc.
Austin, Texas 78758
dglasco@us.ibm.com

Abstract

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

I. Introduction

A tightly-coupled multiprocessor (TCMP) is a multiprocessor where specialized hardware maintains the image of a single shared memory. In order to deploy a TCMP in the commercial world, the TCMP must be fault tolerant. The dominant method of fault tolerance is roll-back recovery and has 2 principal aspects. First, a processor establishes an occasional checkpoint, a consistent state of the system. Second, if a processor encounters a fault, the processor rolls back to the last checkpoint and commences execution from the state saved in that checkpoint. The first aspect, the establishment of checkpoints, is the more important one as it is a cost that the TCMP regularly experiences even if no fault arises. The second aspect, the actual rolling-back, is less important as faults tend to occur infrequently. Hence, much of the research in roll-back recovery for TCMPs has focused on developing efficient algorithms for establishing checkpoints. This paper presents the first apparatus and algorithm enabling a processor to perform roll-back or checkpoint establishment in a way that is independent of any other processor in a TCMP. Our algorithm is called distributed recoverable shared memory with logs (DRSM-L).

II. Background

Throughout our discussion, we assume that a memory block and the highest-level-cache line are identical in size and that the TCMP uses a write-back cache policy. To minimize the cost of the system, we assume that it can hold only 1 level of checkpoint.

Under these assumptions for a TCMP, a dependency can arise when 2 processors, "P" and "Q", access the same memory block. Two types of processor interaction cause dependencies to arise.

1. **<u>write–read</u>**: A write by "P" precedes a read by "Q".
   roll-back dependency: P -> Q   checkpoint dependency: Q -> P
2. **<u>write–write</u>**: A write by "P" precedes a write by "Q".
   roll-back dependency: P <-> Q   checkpoint dependency: P <-> Q

For a write-read interaction, if "P" rolls back to the last checkpoint, then "Q" must also roll back to the last checkpoint. If "Q" establishes a checkpoint, then "P" must also establish a checkpoint. For a write-write interaction, if one processor rolls back to the last checkpoint or establishes a new checkpoint, then the other processor must roll back to its last checkpoint or establish a new checkpoint, respectively [13].

The current schemes for establishing checkpoints in TCMPs fall into 2 major categories: tightly synchronized method (TSM) and loosely synchronized method (LSM) [13]. Wu proposes a TSM-type algorithm [16]. If a checkpoint dependency arises between any 2 processors, the processor supplying the dirty data must immediately establish a checkpoint. Both Ahmed [2] and Hunt [6] have done work related to research by Wu.

An example of a LSM-type algorithm is one presented by Banatre [3]. If a checkpoint dependency arises, the TCMP simply records the dependency without requiring the immediate establishment of a checkpoint. At some time in the future, if a processor establishes a checkpoint, that processor must query the records of dependencies to determine all other processors that must establish a checkpoint as well. Hence, a LSM-type algorithm is more flexible than a TSM-type algorithm in terms of when checkpoints must be established.

There exists a 3rd category of checkpointing algorithms: unsynchronized method (USM) [13]. In a USM-type algorithm, a processor can establish a checkpoint (or roll back to the last checkpoint) without regard to any other processor. Some USM-type algorithms [10][14] exist for a loosely-coupled multiprocessor like a network of workstations, but until now, such algorithms did not exist for TCMPs. In this paper, we present DRSM-L, the first USM-type algorithm for a TCMP.

III. Assumptions

The TCMP into which we shall incorporate DRSM-L is a multi-node multiprocessor like that shown in Figure 2. Each node has a processor module and a memory module. The nodes are connected by a high-speed dedicated network. We assume the following.

1. The processor module is not fault tolerant but is fail-safe. We can use double-modular redundancy to quickly detect whether the output of a processor module is faulty <u>before</u> the output is emitted from the processor module.
2. The TCMP suffers at most a single point of failure.
3. The network and each memory module is fault tolerant.

4. The virtual machine monitor (VMM) is fault-tolerance aware. Specifically, if communication occurs between a processor and the environment outside of the TCMP, then the VMM will invoke the processor to establish a checkpoint.

The first 3 assumptions are commonly found in research papers proposing checkpointing algorithms for a TCMP. Under the 4th assumption, the TCMP views the OS as simply another user application running on top of the VMM [4]. It enables us to run any non-fault-tolerant OS while the entire TCMP remains fault-tolerant.

IV. DRSM-L

A. Apparatus

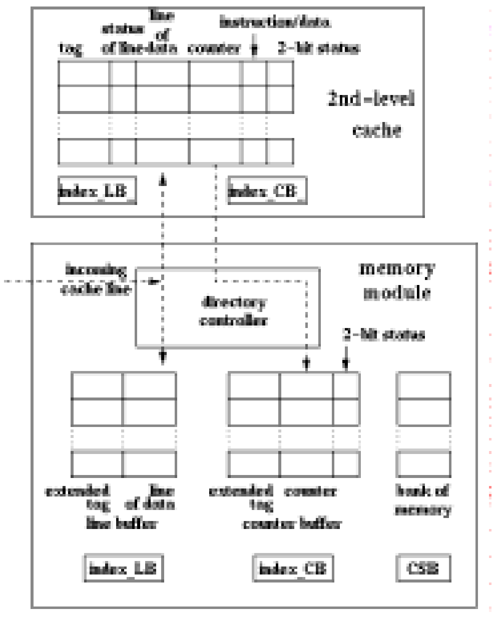

Figure 1. DRSM-L

DRSM-L is an apparatus and algorithm that enables the TCMP to recover from a failure of the processor module. Figure 1 illustrates the apparatus of DRSM-L. Each line of the 2nd-level cache has the traditional fields: tag, status (SHARED, EXCLUSIVE, and INVALID) of line, and line of data. Each line has 3 additional fields: counter, instruction/data flag (IDF), and 2-bit status flag (SF). The SF assumes any 1 of 4 values: "N" (no event), "R" (remote read), "E" (ejection), "V" (counter overflow). The cache also has 2 index registers that mirror 2 index registers in the local memory module.

The local memory module has the traditional directory controller and bank of memory. The module also has a line buffer and a counter buffer. Each buffer has an index register ("index_LB" or "index_CB") pointing at the next free entry. There is also a checkpoint-state buffer (CSB).

In the following discussion, we assume that the TCMP (1) prohibits self-modifying code and (2) requires instructions and regular data to reside in separate memory blocks (i. e. cache lines). If a cache line contains instructions, then the IDF is 0; if the line contains data, then the IDF is 1. DRSM-L uses only regular data to build an audit trail.

B. Algorithm

The general operation of DRSM-L is the following. The directory controller stores each incoming memory block (of regular data) destined for the 2nd-level cache into the next entry (at which the "index_LB" points) in the line buffer and, concurrently, forwards the memory block to the cache. Each data-access hitting on a 2nd-level-cache line causes its counter to increment; the counter records the number of accesses between consecutive events. When a cache-coherence event like (1) a write-back due to a dirty read by a remote processor or (2) an ejection (i. e. eviction/invalidation) occurs on a 2nd-level-cache line, the cache sends a reply to the local directory controller and, concurrently, forwards the counter to it in the reply. The directory controller installs the counter into the counter buffer. Saving (1) the incoming memory block into the line buffer and (2) the counter into the counter buffer incurs <u>no additional delay</u> since these events occur in parallel with the usual activities of (1) forwarding the memory block from the directory controller to the 2nd-level cache and (2) maintaining cache coherence, respectively. The contents of the line buffer and the counter buffer constitute an audit trail of incoming memory blocks and cache-coherence events that occurred since the last checkpoint.

A recovery logic circuit (RLC) in each memory module periodically sends "Are you alive?" messages to the processor. If it does not respond within a timeout period, then the RLC concludes that a fault has occurred. The RLC resets the local processor if the fault is transient or the processor in the spare processor module if the fault is permanent. Then, the processor performs recovery by resuming execution from the state saved in the last checkpoint. On each 2nd-level-cache miss, the VMM installs the next matching memory block from the line buffer into the appropriate cache line and also installs the next matching counter and SF from the counter buffer into that same cache line. Each data-access hitting on a 2nd-level-cache line causes its counter to decrement; once it underflows, the VMM simulates the cache-coherence event indicated by the SF.

Finally, a processor establishes a checkpoint when (1) the line buffer overflows, (3) the counter buffer overflows, (3) a timer expires, or (4) communication occurs between a processor and the environment outside of the TCMP. Establishing a checkpoint empties the 2 buffers.

Below, we use C-like code to precisely describe how the DRSM-L (1) fills the line buffer and counter buffer in the normal mode of execution, (2) rolls the processor back to the last checkpoint after encountering a fault, (3) uses the audit trail to satisfy data-access misses and to simulate cache-coherence events, and (4) establishes a checkpoint. In our code, we refer to the 2nd-level cache as simply "cache". Due to limitations on space, we present only the key aspects of the full algorithmic description in [13].

<u>explanatory notes</u>

States of cache line are INVALID, SHARED, and EXCLUSIVE.
Number of entries in line buffer is 8192.
Number of entries in counter buffer is 8192.
Extended tag is tag appended with index of specific cache line into which memory block is installed.
Width of counter is 32 bits.

<u>execution mode: normal</u>

```
switch (event) {   /*   events in cache   */
```

```
  data_write_has_upgrade_miss_in_cache_data_line: {   ;   }
  data_access_misses_in_cache_data_line: {
    if (index_LB_ == 0x2000) establish_checkpoint();   }
  data_access_hits_in_cache_data_line: {
    if (cache_line.counter == 0x0FFFFFFFF) {
      if (index_CB_ == 0x2000) {
        establish_checkpoint();   }
      else {
        index_CB_++;
        log_counter_into_counter_buffer(cache_line, "V");   }   }
    else {
      cache_line.counter++;   }   }
  evict_cache_line:
  invalidate_cache_line: {
    if (index_CB_ == 0x2000) {
      establish_checkpoint();   }
    else {
      index_CB_++;
      log_counter_into_counter_buffer(cache_line, "E");   }   }
  remotely_read_local_dirty_cache_line: {
    if (index_CB_ == 0x2000) {
      establish_checkpoint();   }
    else {
      index_CB_++;
      log_counter_into_counter_buffer(cache_line, "R");   }   }
  timer_expires:
  communication_between_cpu_and_environment_
                              outside_TCMP_occurs: {
    establish_checkpoint();   }
  default: {
    do nothing special;   }   }

switch (event) {   /*   events in directory controller   */
  memory_block_arrives: {
    if (original access is data access) {
      log_memory_block_into_line_buffer();
      install_memory_block_into_cache_data_line();   }
    else {
      install_memory_block_into_cache_instruction_line();   }   }
  default: {   do nothing special;   }   }

log_memory_block_into_line_buffer() {
  line_buffer[index_LB].extended_tag <=
                        extended_tag(memory_block);
  line_buffer[index_LB].line_of_data <= data(memory_block);
  index_LB++;   }
log_counter_into_counter_buffer(cache_line, event) {
  counter_buffer[index_CB].extended_tag <=
                              extended_tag(cache_line);
  counter_buffer[index_CB].counter <= cache_line.counter;
  counter_buffer[index_CB].SF <= event;
  index_CB++;   }
install_memory_block_into_cache_instruction_line() {
  cache_line.IDF <= 0;   cache_line.counter <= 0;
  set tag, status_of_line, and line_of_data of cache_line;}
install_memory_block_into_cache_data_line() {
  index_LB_++;   cache_line.IDF <= 1;   cache_line.counter <= 0;
  set tag, status_of_line, and line_of_data of cache_line;}
```

fault detection and roll-back

```
if (RLC detects fault in processor module) {
  if (fault == permanent) {
    replace processor module with spare module;
    reset spare processor module, invalidating all entries
      in both 1st-level cache and 2nd-level cache;   }
  else {
    reset processor module, invalidating all entries
      in both 1st-level cache and 2nd-level cache;   }
  trap to virtual machine monitor;
  query all memory modules to find lost cache-coherence messages;
  negatively acknowledge all cache-coherence messages
    until recovery is complete;
  if (CSB.CF == PERMANENT_CHECKPOINT_IS_ACTIVE) {
    complete_permanent_checkpoint();
    i <= CSB.toggle_flag;
    if (CSB.checkpoint_area[i].status !=
              PERMANENT_CHECKPOINT_AREA) {
      i = 1 - CSB.toggle_flag;   }
    load internal state of processor
      from CSB.checkpoint_area[i].processor_state;
    for (each cache_line in cache) {
      load cache_line from CSB.checkpoint_area[i].cache;
      cache_line.counter <= 0;   }
    return from trap to virtual machine monitor;
    exit and resume normal execution;   }
  if (CSB.CF == TENTATIVE_CHECKPOINT_IS_ACTIVE) {
    i <= 1 - CSB.toggle_flag;   CSB.checkpoint_area[i].status <= NULL;
    CSB.CF <= CHECKPOINT_IS_NOT_ACTIVE;
    discard tentative checkpoint;   }
  read all valid entries from line buffer;
  group all entries according to cache index but, for each
    cache index, maintain the temporal order in which the
    entries were originally inserted into the line buffer;
  place grouped entries into sorted_line_buffer;
  read all valid entries from counter buffer;
  group all entries according to cache index but, for each
    cache index, maintain the temporal order in which the
    entries were originally inserted into the counter buffer;
  place grouped entries into sorted_counter_buffer;
  i <= CSB.toggle_flag;
  if (CSB.checkpoint_area[i].status !=
            PERMANENT_CHECKPOINT_AREA) {
    i = 1 - CSB.toggle_flag;   }
  load internal state of processor
    from CSB.checkpoint_area[i].processor_state;
  for (each cache_line in cache) {
    load cache_line from CSB.checkpoint_area[i].cache;
    cache_line.counter <= 0;   cache_line.SF <= "V";   }
  return from trap to virtual machine monitor;
  enter recovery mode of execution;   }
```

execution mode: recovery

```
switch (event) {   /*   events in cache   */
  data_write_has_upgrade_miss_in_cache_data_line: {
    cache_line.status_of_line <= EXCLUSIVE;   }
  data_access_misses_in_cache_data_line: {
    trap to virtual machine monitor;
    cache_line <= available_cache_line();
    get_entry_from_sorted_line_buffer(cache_line)
    get_entry_from_sorted_counter_buffer(cache_line);
    return from trap to virtual machine monitor;
    exit_recovery_upon_completion();   }
  data_access_hits_in_cache_data_line: {
    switch (cache_line.SF) {
      "N": {   cache_line.counter <= 0;   }
      "E": {
        if (cache_line.counter != 0) {
          cache_line.counter--;   }
        else {
          trap to virtual machine monitor;
          cache_line.status_of_line <= INVALID;
          get_entry_from_sorted_line_buffer(cache_line);
          get_entry_from_sorted_counter_buffer(cache_line);
          return from trap to virtual machine monitor;   }   }
      "R": {
        if (cache_line.counter != 0) {
          cache_line.counter--;   }
        else {
          trap to virtual machine monitor;
          cache_line.status_of_line <= SHARED;
          get_entry_from_sorted_counter_buffer(cache_line);
          return from trap to virtual machine monitor;   }   }
      "V": {
        if (cache_line.counter != 0) {
          cache_line.counter--;   }
        else {
          trap to virtual machine monitor;
          get_entry_from_sorted_counter_buffer(cache_line);
          return from trap to virtual machine monitor;   }   }   }
    exit_recovery_upon_completion();   }
  default: {   exit_recovery_upon_completion();   }   }

switch (event) {   /*   events in directory controller   */
memory_block_arrives: {
    if (original access is data access) {   ;   }
    else {
      install_memory_block_into_cache_instruction_line();  }   }
default: {
    do nothing special;   }   }

get_entry_from_sorted_line_buffer(cache_line) {
  get next matching entry from sorted_line_buffer;
  cache_line.tag <= tag(sorted_line_buffer[entry].extended_tag);
  if (data-access == write) {
    cache_line.status_of_line <= EXCLUSIVE;   }
  else {
    cache_line.status_of_line <= SHARED;   }
  cache_line.line_of_data <=
              sorted_line_buffer[entry].line_of_data;   }
get_entry_from_sorted_counter_buffer(cache_line) {
  get next matching entry from sorted_counter_buffer;
  if (no matching counter) {
    cache_line.counter <= 0;
    cache_line.IDF <= 1;   cache_line.SF <= "N";   }
  else {
    cache_line.counter <= sorted_counter_buffer[entry].counter;
    cache_line.IDF <= 1;
    cache_line.SF <= sorted_counter_buffer[entry].SF;   }   }
exit_recovery_upon_completion() {
  if ((sorted_counter_buffer has no entry where SF is "E"
      or "R") && (counters in all valid cache data lines
      where SF is "E" or "R" are 0)) {
    for (each valid cache_line in cache) {
      switch (cache_line.SF) {
        "E": {   cache_line.status_of_line <= INVALID;   }
        "R": {   cache_line.status_of_line <= SHARED;   }
        default: {   ;   }   }   }
    establish_checkpoint();
    write_back_and_invalidate_cache_lines();
    exit recovery and resume normal execution;   }
  else {   ;   }   }
```

checkpoint establishment

```
establish_checkpoint() {
  CSB.CF <= TENTATIVE_CHECKPOINT_IS_ACTIVE;
  wait until all pending memory accesses are completed
                        or negatively acknowledged;
  negatively acknowledge all cache-coherence messages
                          until checkpoint is established;
  establish_tentative_checkpoint();
  CSB.CF <=   PERMANENT_CHECKPOINT_IS_ACTIVE;
  establish_permanent_checkpoint();
  CSB.CF <= CHECKPOINT_IS_NOT_ACTIVE;   }
establish_tentative_checkpoint() {
  i <= 1 - CSB.toggle_flag;
  save tag, status_of_line, line_of_data, and IDF of all
       cache lines into CSB.checkpoint_area[i].cache;
  save internal state of processor
       into CSB.checkpoint_area[i].processor_state;
  CSB.checkpoint_area[i].status <=
        TENTATIVE_CHECKPOINT_AREA;   }
establish_permanent_checkpoint() {
  index_LB_ <= 0;   index_CB_ <= 0;
  index_LB <= 0;   index_CB <= 0;
  for (each cache_line in cache) {   cache_line.counter <= 0;   }
  i <= CSB.toggle_flag;
  CSB.checkpoint_area[i].status <= NULL;
  i <= 1 - CSB.toggle_flag;
  CSB.checkpoint_area[i].status <=
        PERMANENT_CHECKPOINT_AREA;
  CSB.toggle_flag <= i;   }
```

V. Hardware and Software Issues

For a given amount of silicon area from which we can build the line buffer and counter buffer, the optimal size of each is one that minimizes the number of checkpoints. The optimal size of each buffer is one where

optimum ratio = E[CB] / E[LB] = R[CB] / R[LB].   (equation #1)

"E[CB] " and "E[LB]" are the number of entries in the counter buffer and the line buffer, respectively. "R[CB]" and "R[LB]" are the rates at which the counter buffer and the line buffer, respectively, fill [13].

On the software side, the description of the above algorithm applies to a single thread of a single process running on a processor in a TCMP. In order to deal with multiple threads and processes, the DRSM-L must direct a processor, "P", to establish a checkpoint just after "P" switches context (and before "P" sends any dirty data to the rest of the TCMP).

Establishing a checkpoint at each context switch will not cause appreciable deterioration in performance. Establishing a checkpoint involves mainly saving the 2nd-level-cache and processor state into the CSB and, hence, costs about 41 microseconds for a 8192-line cache of a 200 megahertz processor. The fastest context-switch time (of a thread) is approximately 8 microseconds, scaled for a 200 megahertz SPARC processor from the results by Narlikar [9]. The checkpoint time and the context-switch time have roughly the same order of magnitude.

VI. Simulation Methodology

A. Multiprocessor Simulator

We evaluated DRSM-L by simulating its operation within a multiprocessor simulator. Figure 2 illustrates its base TCMP. The model of the memory system and the network is the NUMA model packaged with the SimOS simulator [5]. Instead of SimOS, we use ABSS to simulate the processors and to drive the NUMA model [11]. The delays through the components in figure 2 have values that are typical for a processor running at 200 megahertz and are listed in [13]. Below are the other parameters.

base parameters

processor = SPARC V7 @ 200 megahertz
cache policy = write-back
memory model = sequential consistency
1st-level instruction cache = 32 kilobytes with 4-way set
    associativity, 2 states (INVALID, SHARED), 64-byte line

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

DRSM-L parameters

width of counter = 32 bits
line buffer = 8192 entries
counter buffer = 8192 entries
timer = expiration per 20 million cycles

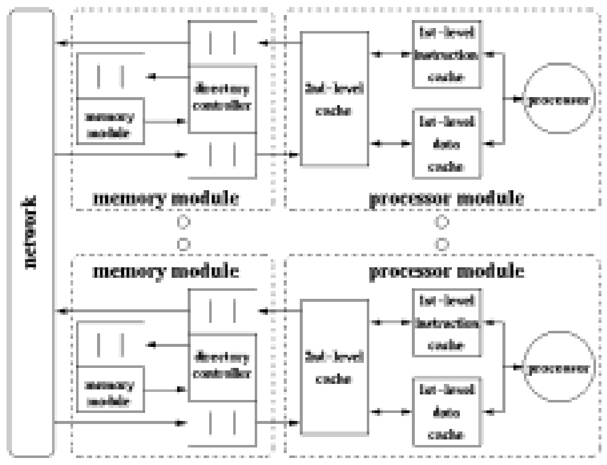

Figure 2. Base TCMP

B. Benchmarks

We run 6 benchmarks -- Cholesky, FFT, LU, ocean, radix, and water -- from the SPLASH2 suite [15] . 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 have 2 common characteristics. First, the working set of each benchmark fits within our large 2nd-level cache. Second, these benchmarks represent a scientific workload. They are useful in representing a wide variety of memory-access patterns but do virtually no communication with the environment outside of the TCMP. So, a checkpoint triggered by communication between a processor and the environment outside of the TCMP does not occur. We note that regardless of the event triggering a checkpoint, the procedure for establishing one remains the same, so we can still evaluate the performance of our algorithms even if checkpoints are triggered by a smaller set of events.

C. LSM-type Apparatus and Algorithm

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

We very briefly describe how DRSM works. Bank #1 of memory holds the working data (and tentative checkpoint), and bank #2 holds the permanent checkpoint. The dependency matrix records checkpoint dependencies that can arise when 2 processors access the same memory block in the memory module. If a processor, "P", wishes to establish a checkpoint, the DRSM recursively queries all the dependency matrices and identifies all processors that are dependent on "P". Then, "P" and all its dependent processors establish a checkpoint. A processor establishes a checkpoint by saving the processor state into the local memory module and by writing all dirty 2nd-level-cache lines back into main memory (i. e. bank #1).

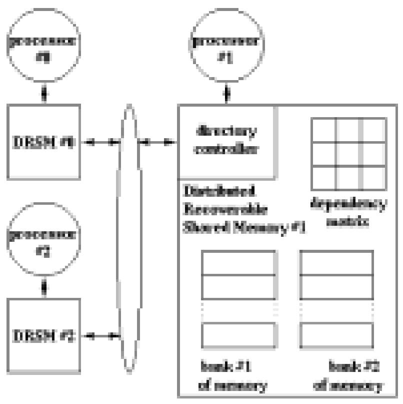

Figure 3. DRSM

The establishment of a checkpoint is triggered by only 2 events: (1) expiration of a timer and (2) communication between a processor and the environment outside of the TCMP.

The DRSM described here differs from the DRSM in prior work [13] in regards to only 1 aspect. In the current DRSM, bank #2 always holds the permanent checkpoint, but in the prior DRSM, bank #2 alternates between holding permanent-checkpoint data and holding working data. The extra functionality in the prior DRSM proved unnecessary, so we removed the functionality and simplified the hardware.

VII. Results and Analysis

A. Overall Performance of Benchmarks

Due to space limitations, we focus on 2 representative benchmarks: Cholesky and ocean. (The full set of results appears in [13].) Figure 5 shows their performance on the base TCMP, the TCMP with DRSM, and the TCMP with DRSM-L. We set the number of processors to 8, 16, and 32. We decompose the execution time into 5 categories: non-idle time of the processor, the instruction stall, the lock stall, the data stall, and the barrier stall. In general, the performance of DRSM-L exceeds the performance of DRSM.

For ocean, the barrier stall and the lock stall increase substantially as the number of processors increases from 8 to 32 processors because ocean, unlike Cholesky, has (1) several locks within global barriers and (2) global locks. All processors compete for these locks, causing hot spots to arise.

B. Checkpoints and Checkpoint Data

Table 2 shows statistics about the rate at which DRSM-L establishes checkpoints. Each application has 4 rows of statistics. The 1st row indicates the total number of checkpoints established per processor. Checkpoints in DRSM-L are triggered by effectively 3 events: (1) timer expiration, (2) line-buffer overflow, and (3) counter-buffer overflow. The checkpoints attributed to each trigger appear in the 2nd row of statistics. For example, the 2nd row for Cholesky running on an 8-processor TCMP has “(9.62 + 0.12 + 0.62)”. The number of checkpoints due to timer expiration, line-buffer overflow, and counter-buffer overflow are 9.62, 0.12, and 0.62, respectively.

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

The data for DRSM-L indicates that the 8192-entry line buffer and the 8192-entry counter buffer are adequately large. They overflow infrequently and, hence, trigger the establishment of checkpoints only infrequently. Based on the number of bits of storage, the size of the combination of the line buffer and the counter buffer is close to the size of the 2nd-level cache.

Table 3 shows statistics about the rate at which DRSM establishes checkpoints. The checkpoints in table 2 are triggered by the expiration of the timer.

Table 4 shows statistics about the amount of data written into the line buffer and counter buffer. Each row has 3 consecutive numbers enclosed within parentheses. The 1st number is the number of entries written into the line buffer. The 2nd number is the number of entries written into the counter buffer. The 3rd number is the ratio of the 2nd number to the 1st number. This ratio is the optimum ratio, according to equation #1.

For DRSM-L with 8, 16, and 32 processors, the ratios represented by the 3rd numbers are concentrated in the range of [0.62, 0.94], [0.69, 0.89], and [0.61, 0.89], respectively, excluding 3 atypical extrema (i.e. 0.42, 0.41, and 0.41). That the ratios are concentrated in a somewhat narrow band over several different applications is opportune. We can then select the average ratio (according to a geometric average) to determine the relative sizes of the line buffer and counter buffer, and this average ratio shall yield good system performance across all the benchmarks. The geometric averages of the ratios within the bands of [0.62, 0.94], [0.69, 0.89], and [0.61, 0.89] are 0.79, 0.80, and 0.79, respectively. Our selected ratio of 1.0 -- ratio of 8192 entries in the counter buffer to 8192 entries in the line buffer -- is slightly larger than these 3 geometric averages.

C. Performance Impact of Checkpoints

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

A processor in DRSM also suffers a 3rd type of interference. During the establishment of a checkpoint, "P" converts much dirty data (in state EXCLUSIVE) in the 2nd-level cache into clean data (in state SHARED) by writing it back into main memory. After "P" resumes execution after establishing the checkpoint, "P" wastes time in submitting many upgrade requests to memory in order to convert clean data (which was dirty prior to the checkpoint) back into dirty data so that "P" can resume writing into it.

Table 5 shows the negative acknowledgments (NAKs) and upgrade misses generated by the base TCMP, the TCMP with DRSM, and the TCMP with DRSM-L. For each of the benchmarks, the 1st row shows the number of NAKs; the impact of the 2nd type of interference is the increase in NAKs over that of the base TCMP. The 2nd row shows the number of upgrade misses; the impact of the 3rd type of interference is the increase in upgrade misses over that of the base TCMP. (A processor in DRSM-L does not suffer the 3rd type of interference.) For DRSM, the large increase in the number of upgrade misses is a major reason that DRSM performs worse than DRSM-L.

D. DRSM Versus DRSM-L

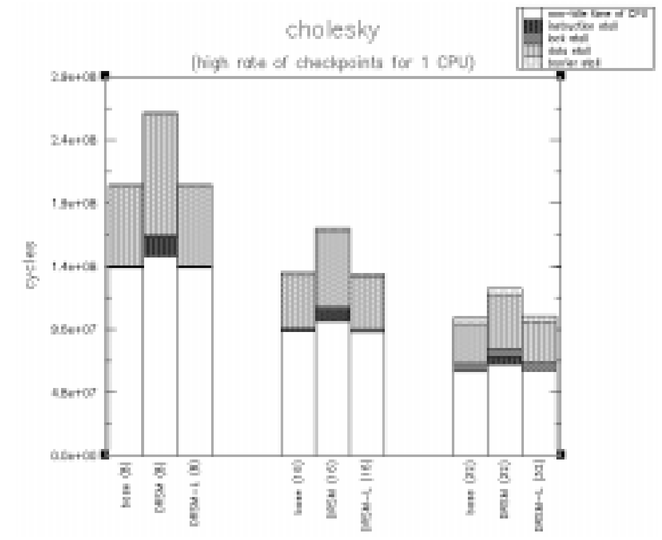

Figure 4. Effect of Irregular Checkpointing

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

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

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

Table 1. Timer-triggered Checkpoints: (number for processor #3; average for other processors)

We focus on Cholesky. Figure 4 shows the overall results. (For DRSM, expiration of a timer is effectively the only event that triggers establishing a checkpoint.) In figure 5, DRSM-L runs about 9.09%, 6.13%, or 4.77% faster than DRSM for a TCMP with 8, 16, or 32 processors, respectively. In figure 4, DRSM-L runs about 26.75%, 25.00%, or 19.28% faster than DRSM for a TCMP with 8, 16, or 32 processors, respectively. DRSM performs much worse then DRSM-L in figure 4. To obtain insight into the extent to which checkpoint dependencies cause a high rate of checkpointing by one processor to impact other processors, we introduce a lumped parameter that is the average number of timer-triggered checkpoints across all processors except processor #3. Table 1 shows the values for this new parameter. Each row has 2 sets of numbers. In each set, the 1st number is the number of timer-triggered checkpoints established by processor #3, and the 2nd number is the average number of timer-triggered checkpoints across all processors except processor #3. Clearly, due to the checkpoint dependencies that are in DRSM, the high rate of establishing checkpoints by processor #3 causes all the other processors to establish checkpoints at almost the same high rate. Hence, DRSM performs much worse than DRSM-L.

VIII. Related Work

Alewine proposes a read buffer that is similar to the line buffer. The read buffer saves the data read from a register file; after the processor encounters a fault, the register file restores its original state from the read buffer [1]. This technique provides fast roll-back but assumes that the read buffer is not affected by faults in another part of the processor.

Also, Janssens analyzes the performance of TSM-type and LSM-type algorithms in an insightful study [8]. It focuses exclusively on bus-based systems. By contrast, we focus on TCMPs created from more general networks, using directories to maintain the coherence of the caches.

The methods of establishing checkpoints in a TCMP are somewhat similar to methods in a loosely-coupled multiprocessor (LCMP) like a network of workstations, where software maintains the image of a single shared memory. An example of a TSM-type algorithm for a LCMP is another checkpointing scheme proposed by Wu[17]. An example of a LSM-type algorithm is the one proposed by Janakiraman[7]. The logging schemes proposed by Richard[10] and Suri[14] are examples of USM-type algorithms.

IX. Conclusion

We conclude that DRSM-L is a good checkpointing apparatus and algorithm for a TCMP. DRSM-L is the first USM-type algorithm for a TCMP. Unlike current algorithms, DRSM-L allows independent establishment of a checkpoint and independent roll-back from a fault and, hence, is much more scalable than DRSM. DRSM-L performs much better than DRSM. Also, DRSM-L is substantially cheaper to implement than DRSM. For example, DRSM-L requires only a single bank of memory, but both DRSM and RSM [3] require 2 banks of memory.

X. References

1. N. Alewine, S. Chen, et. al., “Compiler-Assisted Multiple Instruction Rollback Recovery Using a Read Buffer”, “IEEE Transactions on Computers”, vol. 44, no. 9, pp. 1096-1107, September 1995.
2. 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.
3. 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.
4. E. Bugnion, S. Devine, et. al., "Disco: running commodity operating systems on scalable multiprocessors", "ACM Transactions on Computer Systems", vol. 15, no. 4, pp. 412-447, November 1997.
5. S. Herrod, M. Rosenblum, et. al., "The SimOS Simulation Environment", Stanford University, pp. 1-31, February 1997.
6. 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.
7. G. Janakiraman and Y. Tamir, “Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers”, In Proceedings of the 13th Symposium on Reliable Distributed Systems, pp. 42-51, October 1994.
8. B. Janssens and W. K. Fuchs, “The Performance of Cache-Based Error Recovery in Multiprocessors”, “IEEE Transactions on Parallel and Distributed Systems”, vol. 5, no. 10, pp. 1033-1043, October 1994.
9. G. J. Narlikar and G. E. Blelloch, “Pthreads for Dynamic and Irregular Parallelism”, Proceedings of Supercomputing 98: High Performance Networking and Computing, November 1998.
10. 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.
11. D. Sunada, D. Glasco, M. Flynn, "ABSS v2.0: a SPARC Simulator", Proceedings of the Eighth Workshop on Synthesis and System Integration of Mixed Technologies , pp. 143 - 149, October 1998.
12. D. Sunada, D. Glasco, M. Flynn, "Hardware-assisted Algorithms for Checkpoints", technical report: csl-tr-98-756, Stanford University, pp. 1-38, July 1998.
13. D. Sunada, D. Glasco, M. Flynn, "Novel Checkpointing Algorithm for Fault Tolerance on a Tightly-Coupled Multiprocessor", technical report: csl-tr-99-776, Stanford University, pp. 1-50, January 1999.
14. 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.
15. 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.
16. 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.
17. K. Wu and W. Fuchs, “Recoverable Distributed Shared Virtual Memory”, “IEEE Transactions on Computers”, vol. 39, no. 4, pp. 460-469, April 1990.

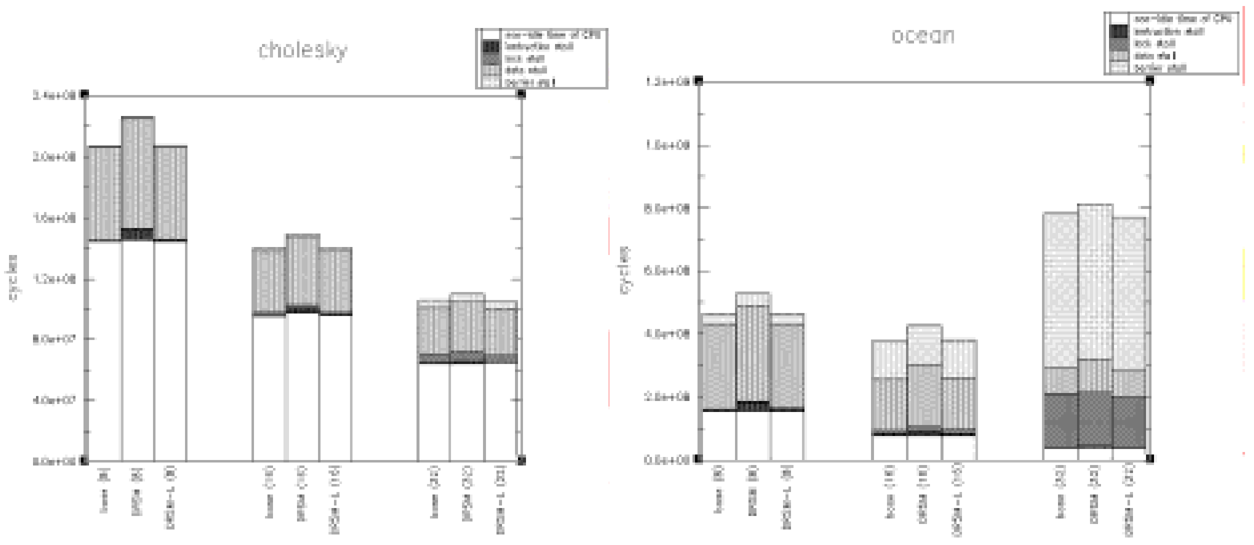

Figure 5. Performance of 2 Benchmarks

	8-processor TCMP	16-processor TCMP	32-processor TCMP	
Cholesky	10.38	7.00	5.19	checkpoints
	(9.62 + 0.12 + 0.62)	(6.94 + 0.00 + 0.06)	(5.03 + 0.12 + 0.03)	(please see text)
	(8.189 + 0.106 + 0.532)	(5.902 + 0.000 + 0.055)	(4.281 + 0.106 + 0.027)	x 1e+4 cycles
	(3.977 + 0.052 + 0.258)	(4.229 + 0.000 + 0.040)	(4.109 + 0.102 + 0.026)	x 0.01 % of run time
ocean	22.38	18.19	37.25	checkpoints
	(21.38 + 0.25 + 0.75)	(18.19 + 0.00 + 0.00)	(37.25 + 0.00 + 0.00)	(please see text)
	(18.186 + 0.213 + 0.682)	(15.474 + 0.000 + 0.000)	(31.692 + 0.000 + 0.000)	x 1e+4 cycles
	(3.903 + 0.046 + 0.146)	(4.045 + 0.000 + 0.000)	(4.098 + 0.000 + 0.000)	x 0.01 % of run time

Table 2. Checkpoints for DRSM-L

	8-processor TCMP	16-processor TCMP	32-processor TCMP	
Cholesky	11.00	8.00	6.00	checkpoints
	(7.530)	(3.447)	(2.039)	x 1e+6 cycles
	(3.353)	(2.327)	(1.868)	% of run time
ocean	24.00	20.00	39.00	checkpoints
	(31.227)	(17.650)	(10.910)	x 1e+6 cycles
	(5.871)	(4.117)	(1.338)	% of run time

Table 3. Checkpoints for DRSM

	8-processor TCMP	16-processor TCMP	32-processor TCMP
Cholesky	(35559; 28548; 0.80)	(24420; 16946; 0.69)	(18822; 11469; 0.61)
FFT	(8782; 6335; 0.72)	(4602; 3155; 0.69)	(2680; 1882; 0.70)
LU	(10571; 4475; 0.42)	(6794; 2813; 0.41)	(5293; 2147; 0.41)
ocean	(124516; 116525; 0.94)	(61962; 54849; 0.89)	(39862; 35659; 0.89)
radix	(8389; 5223; 0.62)	(12267; 10375; 0.85)	(10379; 8968; 0.86)
water	(4893; 3976; 0.81)	(5123; 4326; 0.84)	(4508; 3874; 0.86)

Table 4. Audit-Trail Data (entries in line buffer; entries in counter buffer; ratio)

	8 processors			16 processors			32 processors			
	base	DRSM	DRSM-L	base	DRSM	DRSM-L	base	DRSM	DRSM-L	
Cholesky	127	148	148	331	334	340	736	768	665	negative ack.'s
	9036	22662	9078	4954	9533	4938	3268	5447	3285	upgrade misses
ocean	6222	6404	6347	15936	16624	16480	50931	51009	50128	negative ack.'s
	41021	75475	40980	28386	61812	28430	14449	38453	14511	upgrade misses

Table 5. Negative Acknowledgments and Upgrade Misses for DRSM-L