FAULT TOLERANCE: METHODS OF ROLLBACK RECOVERY

Dwight Sunada
David Glasco
Michael Flynn

Technical Report: CSL-TR-97-718

March 1997

This research has been supported by a gift from Hewlett Packard, Inc.

Fault Tolerance: Methods of Rollback Recovery

Dwight Sunada
David Glasco
Michael Flynn

Technical Report: CSL-TR-97-718

March 1997

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

This paper describes the latest methods of rollback recovery for fault-tolerant distributed shared memory (DSM) multiprocessors. This report discusses (1) the theoretical issues that rollback recovery addresses, (2) the 3 major classes of methods for recovery, and (3) the relative merits of each class.

Key Words and Phrases: fault tolerance, rollback recovery, distributed shared memory (DSM)

0. Introduction

The strategies for building a fault-tolerant distributed shared memory (DSM) multiprocessor fall into at least 2 broad categories: modular redundancy and rollback recovery. Within the latter category, there are 3 principal issues: detecting a fault, containing a fault, and rolling the necessary nodes of the system back to the last checkpoint in order to recover from the fault. This report does not deal with the first 2 issues and merely assumes that (1) the DSM system is fail-stop, where a node encountering a fault simply stops and (2) the system can detect that a node has stopped (i.e. has encountered a fault). This report does focus on the last issue: methods of rollback recovery. They can be partitioned into 3 principal methods for both software-based DSM and hardware-based DSM.

Before presenting the details of the 3 methods, this report illustrates the notion of a "fault" and presents statistics of faulty behavior for a commercial computer system. Then, the discussion moves to some of the theoretical concepts behind the 3 methods of rollback recovery. These concepts lead into a detailed discussion of the 3 principal methods of rollback recovery. Each method is illustrated by either (1) a description of the best algorithm that currently exists in the literature or (2) a description of an algorithm (contrived by the principal author of this report) that is an improved version of what currently exists. Finally, this report concludes with a comparison of all 3 methods.

I. Fault

A. Definitions

A fault in a computer system is any behavior that differs from the behavior specified by its technical design. [Gray] An example of a fault is the operating system (OS) encountering a software bug and halting. Another example is a microprocessor halting after a length of its polysilicon wiring cracks into 2 pieces.

One can classify a fault according to the context in which it occurs. Below are 6 possible classifications.

- software (computer program)
- hardware
- maintenance (adjusting head alignment of disc drive, replacing its air filter, etc.)
- operations (installing new software, restarting a faulty node, etc.)
- environment (external power, earthquake, etc.)
- process (everything else)

"Process" faults are those which do not fall into any of the other categories.

One can also categorize a fault according to whether it manifests itself to the user. A fault which is not tolerated by the computer system and hence is visible to the user is called a "fatal fault". Any fault in a chain of faults leading to a fatal fault is called an "implicated fault".

An example of such a chain of faults is the following. A software bug in the OS causes one processor node in a distributed system to temporarily halt, creating a software fault. The system tolerates the fault by transferring the processes on the failed node to a working node and resuming execution of those processes. Then, a human operator attempts to restart the failed node but mistakenly restarts a working node, creating an operations fault. The software fault and the operations fault are implicated faults, and the latter fault is also a fatal fault.

B. Statistics

Figure #1 lists the fault statistics of Tandem computer systems for 3 years: 1985, 1987, and 1989. Near the bottom of the figure, the table shows the total number of computer systems in current use by all customers for each of those years. A typical Tandem system has 4 processors, 12 disc drives, several hundred terminals and associated equipment for communications. The computer system can tolerate a single fault.

Reading the table is straightforward. For example, in 1989, a total of 29 fatal faults occurred among 9000 systems in current use by customers.

Figure #2 recasts the data from Figure #1 to concentrate on the software faults and hardware faults, lumping all other faults into the category "other faults". The graph shows the number of fatal faults per 1000 Tandem systems from 1985 to 1989. During this period, the number of hardware faults has decreased dramatically; hence, the reliability of hardware has increased. On the other hand, the number of software faults has remained relatively constant.

Therefore, as a percentage of all fatal faults, software faults have become dominant. Figure #3 confirms this observation by recasting the data from Figure #1 to focus on the percentages of software faults, hardware faults, and other faults.

Why has software become dominant? New products tend to have more bugs (or problems) than old products. They have been used (and tested) long enough by customers in the marketplace, so vendors have eliminated many bugs from the old products. From 1985 to 1989, the number of new software products (i.e. application programs) has greatly increased relative to the number of new hardware products. Hence, software faults have become dominant over hardware faults. Furthermore, since the creation of new software generally outpaces the construction of new hardware, the dominance of software faults will likely persist indefinitely.

II. "Theory" behind fault tolerance

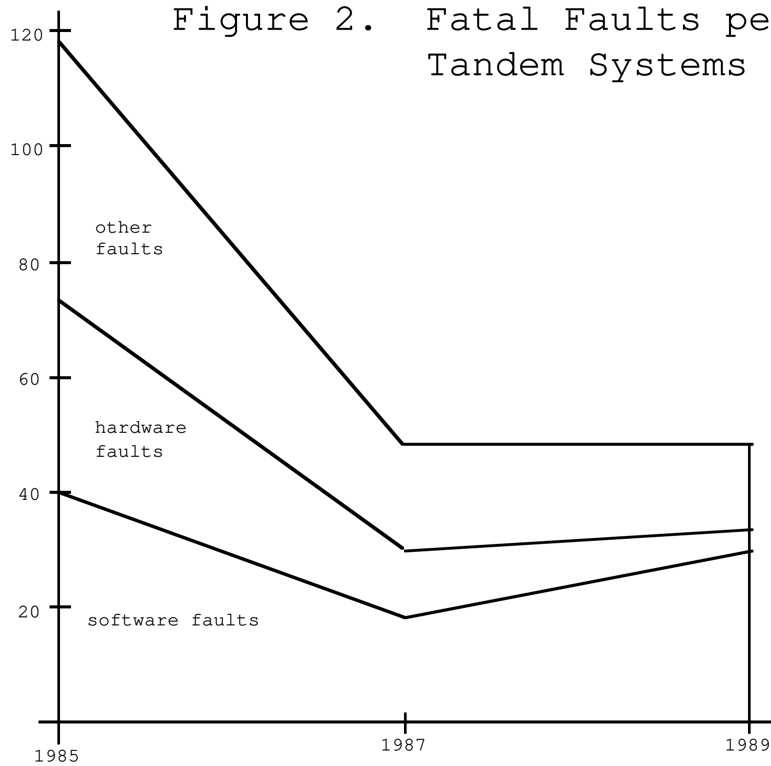
A multiprocessor system that is fault tolerant can (1) detect a fault, (2) contain it, and (3) recover from it. This report does not deal with the first 2 issues and assumes that each component in the system has the "fail-stop" property. Namely, if a component fails, then it simply stops. The rest of the system can detect that it has failed by simply noticing that it has stopped.

Most of the techniques for recovering from a fault in a multiprocessor system use a variation of rollback recovery. The system periodically establishes checkpoints for all processes. If the multiprocessor encounters a fault, the system rolls back to the last set of consistent checkpoints and resumes execution.

	Fatal Faults			Implicated Faults		All Faults
Year	1985	1987	1989	1987	1989	1989
Software	96	114	272	135	297	515
Hardware	82	66	29	106	77	157
Maintenance	53	37	22	42	28	28
Operations	25	35	66	49	86	27
Environment	17	28	26	37	27	103
Process	?	?	0	?	9	61
Unknown	12	14	23	17	23	21
Total	285	294	438	386	538	892
# of systems	2400	6000	9000			

Figure 1. Faults in Tandem Systems by Year [Gray]

Figure 2. Fatal Faults per 1000 Tandem Systems by Year



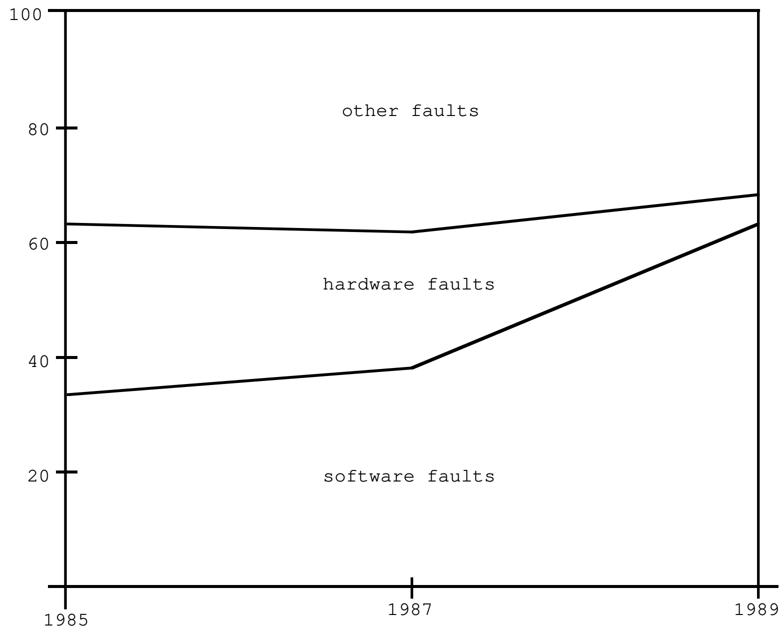


Figure 3. Percentages of Fatal Faults in Tandem Systems by Year

For the sake of clarity in illustrating the concepts of fault tolerance, this discussion assumes that each processor in the system executes exactly one process. Hence, referring to a process "P[0]" executing on processor "R[3]" is synonymous with referring to processor "R[3]". All processes comprise the execution of exactly one application program.

A. Message-passing multiprocessor

1. Consistent checkpoints

A checkpoint is the entire state of a process at a particular point in time. The system saves at least 1 checkpoint for each process. If a permanent fault occurs on a processor executing a process, the system reloads the process from its checkpoint onto another processor and resumes execution from the checkpoint. If a transient fault occurs on a processor executing a process, then the system restarts the failed processor and uses it to resume execution of the process from its last checkpoint.

If the computer system is a uniprocessor, then establishing a periodic checkpoint and rolling back to it is rather straightforward. For example, the uniprocessor can save the state of its process at the end of each 1-minute interval. The system creates a tentative checkpoint, verifies that its creation is successful, and then converts it into a permanent checkpoint, erasing the previous permanent checkpoint. If the creation of the tentative checkpoint is unsuccessful, then the system discards it and resumes execution from the permanent checkpoint created in the prior 1-minute interval. Even in this simple system, 2-phase checkpointing is required. [Koo]

If the computer system is a message-passing multiprocessor, then establishing periodic checkpoints and rolling back to them is complicated. Figure #4 illustrates 1 of the problems. Process "P" establishes a checkpoint at "Rp", and process "Q" establishes a checkpoint at "Rq". In the figure, "P" sends a message to "Q" before "P" encounters a fault. "P" rolls back to checkpoint "Rp" and resumes execution. Immediately after the rollback, the state of the system is inconsistent. Process "Q" has received a message, but the state of process "P" indicates that it has not yet sent the message.

By contrast, Figure #5 illustrates a state which is consistent. "P" receives a message from "Q" before "P" encounters a fault. "P" rolls back to checkpoint "Rp" and resumes execution. Immediately after the rollback, the state of the system is consistent although process "Q" has sent a message that process "P" never receives.

Although the state suggested in Figure #5 is consistent, the state of the system may not be acceptable. Suppose that both processes use synchronous message-passing to communicate. Unless the rollback recovery logs the messages and replays them, process "P" may hang forever. In other words, the system can be in 1 of 3 states:

inconsistent state,

unacceptable consistent state,

and acceptable consistent state.

The aim of fault tolerance is to set the system in an acceptable consistent state.

2. Domino effect

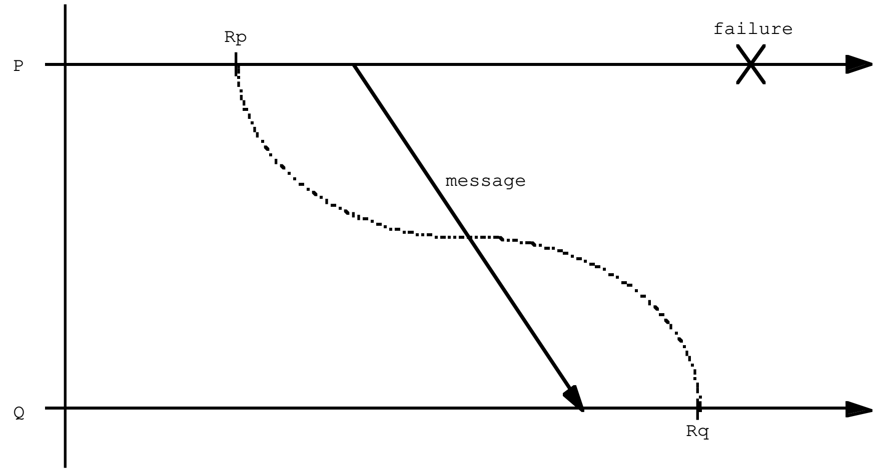


Figure 4. Inconsistent Checkpoints

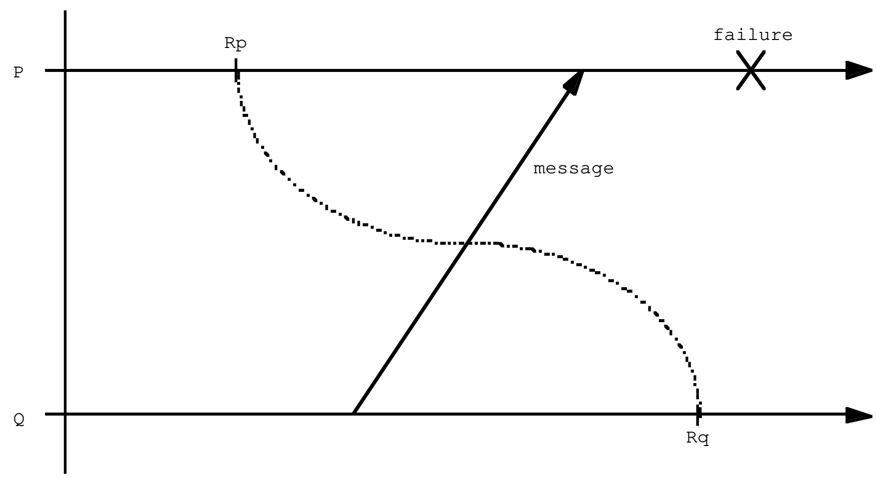


Figure 5. Consistent Checkpoints

The processors in a multiprocessor system either (1) can perform asynchronous checkpointing where each process independently establishes a checkpoint or (2) can perform synchronous checkpointing where processes synchronize to establish a consistent checkpoint. In asynchronous checkpointing, each process may need to establish and maintain many checkpoints in order to enable the computer system to find a set of consistent checkpoints during rollback recovery. In the worst case, the system may need to roll back to the state at the start of the execution of the application program. This phenomenon is called the "domino effect".

Figure #6 illustrates it. Process "P" encounters a fault and rolls back to checkpoint "Rp3", but rolling "P" back to "Rp3" forces the system to roll process "Q" back to "Rq2" in an attempt to find a consistent checkpoint. Still, rolling "Q" back to "Rq2" forces the system to roll "P" even further back to "Rp2". The events cascade until the both processes are rolled back to their initial states at "Rp0" and "Rq0".

3. Livelock effect

The processors in a multiprocessor either (1) can asynchronously roll back affected processes to a set of consistent checkpoints or (2) can synchronously roll back affected processes to a set of consistent checkpoints. In asynchronous rollback, a group of processes can cause each other to repeatedly rollback to the same set of checkpoints and thus can effectively halt the progress of the application process. This phenomenon is called "livelock". [Koo]

Figure #7 illustrates it. Process "P" and process "Q" recover asynchronously. "P" sends message "M1" to "Q", encounters a fault, and then rolls back to checkpoint "Rp". "Q" sends "N1" to "P" and then receives "M1". Since "P" has rolled back to "Rp", the state of "P" indicates that it has not sent "M1". Hence, "Q" must roll back to checkpoint "Rq".

Figure #8 illustrates the succeeding sequence of events. After "P" resumes execution from "Rp", "P" sends "M2" to "Q" and then receives "N1". Since "Q" has rolled back to "Rq", the state of "Q" indicates that it has not sent "N1". Hence, "P" must roll back to checkpoint "Rp", again. This sequence of interlocked events can proceed indefinitely, inhibiting the application program from progressing.

4. Interrupt

One important type of message is an interrupt from the environment (e.g. peripherals) to the multiprocessor. If an interrupt arrives from a device that does not participate in the fault-tolerant scheme of the multiprocessor, it must perform a checkpoint immediately after the receipt of the interrupt. The aim is to avoid losing knowledge of the interrupt. Otherwise, if the system does not perform this immediate checkpoint but does encounter a fault, the system will roll back to a checkpoint prior to the receipt of the interrupt and will lose knowledge of it.

Further, if the multiprocessor system sends information to a device that cannot tolerate duplicate information, then the system must perform a checkpoint immediately after the transmission of information. The aim is to avoid sending duplicate information. Otherwise, if the system does not perform this immediate checkpoint but does encounter a fault, the system will roll back to a checkpoint prior to the transmission of the information and may possibly resend it.

B. Distributed shared memory (DSM) multiprocessor

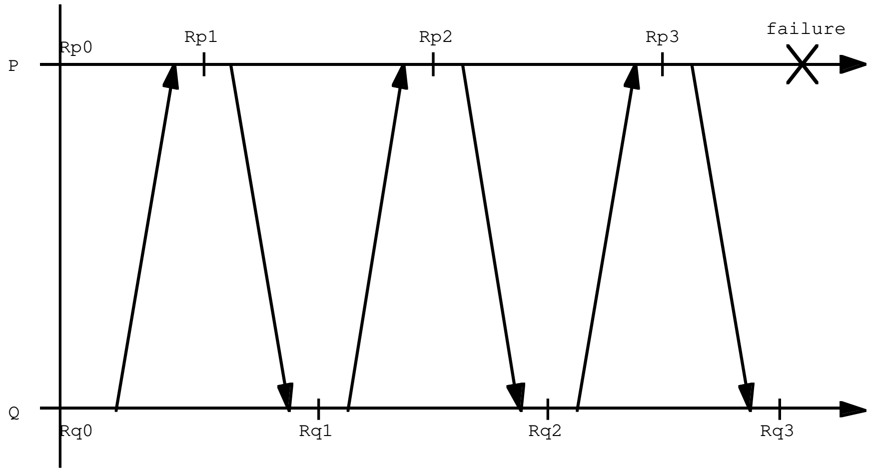


Figure 6. Domino Effect

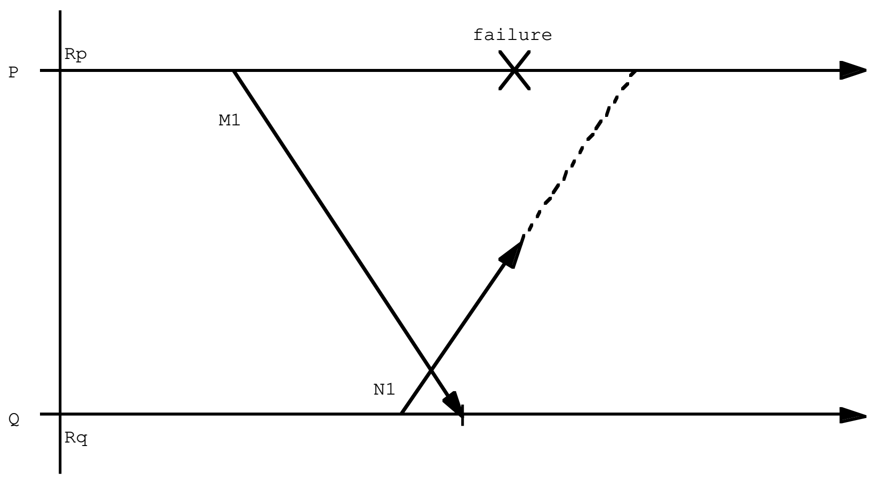


Figure 7. Livelock Effect [Koo]

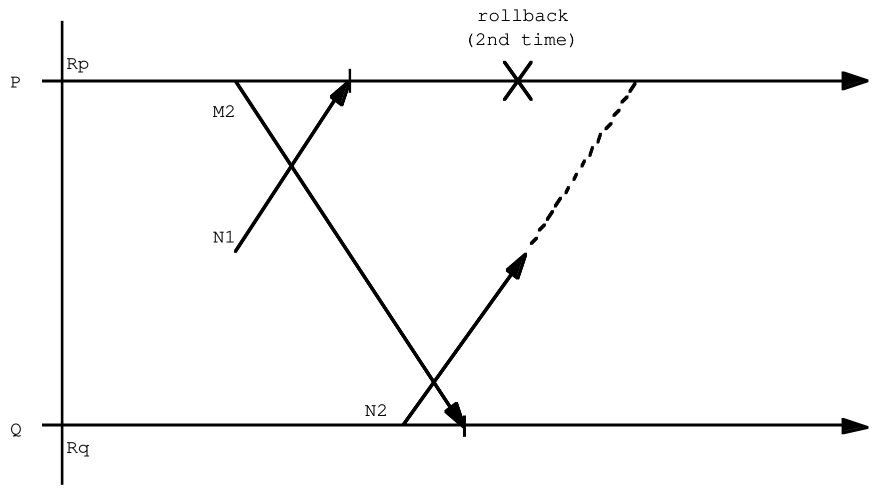


Figure 8. Livelock Effect [Koo]

1. Application of concepts from message passing

The concepts for fault tolerance in message-passing multiprocessors can be applied to DSM multiprocessors. Processes in the latter communicate by accessing shared memory. The mechanism for maintaining the coherence of shared memory generates messages on behalf of the communicating processes.

Figure #9 illustrates the coherence messages in a DSM multiprocessor. This particular system is a loosely-coupled one, where software (like that described by Li) maintains coherence. A tightly-coupled multiprocessor, where hardware maintains coherence, operates in a fashion that is similar to the operation of a loosely-coupled multiprocessor. Hence, the following discussion applies to both types of systems. [Janssens]

In the left half of the diagram, processor "P" attempts to write to block "G". "P" sends its request to "M", the manager for block "G". "M" forwards the request to the current owner, "Q". "Q" sends its copy of "G" and its copyset to "P". "P" receives "G" and sends invalidations to all processors listed in the copyset. In this case, processor "R" receives an invalidation.

In the right half of the diagram, processor "Q" attempts to read "G" and sends a request (for block "G") to processor "M". It, in turn, forwards the request to processor "P". Then, "P" forwards a copy of "G" to "Q".

The dashed line indicates that processor "Q" rolls back to checkpoint "Rq", which is just prior to where "Q" attempts to read block "G". If the messages in this DSM system are treated like those in the message-passing system, then rolling "Q" back to "Rq" requires that the system roll "P" back to checkpoint "Rp" in order to set the system in a consistent state.

In reality, the characteristics of the messages that are generated to maintain coherence enable the DSM system to roll "Q" back to "Rq" without rolling "P" back to "Rp". The message to request a copy of block "G" does not change the state of the processes on processors "P" and "Q". Although the state of the copyset may be inaccurate immediately after rolling "Q" back to "Rq", a slight modification of the coherence algorithm can easily fix that problem. Furthermore, the semantics of the algorithm do not cause "Q" to hang forever, waiting for the reply from "P", even though that reply is not logged. Hence, if the DSM system rolls "Q" back to "Rq" without rolling "P" back to "Rp", the system is in an acceptable consistent state.

2. Consistent global state in a theoretical DSM system

The previous discussion suggests that the interactions among processes in the DSM multiprocessor should be analyzed at the level of read and write accesses to shared memory. Analyzing the system at the lower level of messages is not useful since many messages issued to maintain the coherence of shared memory do not cause dependencies among processes.

Read and write accesses to shared memory introduce 2 kinds of dependencies. A recovery dependency exists from process "P" to process "Q" if rolling process "P" back to a checkpoint "Rp" requires rolling process "Q" back to a checkpoint "Rq" in order to maintain a consistent global state of the system. Similarly, a checkpoint dependency exists from process "P" to process "Q" exists if establishing a checkpoint of process "P" requires establishing a checkpoint of process "Q" in order to maintain a consistent state.

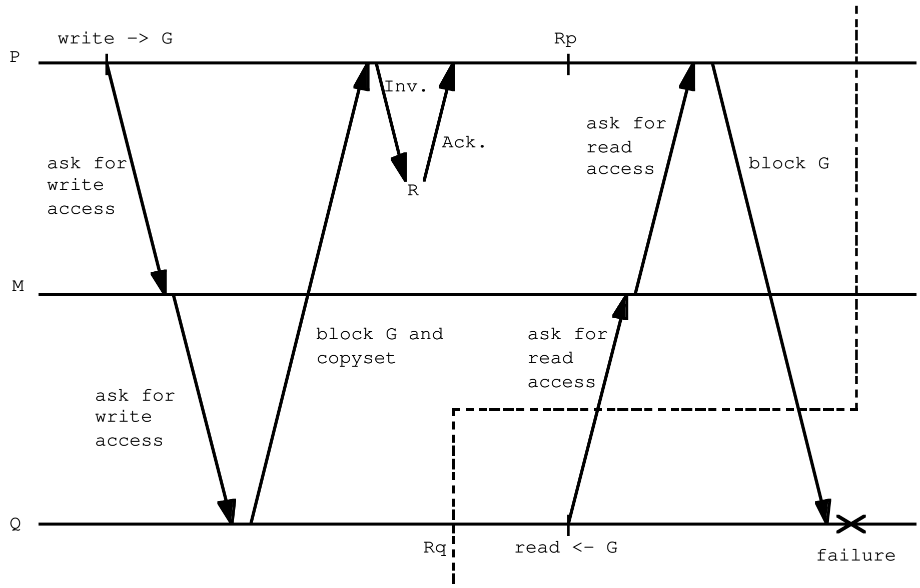


Figure 9. Messages in Distributed Shared Memory

A consistent state is one that is sequentially consistent. Sequential consistency is one type of view of memory accesses that the multiprocessor presents to the user. Memory-consistency models that are less stringent than sequential consistency also appear to the user to be sequentially consistent if the user protects accesses (to shared memory) within critical sections bounded by, for example, locks. For this report, a DSM system is assumed to provide a sequentially consistent view to the user.

There are 4 possible read/write interactions that exist between any 2 processes, "P" and "Q". Figure #10 illustrates a read-read interaction between process "P" and process "Q". The "checkpoint data" indicates data saved at the checkpoints "Rp" and "Rq". They constitute a consistent global state to which the system can roll back if it encounters a fault. In other words, they form a "recovery line". The "active data" indicates the data at a particular point in time. Both the "checkpoint data" and the "active data" designate the values that are stored in the same particular memory location. The interaction between the 2 processes occurs at that particular memory location.

In Figure #10, neither process affects the state of the other process. Hence, at time "t", the system can roll process "P" back to checkpoint "Rp" without rolling process "Q" back to checkpoint "Rq" and still maintain a consistent global state. The system can establish a new checkpoint of process "P" without establishing a new checkpoint of process "Q" and still maintain a consistent state. Similar comments apply to the situations where the roles of "Q" and "P" are switched.

Figure #11 illustrates a read-write interaction between process "P" and process "Q". Again, neither process affects the state of the other process. Hence, at time "t", the system can roll back a process to the last checkpoint or can establish a new checkpoint of the process without involving the other process.

Figure #12 illustrates a write-read interaction between process "P" and process "Q". "P" writes a value that "Q" subsequently reads, so "P" affects the state of "Q". At time "t", if the system rolls "P" back to checkpoint "Rp", the system must also roll "Q" back to checkpoint "Rq" in order to maintain a consistent state. On the other hand, the system can roll "Q" back to checkpoint "Rq" without involving "P". The system has sufficient information to determine that "0" from the checkpoint data should not be restored during a rollback of only "Q"; hence, the state remains sequentially consistent. Finally, at time "t", if the system establishes a checkpoint of process "Q", the system must also establish a checkpoint of process "P". The reason is that once a new checkpoint of "Q" is established, it cannot roll back to the previous checkpoint "Rq" even if "P" rolls back to "Rp". On the other hand, the system can establish a checkpoint of "P" without involving "Q". [Banatre]

In summary, the write-read interaction introduces the following dependencies.

```
   recovery dependency:  P -> Q
 checkpoint dependency:  Q -> P
```

Figure #13 illustrates a write-write interaction between process "P" and process "Q". "P" writes a value to a memory location to which "Q" subsequently writes a different value. At time "t", if the system rolls "Q" back to "Rq", then the system must also roll "P" back to "Rp" in order to maintain a consistent state. The reason is that once "Q" overwrites the value written by "P", the system cannot restore that value if "Q" rolls back to "Rq". (Morin explained, in private communication, that the system maintains only 2 copies of each memory location.) On the other hand, the system can roll "P" back to "Rp" without involving "Q". Finally, at time "t", if the system establishes a checkpoint of process "P", the system must also establish a checkpoint of process "Q" in order to maintain a consistent

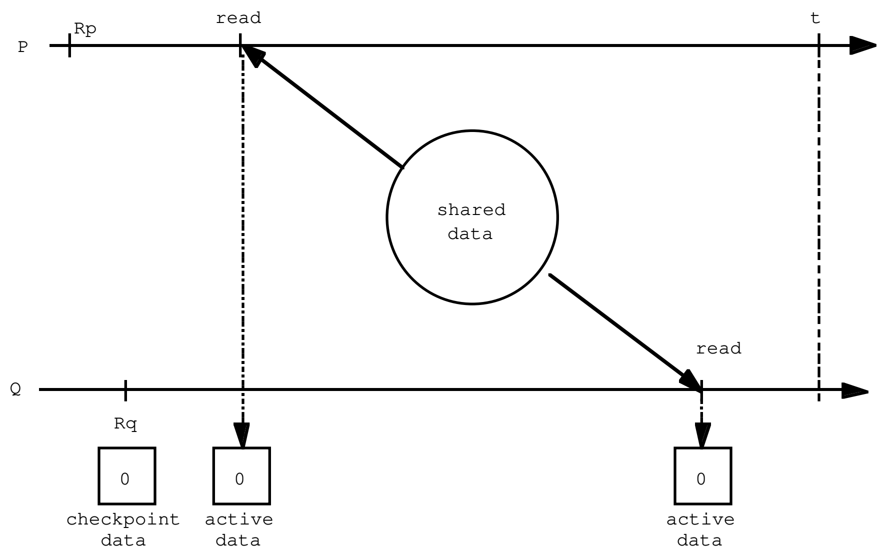


Figure 10. Read–Read “Message”

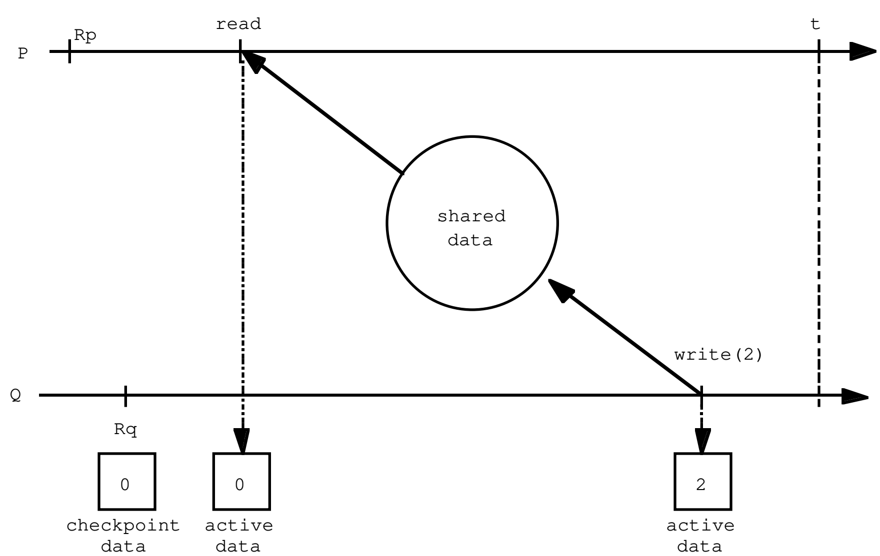


Figure 11. Read–Write “Message”

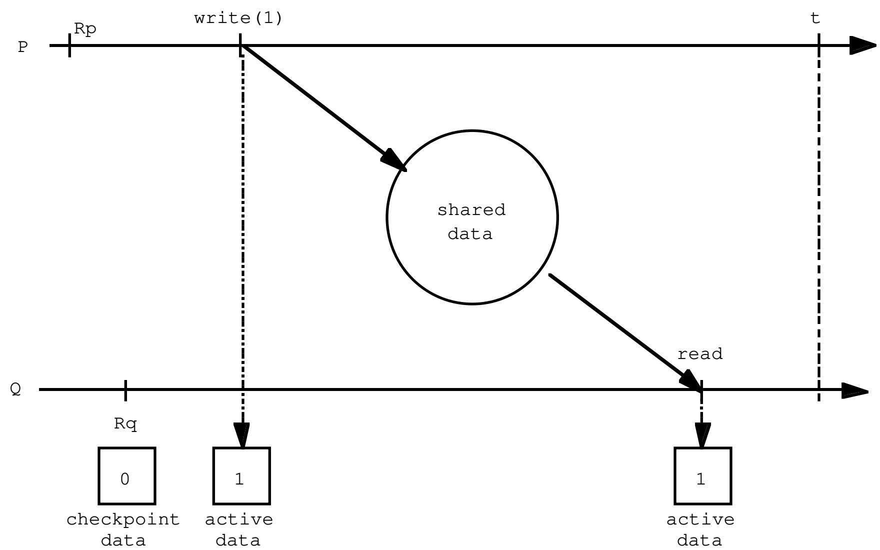


Figure 12. Write-Read “Message”

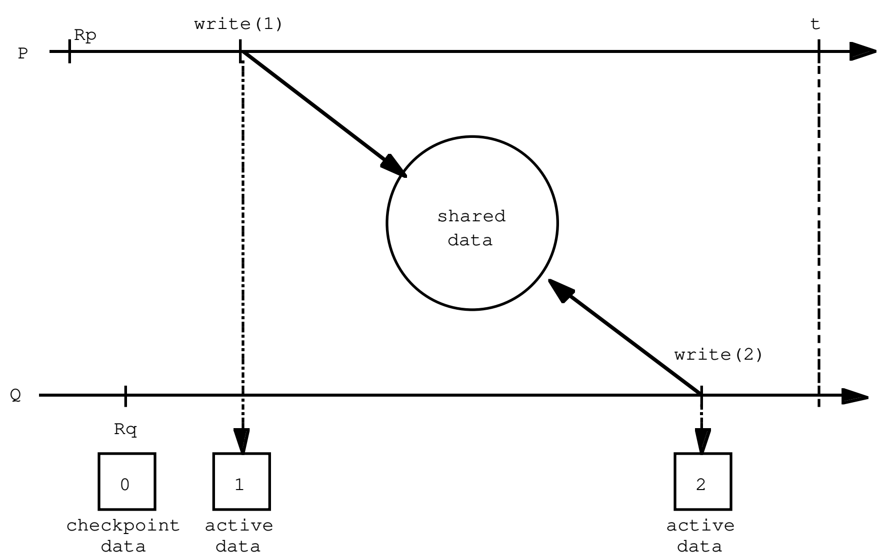


Figure 13. Write-Write “Message”

state. On the other hand, the system can establish a checkpoint of "Q" without involving "P".

In summary, the write-write interaction introduces the following dependencies.

```
  recovery dependency:  Q -> P
checkpoint dependency:  P -> Q
```

For the convenience of the reader, figure #14 summarizes the dependencies derived in this section.

3. Consistent global state in a real DSM system

Figure #12 and Figure #13 illustrate the dependencies that exist in the situation where memory is shared at the smallest granularity, a byte (8-bits) or a word (32-bits). In most real systems, the smallest granularity is the size of a cache block or virtual page of memory. The following discussion extends Banatre's work to include false sharing.

False sharing of the cache block introduces dependencies beyond the previously mentioned ones. Figure #15 illustrates 2 processes, "P" and "Q", falsely sharing a block of 2 bytes of data. When "Q" reads the value of "3", "Q" has no knowledge that "P" only wrote the value of "1" without altering "3". The state of the cache does not contain information to identify modifications to sections of a cache block. Hence, even though both processes are falsely sharing a cache block, they must behave as if they are genuinely sharing it.

For a write-read "message", we have the following dependency.

```
  recovery dependency:  P -> Q
checkpoint dependency:  Q -> P
```

At time "t", if "P" roll backs back to checkpoint "Rp0", "P" forces "Q" to roll back to checkpoint "Rq0". Establishing a checkpoint for "Q" at time "t" requires that TSM establish a checkpoint for "P" at time "t" since "Q" cannot roll past the value that "P" wrote.

In figure #16, "P" and "Q" write to different locations in the same cache block. Similar to the aforementioned argument, when "Q" writes to the cache block, "Q" has no knowledge that "P" only wrote the value of "1" without altering "3". Even though "P" and "Q" are falsely sharing a cache block, they must behave as if they are genuinely sharing it. Thus, this situation introduces the same dependency derived for a theoretical DSM system.

Furthermore, at time "t", if "P" encounters a fault and attempts to roll back to "Rp", neither the cache block "[0 3]" nor the cache block "[1 2]" can be part of a global consistent state of the system. The "0" in "[0 3]" is invalid since "Q" has already written "1" over the "0". The "1" in "[1 2]" is invalid since "P" rolls back past the point where it writes "1" into the cache block. Therefore, false sharing introduces an additional dependency. Namely, rolling "P" back to "Rp0" requires that "Q" roll back to "Rq0". Therefore, establishing a checkpoint of "Q" at time "t" requires that the system establish a checkpoint of "P" as well.

For a write-write "message", we have the following dependencies.

```
  recovery dependency:  Q -> P
                        P -> Q
checkpoint dependency:  P -> Q
                        Q -> P
```

Write-Read “Message”

```
checkpoint dependency:  Q -> P
  recovery dependency:  P -> Q
```

Write-Write “Message”

```
checkpoint dependency:  P -> Q
  recovery dependency:  Q -> P
```

Figure 14. Dependencies in Theoretical Systems

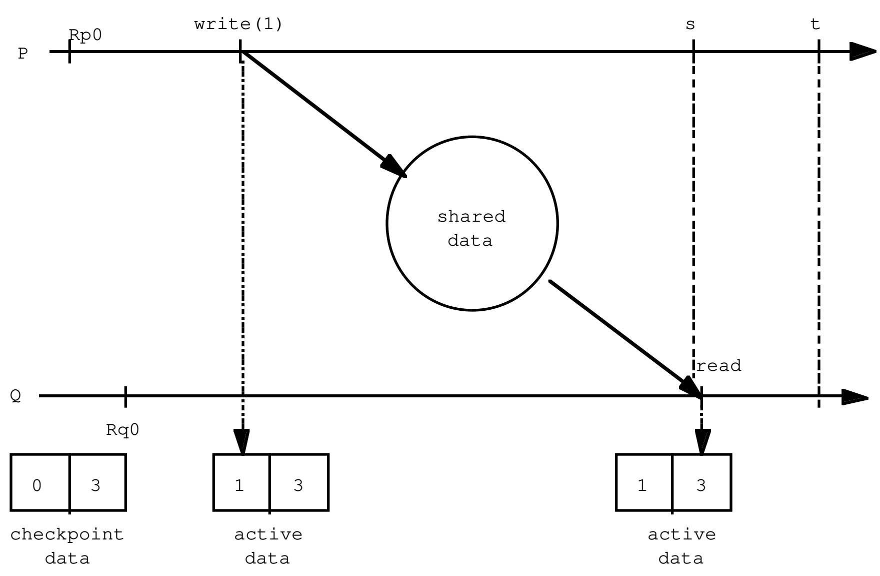


Figure 15. Write-Read “Message” on Block

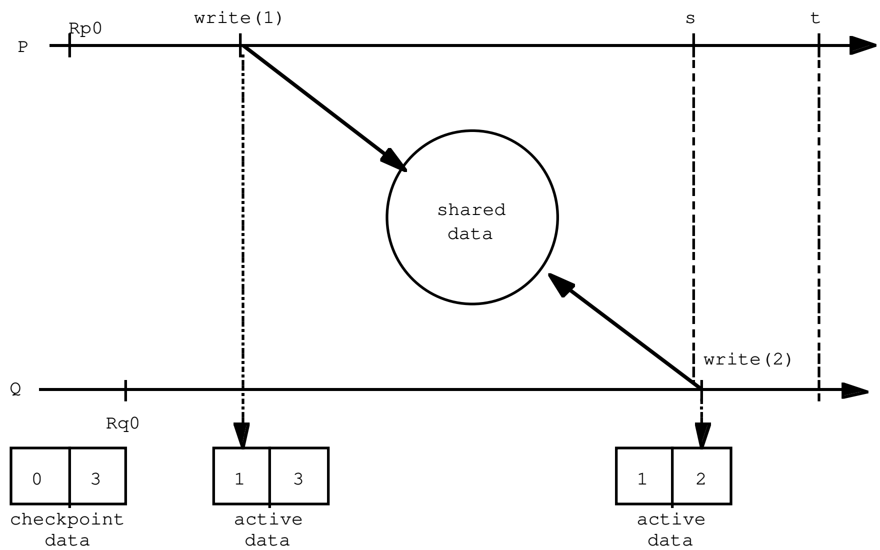


Figure 16. Write-Write “Message” on Block

At time "t", rolling a process back to its recovery point requires that the other process roll back to its own recovery point. Establishing a checkpoint of a process at time "t" requires that the system establish a checkpoint of the other process since it can not undo the value that it wrote.

For the convenience of the reader, figure #17 summarizes the dependencies derived in this section.

III. Taxonomy for methods of rollback recovery

Rollback-recovery schemes can be categorized into 3 principal methods: tightly synchronized method (TSM), loosely synchronized method (LSM), and un-synchronized method (USM). The aim of this naming convention is to convey the degree of strictness by which a particular method forces the establishment of a checkpoint. Under the TSM, a processor can immediately force the establishment of a checkpoint by another processor at the point of an interaction between the 2 of them. Under the LSM, a processor can force the establishment of a checkpoint by another processor, but the establishment of that checkpoint need not occur at the point of an interaction between the 2 of them. In other words, the checkpoint can be postponed (by recording the inter-processor dependencies that arose at the checkpoint). Under the USM, a checkpoint by a processor occurs independently from all other processors in the system.

IV. Tightly synchronized method for fault tolerance

The distinguishing feature of TSM is that a processor immediately establishes a checkpoint when an interaction between it and another processor can potentially cause rollback propagation.

A. General architecture

The most common implementation of a TSM is cache-aided rollback error recovery (CARER), originally proposed by Hunt for uniprocessors and extended to multiprocessors by Wu. Figure #18 shows the generic architecture of a system that uses TSM. There are 2 approaches: TSM for hardware-based DSM and TSM for software-based DSM. In the hardware-oriented approach, TSM assumes that both memory and cache are fault-tolerant and saves checkpoints in main memory. The system can tolerate only a transient fault in any node.

In a software-based DSM system, software maintains memory coherence by sending reliable messages over a local area network (LAN). In the software-oriented approach, the TSM assumes only that the disk and the LAN are fault-tolerant. TSM saves checkpoints to disk.

The following discussion focuses on the hardware approach.

B. Processor interaction

Figure #15 suggests the actions that TSM must take in order to prevent rollback propagation in the case of the write-read "message". Suppose that "Q" incurs a read miss in the process of fetching the cache block owned by "P". Just before it supplies that block, Write-Read "Message"

checkpoint dependency: Q -> P
recovery dependency: P -> Q

Write-Write "Message"

checkpoint dependency: Q <-> P
recovery dependency: P <-> Q

Figure 17. Dependencies in Real System

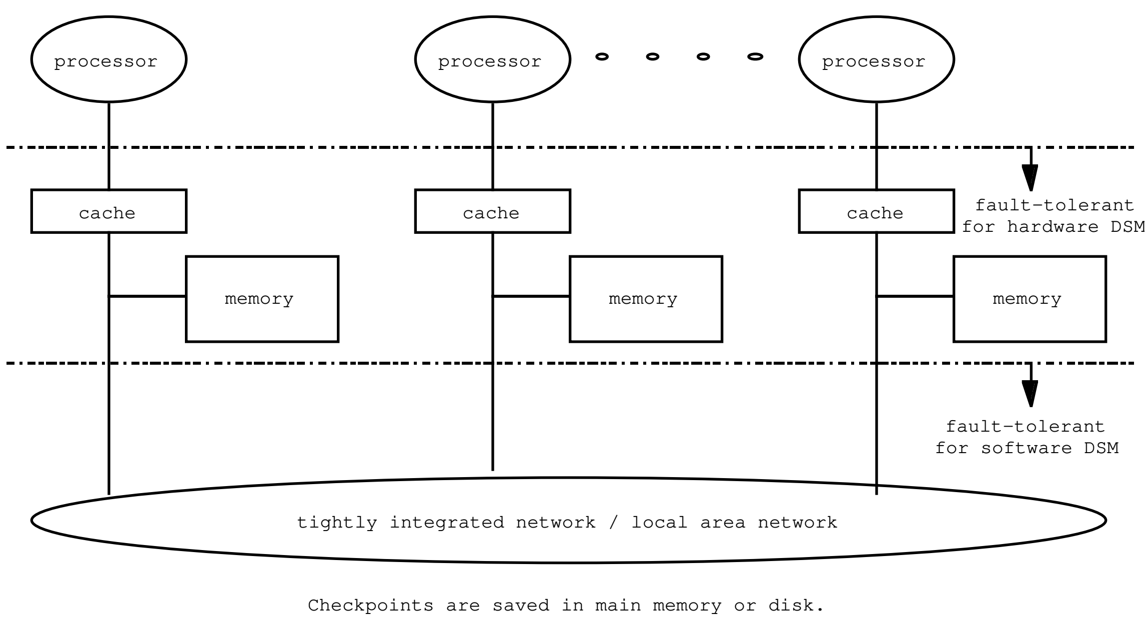


Figure 18. Generic Architecture in TSM

"P" must establish a checkpoint, "Rp1", at time "s". In this way, rolling "P" back to the last checkpoint, "Rp1", does not force "Q" to roll back to "Rq0". Figure #16 suggests the actions that TSM must take in order to prevent rollback propagation in the case of the write-write "message". Suppose that "Q" writes to the cache block. Just before "Q" performs the write, TSM must establish a checkpoint "Rp1" for "P" at time "s". In this way, rolling "P" back to "Rp1" does not force "Q" to roll back to "Rq0". Establishing "Rp1" also eliminates the rollback propagation suggested by dependency "Q -> P". Rolling "Q" back to "Rq0" restores "[1 3]" (saved at "Rp1") as the value of the cache block.

Incidentally, establishing the checkpoint of "P" in the situations illustrated in the 2 figures eliminates both the checkpoint dependency and the recovery dependency identified in figure #17.

C. Initiating checkpoints

Figure #19 illustrates 3 conditions requiring the TSM to establish a checkpoint for a process "P". First, suppose that any process reads or writes to a cache block for which process "P" is the last writer. The TSM must establish a checkpoint for "P" if the TSM has not already established one for "P" (thus having already saved that block) in order to prevent rollback propagation.

Second, suppose that the cache has reached a state where a dirty block must be written back to main memory in order to make room for an incoming cache block. A dirty block is one which is modified by the processor but which has not yet been written back to main memory. Since it contains the last checkpoint for process "P", the TSM must establish a new checkpoint for "P" in order to write the dirty block back to main memory. Simply writing the block back to memory without establishing a checkpoint would leave the original checkpoint (in memory) in a possibly inconsistent state. [Ahmed]

Third, any interaction between "P" and the environment requires that the TSM establish a checkpoint immediately after the interaction in order to prevent "P" from losing knowledge of that interaction. Examples of interactions include an interrupt from an external device, receiving data from it (via an I/O instruction), or sending data to it.

D. Saving checkpoints

To establish a permanent checkpoint, the TSM saves (1) the internal state (registers and internal buffers) of the processor executing "P" and (2) the dirty cache blocks. Afterwards, "P" resumes execution, updating the cache, registers, and internal buffers of the processor. [Wu] They essentially contain the tentative checkpoint of "P". At the next establishment of a permanent checkpoint, the TSM converts the tentative one into a permanent one. Fault tolerance based on rollback recovery requires that the system contain at least 2 checkpoints--a permanent one and a tentative one. For convenience, a permanent checkpoint is simply referenced as "checkpoint".

Frequently writing checkpoints to main memory can overload the interconnection network. Figure #20 illustrates an optimization to reduce the amount of data sent across the network. The figure shows the changes of the state of the cache during a read miss.

In addition to the usual storage to hold the cache blocks, the cache has a "c-id" (checkpoint identifier) field per block and also has a "c-count" register (per cache). When the processor inserts a new block into the cache or writes new data into a block, the TSM inserts the current value of "c-count" into "c-id". The TSM establishes a checkpoint of process "P" by

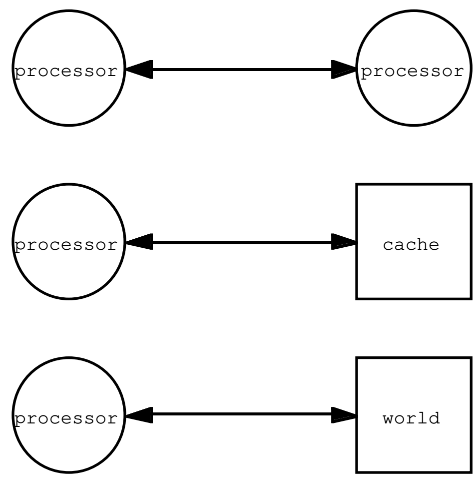


Figure 19. Initiating Checkpoints in TSM

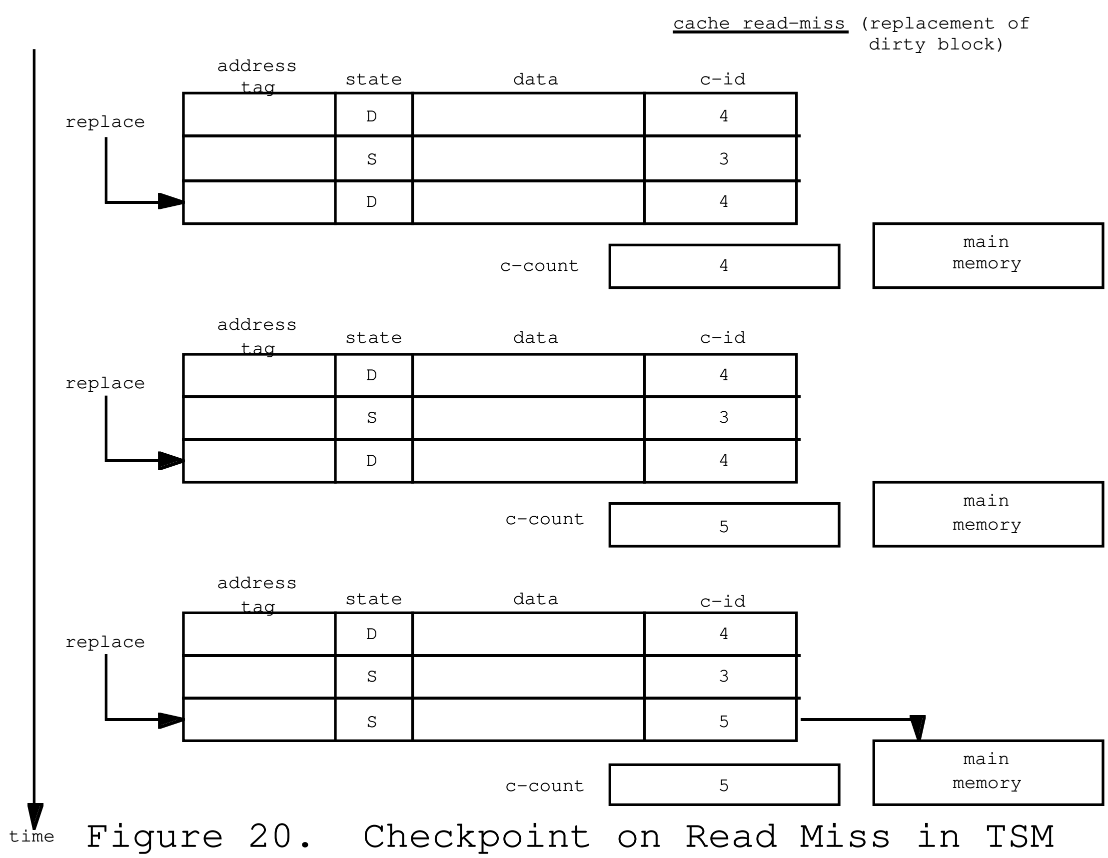


Figure 20. Checkpoint on Read Miss in TSM

merely incrementing "c-count" without writing the dirty blocks back to main memory. The system recognizes that a dirty block is part of a prior checkpoint if

```
c-id < c-count.
```

Dirty blocks are the only blocks that the TSM must write back to main memory in order to establish a checkpoint of "P".

Figure #20 shows the sequence of events in a checkpoint initiated by the processor displacing a dirty block. The processor must write a dirty block back to main memory in order to provide space for an incoming cache block. The TSM establishes a checkpoint by incrementing "c-count". The processor then writes the dirty block back into main memory and places the incoming block into the newly vacated space. All other dirty blocks that are part of the new checkpoint remain in the cache and do not load the interconnection network. Such dirty blocks are called "un-writable" as they are part of the checkpoint even though they remain in the cache.

Figure #21 shows the sequence of events in a write hit to an un-writable block. The TSM copies the block into main memory. The processor then writes new data into the block and inserts the value of "c-count" into "c-id". The block with the new data becomes part of the tentative checkpoint.

Figure #22 shows an optimization that obviates the need to write the block back into main memory in figure #21. The TSM merely copies the block into a recovery stack. At the next checkpoint, the TSM clears the contents of the recovery stack (by erasing it).

E. Recovery

The TSM recovers a node from a fault by discarding the tentative checkpoint. In other words, the TSM invalidates all entries in the cache for which "c-count" = "c-id" and re-loads the internal state (of the processor) from main memory. Each processor recovers independently from other processors.

F. Extension to software-based DSM systems

The principles of the TSM can be applied to software-based DSM systems. The main memory of each workstation operates like a huge cache of pages, and the operating system (OS) maintains the coherence of those pages. The OS sends coherence messages across the LAN.

The TSM establishes a checkpoint of a process "P" (with one process per workstation) whenever 1 of the following 3 conditions arises. (1) Another process reads or writes a page that was last written by "P". (2) "P" must write a dirty page back into the disk (which contains the virtual main memory of the process). (3) An interaction between "P" and the environment occurs.

Establishing a checkpoint and rolling a process back from a fault are straightforward. The TSM establishes a checkpoint of "P" by writing its state onto disk, including the dirty pages of "P" and the internal state of the processor executing "P". The TSM recovers a workstation from a fault by loading the state of its process from disk and by using simple algorithms to recover the data structures ("copyset" and "owner") used for memory coherence. [Wu]

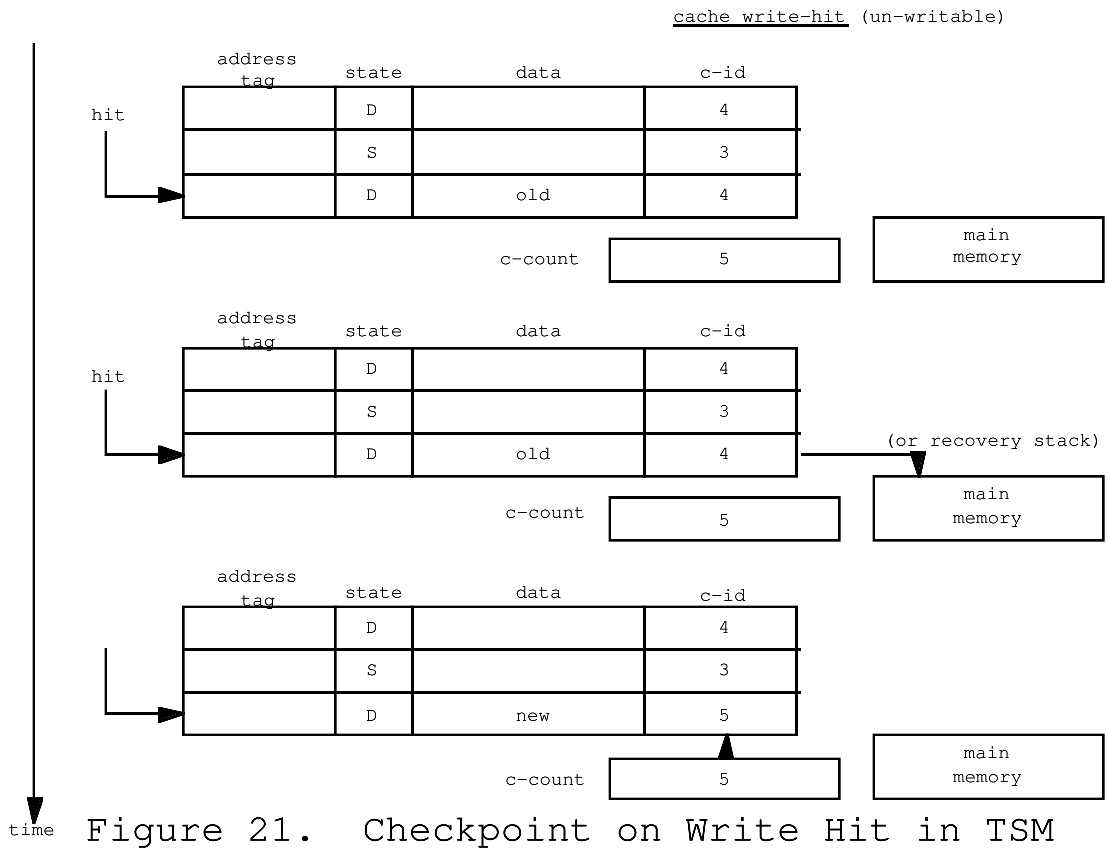


Figure 21. Checkpoint on Write Hit in TSM

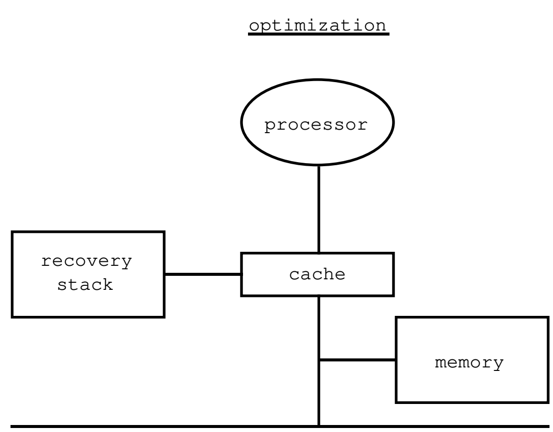


Figure 22. Recovery Stack in TSM [Wu]

The bottleneck in this system is clearly the disk. The slow and simple-minded approach is to designate a fixed area of the disk for storing the checkpoint and a separate area for storing the working ("W") pages. Upon recovery, TSM must copy the entire checkpoint into the area for the "W" pages.

Figure #23 illustrates an optimization to improve the responsiveness of the disk during rollback. The TSM maintains 2 pages, "X0" and "X1", on disk for each page, "X", in the virtual memory of the DSM system. "W" designates the tentative checkpoint to which the system occasionally writes dirty pages. "C" designates the disk page that is the current checkpoint. "I" designates the disk page which was a tentative checkpoint but which the TSM discarded after it rolled "P" back to its checkpoint. "O" designates the last checkpoint.

The disk pages of a system begin in either state "S0" or state "S4". When the processor writes a dirty page back into the virtual memory maintained on the disk, the processor writes the dirty page onto the "O" disk page, creating a "W" page. The processor writes further copies of the dirty page onto the "W" page. When a processor reads a virtual page from disk, it supplies both disk pages. The processor selects the "C" disk page if the other disk page is "I" or "O". Otherwise, the processor selects the "W" page.

If the TSM establishes a checkpoint, the TSM merely re-labels all the "W" pages as "C" pages and all the "C" pages as "O" pages. On a rollback, the TSM merely discards all the "W" pages, re-labeling them "I". Rollback is quick because it involves merely re-labeling the page that is discarded.

G. Bottleneck in TSM

One of the greatest drawbacks of the TSM is that the frequency of establishing checkpoints is highly dependent on the read-write patterns of the application running on the system. Whenever a remote process reads or writes a cache block (or page) that is written by the local process "P", it must establish a checkpoint in order to prevent rollback propagation.

What causes rollback propagation is the need to find a globally consistent state of the system. Figures #12, #13, #15, and #16 indicate that inconsistencies can potentially arise when processes access shared memory. Therefore, a way to reduce the frequency of checkpoints is to place constraints on the way in which processes can access shared memory. [Janssens]

Figure #24. illustrates such a constraint. "P" and "Q" access only shared blocks (memory) that are guarded by synchronization variables. In the figure, "P" reads synchronization variable "<sync>" to obtain entry into the critical region where shared access to block "<x>" occurs. Then, "P" accesses "<x>". "P" exits the critical region by writing to "<sync>". "Q" reads "<sync>" to enter the critical region, accesses "<x>", and then writes "<sync>" to exit the critical region.

If all processes access shared blocks by first obtaining permission via a synchronization variable, then the TSM does not need to establish a checkpoint of "P" when another process accesses the <u>non-synchronization</u> variables written by "P". The TSM can prevent rollback propagation by merely establishing a checkpoint when a process accesses a <u>synchronization</u> variable. Figure #24 illustrates 2 possibilities. (1) The TSM can establish a checkpoint when "P" writes "<sync>". (2) Alternatively, the TSM can establish a checkpoint when "Q" reads "<sync>". Variation (1) works with write-back or write-through cache coherence. Variation (2) only works with write-back cache coherence.

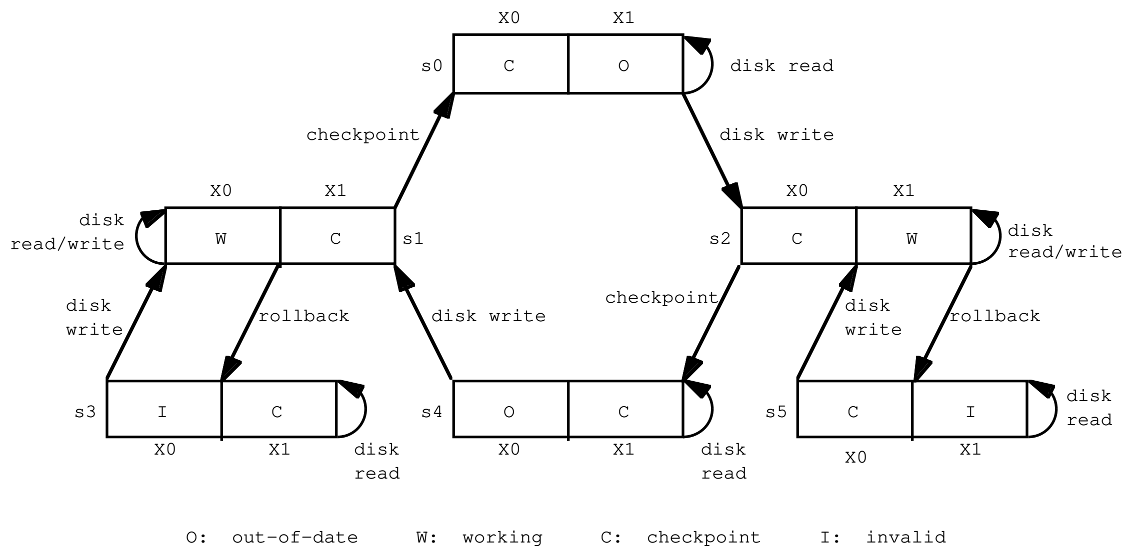


Figure 23. Fast Checkpoints on Disk In TSM

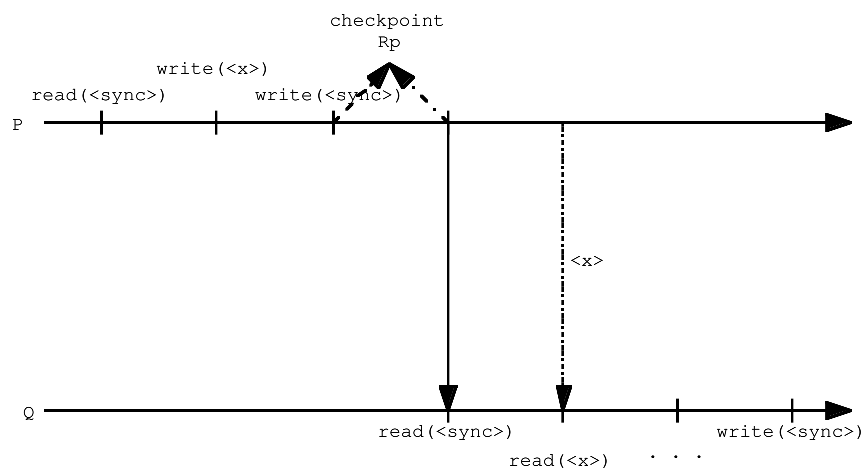


Figure 24. Optimization for TSM

H. Miscellaneous

For a hardware-based DSM, the TSM has several drawbacks. First, the frequency of establishing checkpoints is highly dependent on the read-write pattern of the application. A high frequency can significantly slow down the system. Second, the frequency of establishing checkpoints is also highly dependent on interactions with the environment, over which the TSM has no control. Third, the TSM requires expensive features: fault-tolerant memory and fault-tolerant cache. Finally, the TSM can only tolerate transient failure of a processor but cannot tolerate its permanent failure.

For a software-based DSM, the TSM does not have the 3rd and 4th drawbacks. The TSM requires only a fault-tolerant network and disk and can tolerate single-node permanent failure. Of course, a software-based DSM system runs slower than a hardware-based one.

TSM does have some advantages. First, it can be transparent to the application program. Only the hardware and the OS require modification. Second, if a processor fails, it recovers independently from the other processors. In other words, TSM experiences no rollback propagation and, hence, <u>may</u> cause minimal delay in recovering the system to its state just prior to the occurrence of a fault.

V. Loosely synchronized method for fault tolerance

Unlike the TSM, the LSM makes no attempt to immediately preclude any possibility of rollback propagation. Rather, the LSM records any interaction (between processors) that could lead to such propagation and uses the history of interactions to determine which processors need to simultaneously (1) establish a new checkpoint or (2) rollback to the last checkpoint in order to maintain a consistent global state. The LSM guarantees that the system will not roll back past the last permanent checkpoint.

A. General architecture

Figure #25 illustrates the general architecture on which the LSM is implemented. The figure is valid for both a software-based DSM system and a hardware-based DSM system. For both types of systems, the LSM assumes only that the interconnection network and the disk is fault-tolerant.

B. Processor interaction

Figure #17 summarizes the dependencies (arising from interactions between processors) that the LSM must record. LSM can record more dependencies than those listed in the figure, and the system will operate correctly. The drawback is that it will run slower than what it otherwise would run. For example, the LSM can record a 2-way dependency, "P <-> Q", for the checkpoint dependency caused by a write-read "message".

C. Hardware-based DSM system

1. Recoverable shared memory

Figure #26 illustrates the application of the LSM to a hardware-based DSM system. Memory is organized into multiple modules of recoverable shared memory (RSM), which is designed to be fault tolerant. Each module contains 2 banks of memory. "Bank #1" holds the tentative checkpoint of a block of memory, and "bank #2" holds the permanent

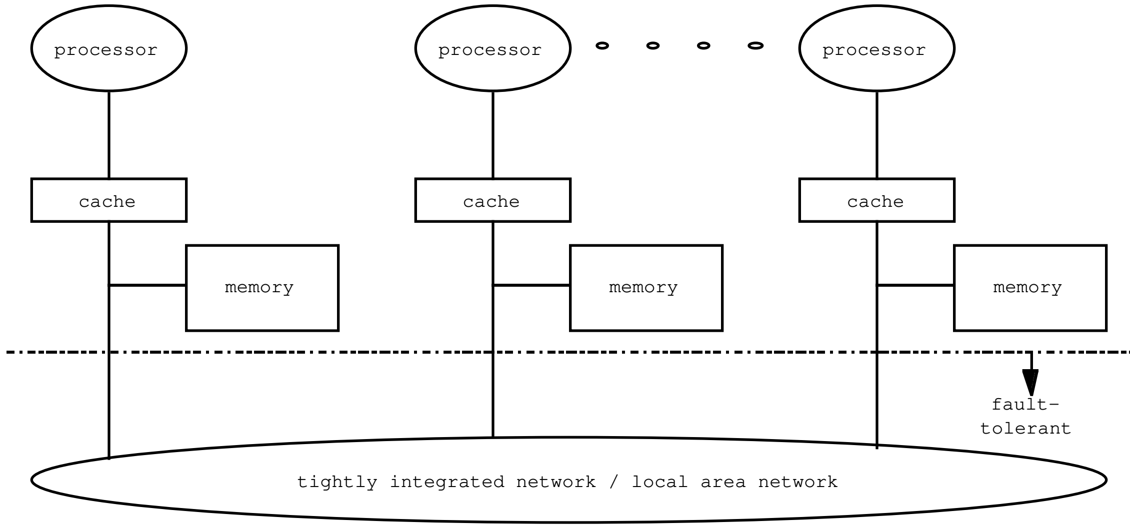


Figure 25. Generic Architecture

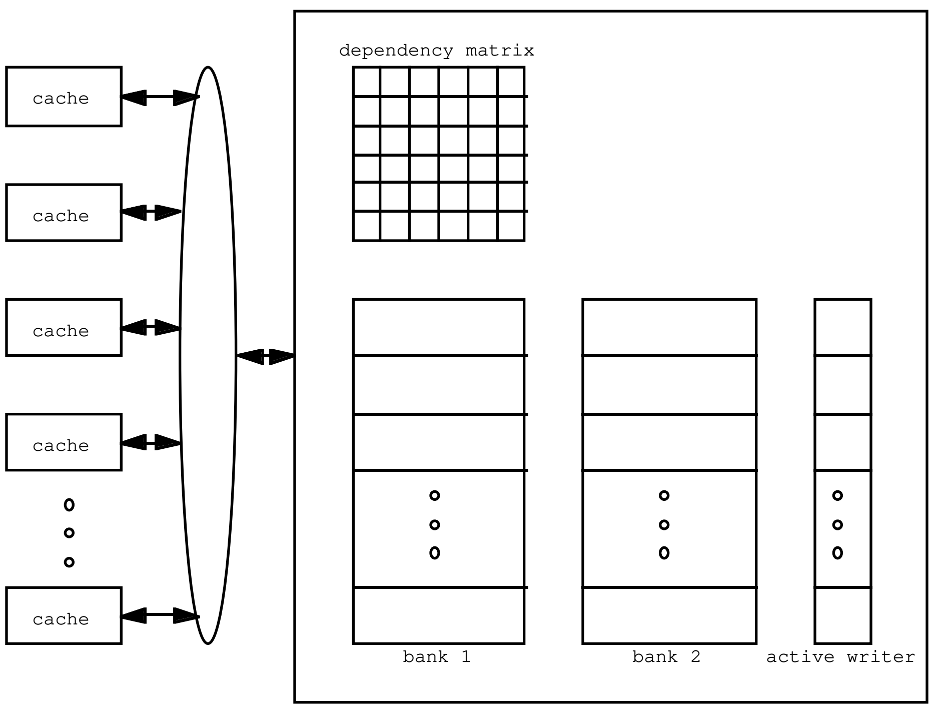


Figure 26. Recoverable Shared Memory

checkpoint. Associated with each block is a field indicating which processor is the last active writer. [Banatre]

Also, each RSM module contains a dependency matrix, "matrix[]". The LSM records the dependencies in "matrix[]". If the dependency "P[i] -> P[j]" arises, then the LSM sets element "matrix[i, j]" to 1. If "P[i] <-> P[j]" arises, then the LSM sets both the "matrix[i, j]" and "matrix[j, i]".

2. Dependency tracking

The LSM monitors messages from the caches in order to detect any dependency. There are 2 cases: cache coherence using a write-invalidate policy and cache coherence using a write-update policy. For a write-invalidate policy, a read-miss, a write-hit, and a write-miss generate messages that are sent onto the interconnection network. The LSM detects a write-read "message" (and the associated dependencies) when a read-miss follows a write-invalidate to the same memory block. The LSM detects a write-write "message" (and the associated dependencies) when 2 consecutive write-invalidate messages for the same memory block appear. [Banatre]

For a write-update policy, LSM can detect 2 consecutive writes by 2 different processors to the same memory block; hence LSM can detect write-write "messages". On the other hand, if processor "P0" updates a memory block that is cached by processor P1 which subsequently reads that memory block, the read-hit generates no message across the network, and LSM cannot detect a write-read "message". Therefore, in order to implement LSM with a write-update policy, LSM assumes that a processor caching a memory block being updated by another processor will surely read that memory block. In other words, LSM assumes that a write-read "message" will occur whenever a write-update occurs, thus overestimating the number of such "messages". (LSM can obtain the identity of all processors caching a particular memory block from the directory used for cache coherence.)

Figure #27 summarizes these conclusions.

3. Initiating checkpoints

Figure #28 shows the situation in which the LSM must establish a checkpoint of a processor. The LSM guarantees that the system will not roll back past the last permanent checkpoint. Since the LSM (unlike the TSM) does allow rollback propagation, an interaction between processors does not force the establishment of a checkpoint to prevent rollback propagation. Only an interaction between a processor and the environment forces the establishment of a checkpoint.

Of course, a processor "P" can arbitrarily choose to establish a checkpoint. In the absence of checkpoints induced by interaction with the environment, "P" should occasionally initiate a checkpoint. If "P" has not initiated a checkpoint in a long time and "P" encounters a fault causing it to roll back to its last checkpoint, "P" will suffer a long recovery time since the last checkpoint is far back in the past. "P" can use an internal timer to determine the temporal spacing between the establishment of checkpoints, thus limiting the duration of recovery.

4. Saving checkpoints

<u>Write-invalidate Policy</u>

okay

<u>Write-update Policy</u>

Assume write-read message.

Figure 27. Dependency Tracking

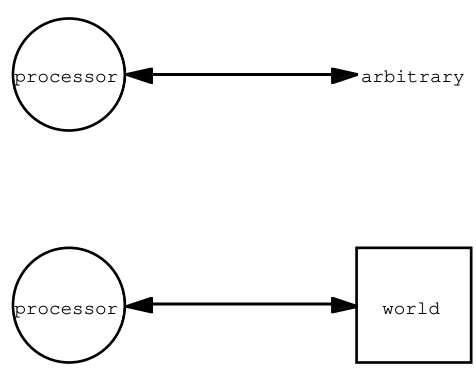


Figure 28. Initiating Checkpoints in LSM

In an actual implementation of the LSM, there will be multiple RSM modules to prevent a single module from being a bottleneck, but to simplify the following discussion, it assumes that the entire memory of the DSM system is contained in 1 module of RSM.

Once a processor, say, "P[7]", initiates a checkpoint, the LSM must perform 2 main operations: determining the dependency group of processors and saving their tentative checkpoints as permanent checkpoints. After "P[7]" informs the RSM to initiate a checkpoint, the RSM examines the dependency matrix to determine the group of all processors which have a checkpoint dependency on "P[7]". The dependency can arise transitively. For example, if both "matrix[7, 5]" and "matrix[5, 9]" (of the dependency matrix) are set to "1", indicating that "P[7] -> P[5]" and "P[5] -> P[9]" exist, then the dependency "P[7] -> P[9]" also exists.

After LSM determines the dependency group, all processors in that group save flush both their internal states (registers and internal buffers) and their dirty cache blocks into the tentative checkpoint. Then, LSM copies the tentative checkpoints of all processors (in this group) from bank #1 into bank #2. Finally, LSM clears "matrix[i, *]" and "matrix [*, i]" for all "i" such that "P[i]" is a member of the previously calculated dependency "group". ("*" is a "wild card" that represents all indices.) Establishing the checkpoints eliminates the dependencies.

5. Recovery

Once the LSM determines that a processor, say, "P[7]" has encountered a fault (and, hence, has failed), the LSM performs 2 main operations: determining the dependency group of processors and copying their permanent checkpoints from bank #2 back into bank #1. After the LSM identifies a failure at "P[7]", the LSM examines the dependency matrix to determine the group of all processors which have a recovery dependency on "P[7]". The LSM then copies the permanent checkpoints of all processors (in this dependency group) from bank #2 back into bank #1. the LSM loads the internal states of the recovering processors from the states that are saved in the checkpoint.

If "P[7]" fails transiently, then the LSM reboots it during recovery. "P[7]" resumes execution from the last checkpoint. If "P[7]" failed permanently, the LSM assigns the process (that was executing on "P[7]") to a working processor in the DSM system.

D. Software-based DSM system

1. Architecture

Figure #29 illustrates the architecture of a software-based DSM system implementing the LSM. The smallest granularity at which data is shared is a page of virtual memory whereas the smallest granularity in the hardware-based DSM is the cache block. Each page of memory has a manager node and an owner node. The manager maintains a directory structure that records the owner and copyset (the set of nodes with read-access to a page) of each page. Each node in the system has a page table indicating whether a page is in 1 of the following 3 states: shared, write, invalid. The system uses a write-invalidate policy.

The LSM tracks the dependencies (that arise from interactions between processors) by storing them in a data structure called the dependency group on each workstation. The LSM assumes the disk to be fault-tolerant and saves the checkpoints onto disk. Both the checkpoint pages and the working pages co-exist on the disk in the fashion indicated by figure #23.

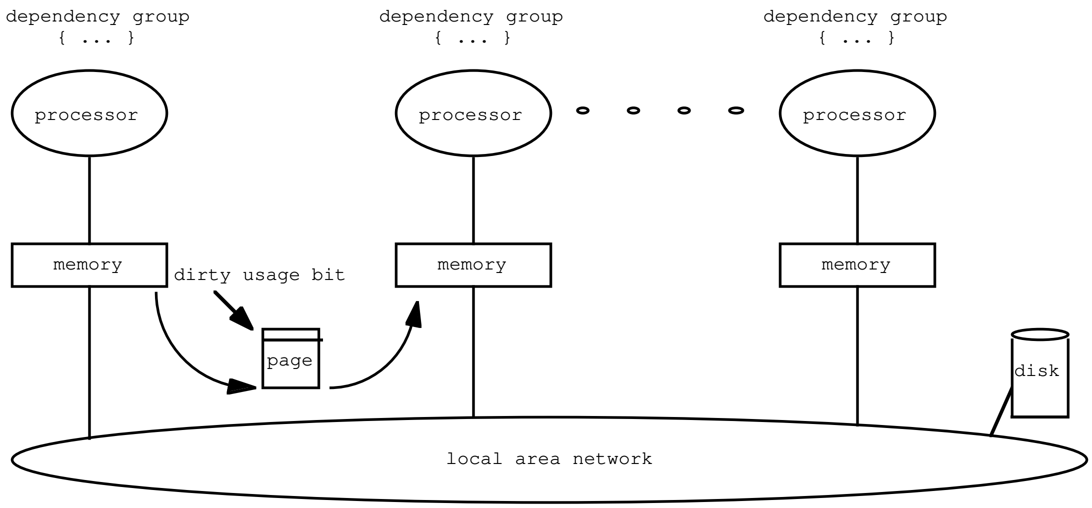


Figure 29. Architecture for Software-based DSM

2. Tracking dependencies

To simplify the implementation, the LSM records more dependencies than the minimum dependencies indicated in figure #17. Specifically, for both the checkpoint dependency and the recovery dependency of a write-read "message", the LSM records "Q <-> P". Consequently, a processor writing a page marks the start of a dependency for that page. To record the fact that a processor has written data into a page, the LSM needs only to associate a 1-bit flag with each page. Figure #29 shows that the flag is called the "dirty usage bit" (DUB).

In the LSM, the following events occur in the course of recording dependencies. Suppose that processor "P[2]" is the last writer to a page, "<X>", and sets DUB to "1". Then, "P[1]" reads data from or writes data to "<X>". "P[1]" examines the DUB. Since it is set to "1", there is a dependency between "P[1]" and "P[2]". "P[2]" stores both "1" and "2" in the recovery group. Also, "P[1]" stores both "1" and "2" in its recovery group. Both "P[1]" and "P[2]" are always aware of each other's identity. Dependencies occur only between the owner node (the last writer) and another node which is not necessarily the new owner. (Only owner nodes supply pages for read misses.)

3. Initiating checkpoints

The conditions for initiating checkpoints are identical to those conditions for a hardware-based DSM system.

4. Saving checkpoints

In the LSM, each processor initiating a checkpoint examines all other processors in its dependency group and requests them to initiate checkpoints. The initiation of a checkpoint propagates in a tree-like fashion from the initial processor, the root processor, that initiates a checkpoint. The processors in this tree comprise the recovery group.

Each processor, "P[i]", participating in this tree-like synchronization (1) flushes both its internal state (registers and internal buffers) and its dirty pages into the tentative checkpoint, (2) establishes a new permanent checkpoint on the disk, and (3) waits for child processors (in the tree) to establish a new permanent checkpoint on the disk. Once all child processors of "P[i]" report to it that they have established their new permanent checkpoints, "P[i]" tells its parent processor that it has established a new permanent checkpoint. A processor establishes a permanent checkpoint by using the technique illustrated in figure #23. The processor merely re-labels a tentative-checkpoint page (i.e. a working page) as a checkpoint page.

Each processor (in the group) resets the DUBs of its dirty pages to zero. If the construction of the tentative checkpoints fails, then the processors in the group restore each out-of-date checkpoint page as the checkpoint page and resume execution from the last permanent checkpoint.

After the root processor receives the message indicating that all processors in the recovery group have established their permanent checkpoints, the root processor sends a message telling its descendants to resume execution. This "resumption" message flows from the root down to the leaves of the tree (in the same way that the request to initiate a checkpoint flows).

5. Recovery

In the LSM, if a processor "P[1]" fails, each surviving processor in the system examines its dependency group. If "P[1]" is in the dependency group of "P[i]", then "P[i]" is a member of the recovery group of processors. All processors in the recovery group synchronize to invalidate their dirty pages and to restore them from the copies that are part of the last permanent checkpoint. The LSM re-labels the working pages (in figure #23) of the recovery group's processors as invalid pages; the LSM also re-labels the working pages of "P[1]". Each processor (in the group) loads its internal state from that saved with the last permanent checkpoint.

The re-labeling of pages requires the participation of all nodes in the DSM system. Determining the identity of the dirty pages owned by "P[1]" requires assistance from all other processors.

If "P[1]" fails transiently, then the LSM reboots "P[1]" and loads its state from the last permanent checkpoint. If "P[1]" failed permanently, then LSM assigns the process (that was executing on "P[1]") to another node in the system.

E. Miscellaneous

The LSM has several drawbacks. First, establishing checkpoints is highly dependent on interactions with the environment, over which the LSM has no control. Second, the LSM tends to require more memory than the TSM; LSM always maintains 2 copies of every block of memory. Finally, LSM suffers a limited degree of rollback propagation and, hence, <u>may</u> increase the time that the system needs to recover to its state just prior to a fault.

The LSM has several advantages. In a fault-free environment, the LSM <u>may</u> allow an application program to run faster than the TSM since the LSM appears to have a lower rate of establishing checkpoints than the TSM. Second, the LSM can tolerate either a transient failure or a permanent failure of a node. Finally, recovery is transparent to the application program. Unfortunately, recovery is not transparent to the OS as assigning a process (from the failed node) to a working node requires scheduling by the operating system.

F. Fault-tolerance through redundancy

The designer of the RSM (illustrated in figure #26) must design its hardware to be fault tolerant. This requirement is expensive but can alleviated by using the inherent redundancy of the DSM system to provide the needed fault tolerance. Figure #30 illustrates a 5-node system where 3 nodes maintain identical copies of the same data. The system will survive even if 2 nodes fail. In other words, figure #30 illustrates that the LSM can be implemented in such a way that the DSM system can survive failures of multiple nodes.

VI. Un-synchronized method for fault tolerance

Like the TSM, the USM prevents rollback propagation, but like the LSM, the USM does not establish a checkpoint in response to an interaction between 2 processors. The USM can establish a checkpoint of a process without involving other processes (on other processors). Further, the USM can roll a process back to its last checkpoint without involving other processes. The USM does not require synchronization of processes in either establishing a new checkpoint or rolling back a process to its last checkpoint.

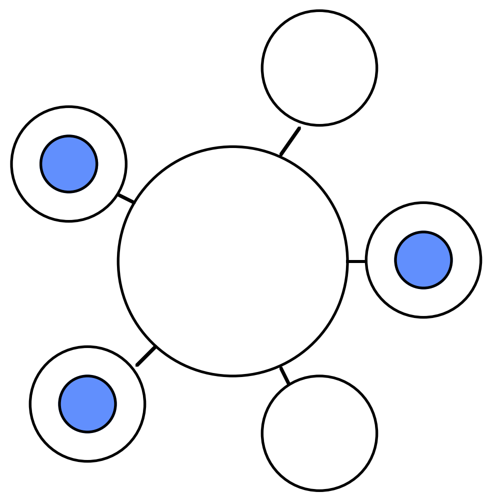


Figure 30. Fault Tolerance through Inherent Redundancy

The basic mechanism of the USM is the following. For each process, the USM records (in a log) the page of data that is received in response to an access miss. If the USM rolls a process back to its last checkpoint, the USM resumes execution of the process from that checkpoint. The USM uses the data in the log to satisfy access misses by the process until the log is exhausted. [Suri]

A. Architecture in the USM

Currently, the only systems that use the USM are software-based DSM systems like that shown in figure #31. The log of data received in response to an access miss is best stored on a disk.

B. Logging data and interactions for sequential consistency

Figure #15 suggests that a way to prevent the rollback of process "P" from propagating to "Q" is to guarantee that "P" after rollback writes the same value (into the page) that "P" wrote prior to rollback. This guarantee can be implemented by (1) recording (in a log) all pages of data supplied in response to access misses by "P" since the last checkpoint and (2) using the recorded pages (in the log) to respond to access misses by "P" after it rollbacks back to and resumes execution from the last checkpoint.

During rollback, the USM must determine which access should miss and hence should retrieve its page from the log. Figure #32 illustrates that only invalidations cause access misses. So, recording the point in "P" at which the invalidation occurs is sufficient to allow the USM to invalidate the appropriate page at the appropriate time during the execution of "P" after rollback. Thus, the USM forces the appropriate access encounter a miss.

Figure #33 illustrates that USM maintains both (1) an interaction log which records the point at which an invalidation occurs and (2) a data log which records data in each page that is received in response to an access miss. Each record in the interaction log contains 2 fields. One contains the number of accesses before an invalidation. The other field contains the page number which is invalidated. Each record in the data log contains data that is supplied for an access miss.

The interaction log in figure #33 is based on process "P" in figure #32. For example, the second record indicates that "P" performed 3 accesses between the first invalidation and the second invalidation after the last checkpoint, "Rp0".

The USM initially saves the logs of "P" in volatile storage. If "P" supplies a dirty page to another process, then the USM flushes the volatile log onto the disk just before "P" delivers the page. In order to ensure that the values written from the volatile log into the disk have not been corrupted by a fault, just before the USM flushes the log, the USM waits for a length of time such that the time since the receipt of the last page recorded in the volatile log is at least the fault-detection delay. In this way, the USM guarantees that "P" after recovery writes the same values that it wrote to the page before recovery.

C. Initiating checkpoints

Figure #28 illustrates the only situation in which the USM must establish a checkpoint of a process "P". The USM must establish a checkpoint of "P" whenever it interacts with the environment.

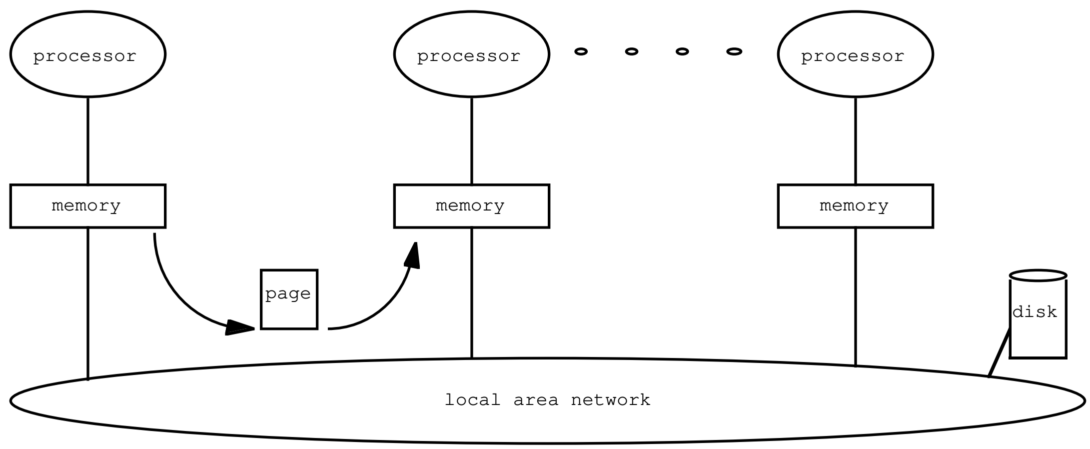


Figure 31. Architecture for Software-based DSM

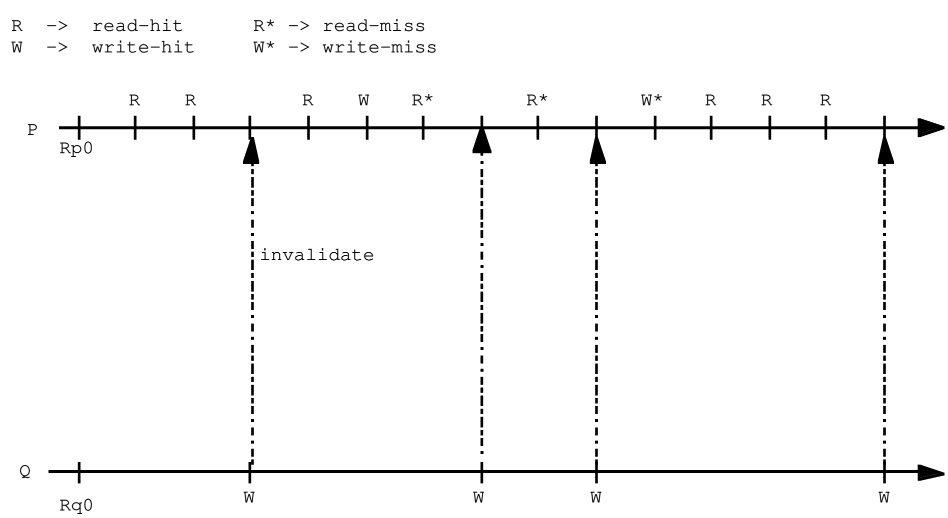


Figure 32. Nondeterministic Interactions
with Remote Processor

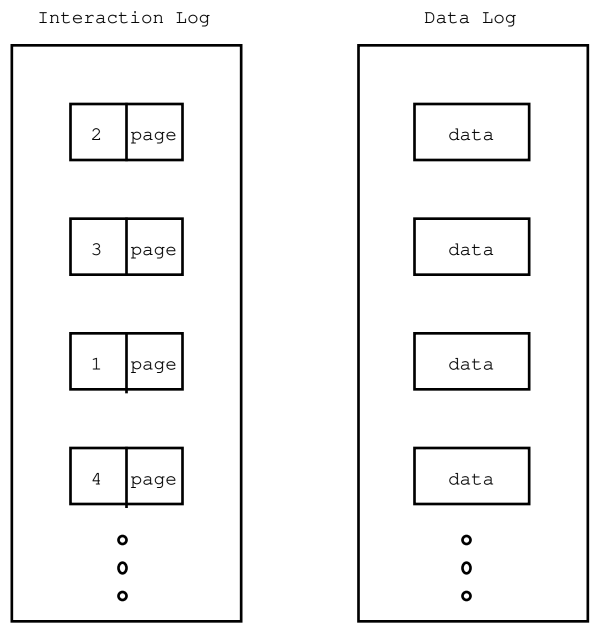


Figure 33. Logging Data and Interactions

Even if "P" does not interact with the environment for a long time (and hence the USM is not forced to establish a checkpoint of "P"), the USM should establish an occasional checkpoint of "P". Otherwise, if "P" encounters a fault a long time after it last established a checkpoint, "P" will require a long time to recover from that checkpoint. Also, if USM does not establish a checkpoint for a long time, both the data log and the interaction log will be very large.

D. Saving checkpoints

To establish a checkpoint of a processor "P", the USM flushes the volatile log onto disk. The USM then saves the internal state (registers and internal buffers) of "P" and its dirty pages into the tentative checkpoint. Then the USM re-labels the tentative checkpoint as the permanent checkpoint, according to the algorithm in figure #23.

The USM must necessarily save all the shared pages of "P", including the pages that have been invalidated. Suppose that the USM establishes a checkpoint of the process running on processor "P[3]" but does not save an invalidated page, "<5>", which has migrated from "P[3]" to "P[7]". If "P[3]" completes the establishment of its checkpoint but "P[7]" rolls back without saving "<5>" onto disk, then "<5>" will be lost.

After the USM establishes the checkpoint of "P", the USM erases both the data log and the interaction log from the disk.

E. Recovery

During recovery, the USM performs the following operations. Suppose that the USM rolls process "P" back to its last checkpoint. The USM executes "P" from that checkpoint and tracks the number of accesses that "P" performs. The USM uses the interaction log to determine the number of accesses that "P" can perform before the USM should deliver an invalidation to "P". The USM uses the data log to satisfy any access miss that "P" encounters.

If "P" resides on a processor that fails transiently, the USM reboots the processor and resumes execution of "P". If the processor fails permanently, then the USM assigns "P" to another node.

F. Optimization

A problem posed by the interaction log is that each access incurs the overhead of updating a counter that counts the number of accesses between invalidations. The USM can dispense with counting the number of such accesses if the application program only accesses shared data that is protected by synchronization variables. Most application programs do access shared data in that way.

Figure #34 illustrates how invalidations from remote processes like "Q" interact with "P". Invalidations arrive only outside the critical region, the section of code protected by synchronization variables. Since accesses to shared variables occur only after "read(<sync>)" (i.e. acquiring a synchronization variable and entering the critical region), "read(<sync>)" designates the precise point by which all invalidations must be delivered to shared pages of data. Hence, the only information that USM must record in the log is (1) the occurrence of "read(<sync>)" and (2) the page of data that is supplied for an access miss (which can only occur within a critical region).

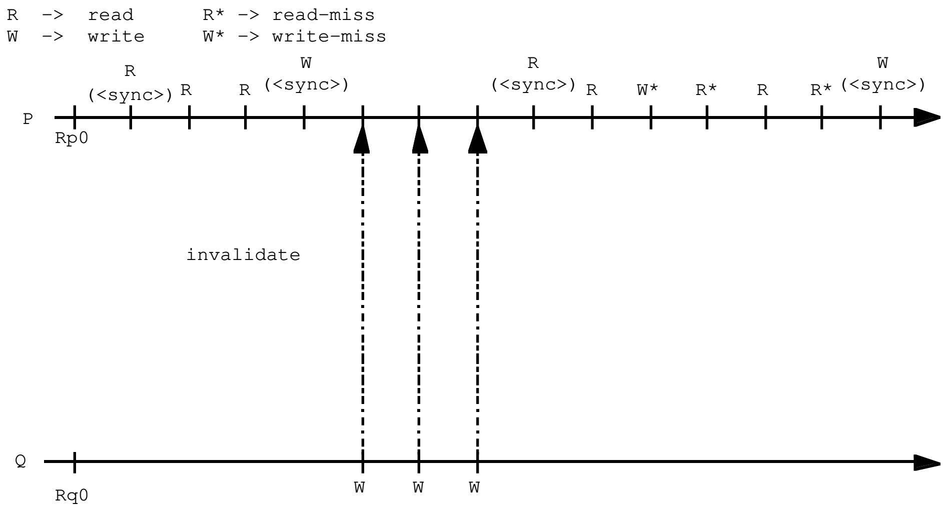


Figure 34. Constrained Nondeterministic Interactions with Remote Processor

Figure #35 illustrates the new structure of the log. All "read(<sync>)"'s since the last checkpoint are stored. Also, both (1) the page number of the access miss and (2) the data supplied for the miss are stored.

If a process "P" encounters a fault, the USM rolls "P" back to its last checkpoint and resumes the execution of "P". When it executes a "read(<sync>)" during recovery, the USM invalidates all the pages following the corresponding "read(<sync>)" recorded in the log. The USM uses the data for the invalidated pages recorded in the log to respond to the access misses.

G. Miscellaneous

USM has a significant drawback. Implementing the USM in hardware for a hardware-based DSM system is very difficult.

USM has several advantages. Each processor can establish a new checkpoint or roll back to the last checkpoint without involving any other processor although saving data from the volatile log into stable storage is synchronized with a remote processor. The system can tolerate permanent failures of multiple nodes. Finally, the system suffers no rollback propagation.

VII. Evaluation of TSM, LSM, and USM

A. Types of implementations

Figure #36 illustrates the 5 variations of the 3 principal methods of rollback recovery. The algorithms previously described for these 5 variations are the best algorithms that are currently available. Unfortunately, there is no implementation of the USM for a hardware-based DSM system. The reason is that the logs required for the USM are generally too large to be stored in main memory.

B. Comparison of methods for hardware-based DSM

Figure #37 compares the TSM and the LSM for various parameters. The column titled "ideal" lists what is considered ideal for each parameter.

In the ideal case, a fault-tolerant method can be implemented in such a way that neither the OS nor the application program must be modified. Ideally, an of-the-shelf OS like Windows NT that is not fault-tolerant can run on a DSM system that is made fault tolerant by only changes to the underlying hardware. Unfortunately, this goal cannot be achieved because (1) the OS must explicitly load the state of the processor from its last checkpoint for the TSM or (2) the OS must schedule a recovering process for execution for the LSM. In other words, the OS must recognize that the system is recovering from a fault and act appropriately. So, the OS must be modified. On the other hand, the mechanisms that implement fault tolerance are transparent to the application program.

In terms of hardware expense, the TSM is more expensive than the LSM since the former requires both fault-tolerant caches and fault-tolerant memory. The LSM requires only fault-tolerant memory. On the other hand, the TSM uses much less memory than the LSM since the former maintains a tentative checkpoint (in the cache) for each block of memory only if the block is dirty. The LSM always maintains both a tentative checkpoint and a permanent checkpoint for each block of memory.

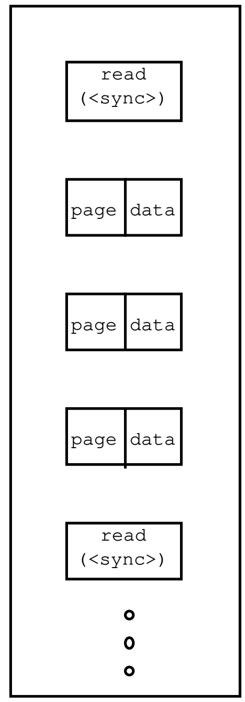


Figure 35. Logging For Constrained Interactions

	TSM	LSM	USM
hardware -based DSM	X	X	
software -based DSM	X	X	X

Figure 36. Methods of Rollback Recovery

	TSM	LSM	USM	ideal
transparency				
operating system	no (*)	no	???	yes
application program	yes	yes	???	yes
expense				
expense of hardware	high	medium	???	low
waste of memory	low	high	???	low
fault-toleration				
1-proc transient failure	yes	yes	???	yes
n-proc transient failure	yes	yes (*)	???	yes
1-proc permanent failure	no	yes	???	yes
n-proc permanent failure	no	yes (*)	???	yes
checkpoints				
non-environmental rate	high	low	???	low
environmental rate (*)	high	high	???	low
independence				
establishment of checkpoint	no	no	???	yes
recovery	yes	no	???	yes

Figure 37. Comparison of Hardware Methods

The TSM can tolerate the transient failure of a processor but cannot tolerate its permanent failure since loss of the cache can mean loss of the checkpoint. The LSM can tolerate permanent failures, but all the processors in the system must cooperate to repair the state of the directory for the memory blocks. The LSM previously described for the hardware-based DSM assumes that memory (i.e. RSM) is fault-tolerant. Without this assumption, the LSM must use the inherent redundancy of the processors to store multiple copies of each memory block in order to make memory fault-tolerant. In such a case, the number of copies of each block detects the degree of fault-tolerance; if only 2 copies are stored, then the LSM can only tolerate the failure of a single node.

As for checkpoints, the TSM may need to establish checkpoints at a high rate due to dependencies that arise from the interaction between 2 processors. By contrast, interactions between 2 processors do not require that the LSM establish a checkpoint of either processor, and the LSM can establish checkpoints at a low rate. Also, since neither the LSM nor the TSM have total control over the environment, it can force both of them to establish checkpoints at a high rate.

Under both the TSM and the LSM, a processor can force another processor to establish a checkpoint. Under the LSM, a processor rolls back to its last checkpoint can force another processor to rollback to its last checkpoint. Under the TSM, a processor performs a rollback independently from any other processor.

C. Comparison of methods for software-based DSM

Figure #38 compares the TSM, the LSM, and the USM for various parameters. The column titled "ideal" lists what is considered ideal for each parameter.

For all 3 methods, the OS must be changed to implement them. The reason is that the OS itself maintains the coherence of memory and hence tracks reads and writes into pages. On the other hand, the mechanisms that implement fault tolerance is transparent to the application program.

As for expense, the hardware expense is low for all 3 methods since the aim of software-based DSM is to create a multiprocessor using only software. Unfortunately, all methods consume a large amount of disk space in order to maintain 2 copies (both a tentative checkpoint and a permanent checkpoint) of each virtual page. Under the USM, the log also requires much space.

Also, all 3 methods can tolerate the permanent failures of multiple nodes.

As for checkpoints, the TSM may need to establish checkpoints at a high rate due to dependencies that arise from the interaction between 2 processors. By contrast, interactions between 2 processors do not require that the LSM establish a checkpoint of either processor, and the LSM can establish checkpoints at a low rate. Also, since neither the LSM nor the TSM have total control over the environment, it can force both of them to establish checkpoints at a high rate.

In the case of the USM, it also does not have total control of the environment and can suffer a high rate of establishing checkpoints in response to interactions with the environment. In addition, even though the USM need not establish a checkpoint in response to the interaction between 2 processors, an interaction where processor "P" supplies a dirty page to another processor forces the USM to copy the volatile log of "P" onto disk. Depending on the characteristics of the application program, "P" can be forced to save data (i.e. the volatile log) at a high rate onto the disk.

	TSM	LSM	USM	ideal
transparency				
operating system	no	no	no	yes
application program	yes	yes	yes	yes
expense				
expense of hardware	low	low	low	low
waste of disk space	high	high	high	low
fault-toleration				
1-proc transient failure	yes	yes	yes	yes
n-proc transient failure	yes	yes	yes	yes
1-proc permanent failure	yes	yes	yes	yes
n-proc permanent failure	yes	yes	yes	yes
checkpoints				
non-environmental rate	high	low	low (*)	low
environmental rate (*)	high	high	high	low
independence				
establishment of checkpoint	no	no	yes (*)	yes
recovery	no	no	yes	yes

Figure 38. Comparison of Software Methods

Under the TSM, all processors in the software-based DSM must synchronize in order to recover the copysets and owners of pages whereas such synchronization is not required in a hardware-based DSM. The reason is that the latter type of DSM has both fault-tolerant memory and fault-tolerant caches. The directory, which is the hardware equivalent of the copyset data structure and the owner data structure, is part of fault-tolerant memory and is never corrupted.

Under the LSM, multiple processors must synchronize in order to establish a new checkpoint or to rollback to the last checkpoint.

Under the USM, a process can establish a new checkpoint or rollback to the last checkpoint without involving any other processor. Even though a processor "P" can establish a checkpoint (by saving the state of the processor onto disk) independently from all other processors, another processor "Q" can force "P" to save its volatile log onto disk if "P" supplies a dirty page to "Q".

D. Performance

Currently, a broad-based comparison of the performance of all 3 methods does not exist. There are a few studies which compare the performance of the TSM with the performance of the LSM. Banatre conducts one such study and claims that the LSM allows a DSM system to run faster than the TSM.

A comparison of all 3 methods requires that they be examined in 2 contexts. First, in the absence of any fault, how fast does the DSM system run? Second, after the system encounters a fault, how long does the system require in order to recover from the fault? Figure #39 illustrates the 2 contexts.

1. Fault-free context

The most important parameter is the execution time of the application program. At least 2 factors can affect the execution time. One is the amount of redundant data that is saved (e.g., to disk) in order to allow the system to recover to a consistent state after the system encounters a fault. Both the number of events in which data is saved and the amount of data that is saved at each event determines the total amount of data that is saved.

The other factor that can affect the execution time is the percentage (of network traffic) that is attributed to this redundant data. All of the redundant data is not necessarily transferred across the network; for example, some of the redundant (checkpoint) data in the TSM is saved locally in the cache. Hence, the network traffic that is due to the redundant data can be viewed as a factor (affecting execution time) that is distinct from simply redundant data.

2. Faulty context

Once the system encounters a fault, the most important parameter is the time that the system requires to recover from the fault. The recovery time is highly dependent on the rate at which checkpoints are established; the recovery time is inversely proportional to the rate of checkpoints. For example, if the system established a checkpoint recently, then the recovery time is short.

An alternative measure of performance in the faulty context is the product of the recovery time and the rate of checkpoints. The aim of the DSM system is to achieve a same value for the product.

fault-free context

1. amount of redundant data (data for log or non-environmental checkpoint)
    a. rate of establishment of checkpoints
    b. amount of data transferred per checkpoint
    c. rate of logging
    d. amount of data transferred per "save" in log
2. percentage of network traffic due to redundant data (per processor)
3. execution time

faulty context

4. time of recovery
5. recovery-time * rate of checkpoint

Figure 39. Metrics Of Performance

VIII. Future work

To extend the state of the art that has been presented in this report, we plan to evaluate the performance of each of the 3 methods of fault recovery by implementing and running them on a simulator. The simulator shall model the same basic hardware for all 3 methods in order to provide a fair comparison. Concurrently, we will develop an efficient hardware implementation of the USM.

Bibliography

A. Avizienis, "Design of fault-tolerant computers", "Proceedings of the 25th International Symposium on Fault-Tolerant Computing--Special Issue", pp. 15-25, June 1995.

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.

M. Banatre and P. Joubert, "Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor", "Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems", pp. 89-96, 1990.

M. Banatre, G. Muller, et. al., "Design Decisions for the FTM: A General Purpose Fault Tolerant Machine", "Proceedings of the 21st International Symposium on Fault-Tolerant Computing Systems", pp. 71-78, 1991.

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.

T. Bressoud and F. Schneider, "Hypervisor-Based Fault-Tolerance", "ACM Transactions on Computer Systems", vol. 14, no. 1, pp. 80-107, February 1996.

J. Chapin, M. Rosenblum, et. al., "Hive: Fault Containment for Shared-Memory Multiprocessors", "15th ACM Symposium on Operating Systems Principles", pp. 1-15, December 1995.

E. Elnozahy, D.B. Johnson, et. al., "Performance of Consistent Checkpointing", "Proceedings of the Eleventh Symposium on Reliable Distributed Systems", pp. 39-47, October 1992.

M. Feeley, J. Chase, et. al., "Integrating Coherency and Recoverability in Distributed Systems", "First Symposium on Operating Systems Design and Implementation (OSDI)", pp. 215-227, November 1994.

A. Gefflaut, C. Morin, et. al., "Tolerating Node Failures in Cache Only Memory Architectures", Proceedings of Supercomputing 1994, pp. 370-379, 1994.

K. Gharachorloo, D. Lenoski, et. al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", "Proceedings of the 17th Annual International Symposium on Computer Architecture", pp. 15-26, May 1990.

J. Gray, "A Census of Tandem System Availability Between 1985 and 1990", "IEEE Transactions on Reliability", vol. 39, no. 4, pp. 409-418, Oct. 1990.

J. Gray, P. Helland, et. al., "The Dangers of Replications and a Solution", ACM. (This paper is currently available from Dr. Gray's web site at Microsoft.)

E. Hagersten, A. Landin, et. al., "DDM -- A Cache-Only Memory Architecture", "IEEE Computer", vol. 25, no. 9, pp. 44-54, September 1992.

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.

G. Janakiraman and Y. Tamir, "Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers", ???, pp. 42-51, 1994.

B. Janssens and W. K. Fuchs, "Experimental Evaluation of Multiprocessor Cache-Based Error Recovery", "Proceedings of the 1991 International Conference on Parallel Processing", vol. 1, pp. 505-508, 1991.

B. Janssens and W. Fuchs, "Relaxing Consistency in Recoverable Distributed Shared Memory", "Proceedings of the 23rd International Symposium on Fault-Tolerant Computing", pp. 155-163, 1993.

B. Janssens and W. Fuchs, "The Performance of Cache-Based Error Recovery in Multiprocessors", "IEEE Transactions on Parallel and Distributed Systems", vol. 5, no. 10, pp. 1033-1043, Oct. 1994.

B. Janssens and W. K. Fuchs, "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory", "Proceedings of the 13th Symposium on Reliable Distributed Systems", Oct. 1994, pp. 34-41.

P. Keleher, A. Cox, et. al., "Lazy Release Consistency for Software Distributed Shared Memory", "Proceedings of the 19th Annual International Symposium on Computer Architecture", pp. 13-21, May 1992.

P. Keleher, S. Dwarkadas, et. al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", "Proceedings of the Winter 94 Usenix Conference", pp. 115-131, January 1994.

P. Keleher, A.L. Cox, et. al., "An Evaluation of Software-Based Release Consistent Protocols", "Journal of Parallel and Distributed Computing, Special Issue on Distributed Shared Memory", vol. 29, pp. 126-141, October 1995.

A. Kermarrec, G. Cabillic, et. al., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", "Proceedings of the 25th International Symposium on Fault-Tolerant Computing", pp. 289-298, June 1995.

R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", "IEEE Transactions on Software Engineering", vol. se-13, no. 1, Jan. 1987, pp. 23-31.

H. Kuefner and H. Baehring, "Dynamic Fault Tolerance in DCMA - A Dynamically Configurable Multicomputer Architecture", "Proceedings of the 15th Symposium on Reliable Distributed Systems", pp. 22-31, 1996.

A. Leon-Garcia, Probability and Random Processes for Electrical Engineering, 2nd ed., chapters 8 and 9, 1993.

V. Lo, "Operating Systems Enhancements for Distributed Shared Memory", Advances in Computers, vol. 39, pp. 191-237, 1994.

K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", "ACM Transactions on Computer Systems", vol. 7, no. 4, pp. 321-359, November 1989.

K. Li, J. F. Naughton, et. al., "Low-Latency, Concurrent Checkpointing for Parallel Programs", "IEEE Transactions on Parallel and Distributed Systems", vol. 5, no. 8, pp. 874-879, Aug. 1994.

C. Morin, private communication by e-mail from IRISA/INRIA in France, December 1996.

N. Neves, M. Castro, et. al., "A Checkpoint Protocol for an Entry Consistent Shared Memory System", "Proceedings of the 13th ACM Symposium on Principles of Distributed Computing", August 1994, pp. 121-129.

J. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers", "IEEE Parallel & Distributed Technology", vol. 2, no. 2, pp. 62-67, 1994.

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.

A. Silberschatz, Operating System Concepts, 4th ed., 1994.

M. E. Staknis, "Sheaved Memory: Architectural Support for State Saving and Restoration in Paged Systems", "3rd International Conference on Architectural Support for Programming Languages and Operating Systems, 1989, pp. 96-102.

R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems", "ACM Transactions on Computer Systems", vol. 3, no. 3, August 1985, pp. 205-226.

M. Stumm and S. Zhou, "Fault Tolerant Distributed Shared Memory Algorithms", "Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing", pp. 719-724, December 1990.

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.

O. Theel and B. Fleisch, "A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs", "IEEE Transactions on Parallel and Distributed Systems", vol. 7, no. 9, pp. 915-930, September 1996.

O. Theel and B. Fleisch, "The Boundary-Restricted Coherence Protocol for Scalable and Highly Available Distributed Shared Memory Systems", "The Computer Journal", Oxford Press, future edition. (This paper is currently available from "http://www.cs.ucr.edu/~brett/".)

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.

K. Wu and W. Fuchs, "Recoverable Distributed Shared Virtual Memory", "IEEE Transactions on Computers", vol. 39, no. 4, pp. 460-469, April 1990.