Recovery of Memory and Process in DSM Systems: HA Issue # 1

Zheng Zhang
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-76
March 30th , 2001*

E-mail: zzhang@hpl.hp.com

multiprocessor, shared memory, high availability

In this report, we discuss the recovery of memory and processes on the platform of a shared-memory DSM system. We divide the problem into recovery of unaffected memory (RUM), and recovery of affected processes (RAP). We point out that specially designed fault-tolerant, non-volatile memory is neither sufficient nor necessary to solve the problem of RUM. It is not sufficient that the system can go down when one node goes away, which can be a result of many types of faults: power failure is but one of them. It is not necessary either, because the system is distributed in nature; information redundancy across fault units can be realized, therefore, without using special memory. We discuss several ways of implementing a fault-tolerant memory system using plain memory by modifying the write-back protocols in DSM systems. The proposed techniques include mirroring and RAIM, which stands for **R***edundant* **A***rray* of **I***ndependent* **M***emory*. The fault-tolerant memory system lays the foundation for other HA solutions, in addition to attack the problem of RUM. We use a novel approach to survey the space of transparent rollback recovery alternatives as our means to target RAP. There are two axes that constitute our space. The first axis is the fraction of fault-tolerant memory system which is part of the reliable storage. This, in many ways determines the cost of the system as well as the checkpoint bandwidth. The second axis is how and when the checkpoint image is established and committed. The three options, built-on-the-fly, stop-and-forward and copy-on-write, have different system complexity and performance implications.

* Internal Accession Date Only
Approved for External Publication

1. Introduction

The faulty scenario of the shared-memory multiprocessor system investigated in this report is single-node (or single protection-domain) failure. We assume a robust, HA-enabled IO subsystem is already in place. We describe the impact of the fault with dependency arcs originated from elsewhere but end at the faulty unit. One example is shown in Figure 1.

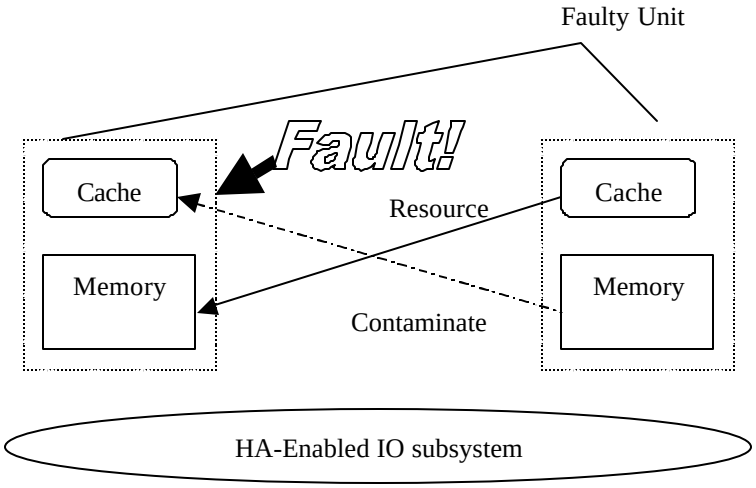


Figure 1 Faulty Unit and the Impact of Fault in MCS System

We distinguish two types of dependencies, *Resource* and *Contaminate*. The first case refers to the memory homed in the faulty unit and is needed by processes running elsewhere. The data contained in these memories are not contaminated by the fault. The second type refers to the cases where the processes ran on the faulty node have contaminated data and memory. An example of such may be that a process running on the faulty unit writes an erroneous data, which is consumed by other running processes before the fault is detected.

The system that we are interested in delivering is not going to be equipped with advance and expensive hardware/software technology that masks the fault completely. Rather, the goals are to:

1. Recover unaffected memory (RUM) so as to resolve the resource dependency, and
2. Recover affected processes (RAP) to reduce the impact on contaminate dependency

We assume OS technique is there to track the dependencies and distinguishe the above two cases. This report intends to investigate different alternatives to achieve the above two goals.

2. Recovery of Unaffected Memory (RUM)

The goal here is to rescue the memory content so that processes depend on these data can survive the fault. We first enlist the current options of fault-tolerant memory. After pointing out why the memory alone does not solve the problem, we discuss alternatives to build fault-tolerant memory system in the context of shared-memory multiprocessors.

2.1 Fault Tolerant Memory

One basic form of fault-tolerant memory is the *non-volatile memory*, which does not lose the contents to power outage. Today NV-RAM is employed in many industry products already, for example as a part of the disk cache. There have been other proposals as well. eNVy, for instance, is a research proposal of non-volatile memory that uses a combination of battery-backup SRAM and Flash memory [1].

Some form of redundancy is required in order to deal with hazards other than the power failure. Sheaved memory [2] is one such proposal, in which several memory page frames may be *bundled* together to permit read-one/write-many semantics.

The fault-tolerant memory alone is neither sufficient nor necessary in a distributed shared-memory platform. They are not sufficient because the system can go down when one node goes away, which can be a result of many types of faults: power failure is but one of them. They are not necessary because the system is distributed in nature; information redundancy across fault units can be realized, therefore, without using special memory.

2.2 Building a Fault-Tolerant Memory System

The solutions discussed here all assumes single-node failure. It is, however, trivial to extend them to tolerate *k*-node failure. All these are done via the modification of protocol, at the relevant stage when the memory content is to be altered. This is the point where redundancy must be applied.

2.2.1 Fault-Tolerant Memory System using Mirroring

One popular way of avoiding single-node fault is simply to make sure each data has two copies in the memory, each of which residing on different node. We call this approach *dual-copy* or *memory mirroring*. This is the approach taken by many studies, in the context of message-passing platform [3, 4] as well as shared-memory multiprocessors [5]. For the platform that we concern here, we simply modify the coherent protocol so that the protocol always updates the *mirror home* of a data whenever its primary home is updated with inbound DMA or cache write-back. This way any single node failure can be tolerated and unaffected memory can be recovered.

The problem with the mirroring approach is that it is very expensive. Indeed, to make *P* pages recoverable we need a total of 2*P* pages. Alternatively, one can think of the mirroring as the analogy of RAID-1 implemented in the memory space. Consequently, there can be a counterpart of RAID-5 [6] in the memory space as well.

2.2.2 Fault-Tolerant Memory System using RAIM

RAIM stands for the ***R**edundant **A**rray of **I**ndependent **M**emory*, which is simply the algorithm of RAID-5 implemented in the memory system. This organization is shown in Figure 2, with an instance of updating a piece of data and its parity. Each frame here is presumably one page. Note we rotate the parity to avoid bottleneck for parity writes.

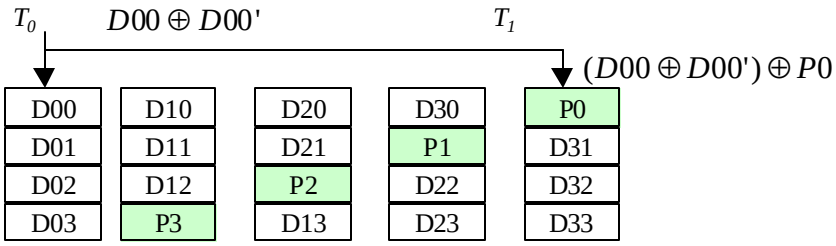


Figure 2 RAIM: D00 is updated with D00'

Assuming the system has *N* independent nodes. With RAIM, we can reconstruct the memory image of the faulty node when single node fault occurs. The operation is *O(LogN)*. This is in contrast with the mirroring which takes, theoretically, no time to recover the memory.

The hardware support for RAIM is relatively moderate. The memory overhead we pay becomes dependent on number of nodes, *N*. The total memory space needed for *P* pages will be *PN/(N-1)*. In other words, the overhead fraction is *N/(N-1)* for RAIM, as opposed to 2 in mirroring. This is a significant saving. The amount of additional traffic is the same for both organizations, namely doubling those corresponds to whenever the memory gets updated for cache write-backs and inbound DMA.

RAIM as is does not have the same fault-tolerant capability as the mirroring. This is so because the parity update depends on the success of the primary update. If the primary update fails, we have to make the pessimistic assumption that the parity update either does not fire at all or is corrupted subsequently. This is in sharp contrast with the mirroring, where the primary and secondary update is independent after leaving the source node. However, if dirty data transfers are made closed and buffered at the sending nodes, this situation can be well dealt with. For more details of this *source-buffering* technique, see a previous report [7]. RAIM with source buffering, therefore, is as strong as mirroring in terms of fault-tolerant capability. It is much cheaper with the tradeoff of slower recovery time.

2.2.3 Some More Comments

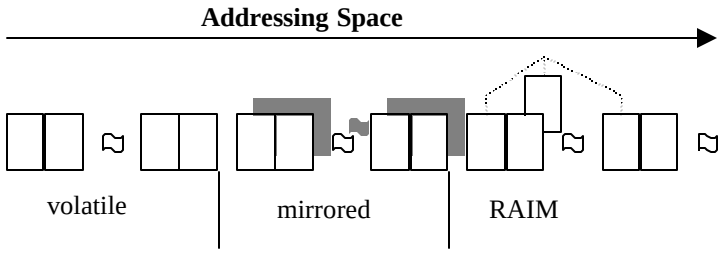


Figure 3 Addressing Space Partitioned into Regions with Different HA Support

We would like to make several comments before concluding this section:

- Both RAIM and Mirroring can be implemented in software – there is nothing that restricts the way that they should be carried out.
- Both schemes can be applied at different scales in the system. For example, either RAIM or mirroring can be done *within* one node. This will make interesting applications such as hot-

replaceable memory banks possible. Also, a *M* node system can use *K* RAIM groups, each with degree *M/K*.

- It is also possible or even desirable, to apply them to separate ranges of addressing space. In fact, one can envision that the addressing space being partitioned into regions with different HA support and guarantee, as shown in Figure 3.

The importance of a fault-tolerant memory system is not to be under-estimated. It alone solves the problem of recovering unaffected memory. Furthermore, it lays the technical foundation for recovery affected processes using checkpointing. Fault-tolerant memory system is the key ingredient for HA solutions discussed in this report and others [8].

3. Recovery of Affected Process (RAP) with Backward Error Recovery (BER)

The recovery technique we consider here falls into the class of *backward error recovery* (BER). BER establishes checkpointing at discrete points. If an error occurs, one only needs to roll back to the last checkpoint and re-establish the application image and continue from there. The processor states to be checkpointed into reliable storage is called the *checkpoint image (data)*, while the data that has been modified since the last checkpoint is called the *active data*. In other words, the checkpoint image is the active data of the very last checkpointing interval. Checkpoint data typically includes all or a subset of following: register sets of the processors, the PCs, and all the memory contents that have been modified since last checkpoint. For non-deterministic applications, logs of inputs must also be maintained and checkpointed.

The checkpointing techniques we will be looking at is *globally coordinated* checkpointing, where all processes of the application being checkpointed will synchronize and wait till all their outstanding transactions to complete, and then proceed to commit the checkpoint. The time breakdown, seen by the application, is outlined in Figure 4. The techniques are also *incremental*, in the sense that only modified data since the last checkpoint will be the subject of the process.

Checkpointing is a well-researched area and there exists a vast body of literature. Some of the interesting survey materials (through which I started the learning of HA issues) can be found in [9-11]. The objective here is to bring this error recovery scheme into the context of the platform we are interested in, and examine/propose adequate solutions. We now briefly review the requirement and API, and then proceed to explore the alternatives.

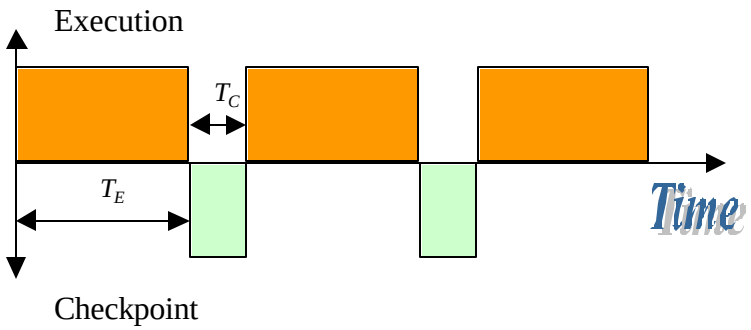


Figure 4 the Time Breakdown for Checkpointed Process

3.1 Cost-Performance Factors of Checkpointing

We assume that the customers are not interested in an expensive system of full redundancy that will mask the faults. Rather, they prefer a system with some HA guarantees *and* at the same time deliver high performance in the absence of faults. In addition, the recovery speed is not their primary concern. Thus, the following two requirements are in order:

- ***HA resolution***: tolerance of loss of *raw* work in case of fault; measured in unit of time.
- ***Performance degradation***: execution time slowdown when HA support is turned on.

Therefore, a good implementation should allow fine recovery resolution, and with as little performance degradation as possible.

Providing checkpointing ability necessarily implies the cost of additional fault-resilient storage where the checkpoint data is stored. The hardware cost of this fault-resilient storage depends on the technology that implements them. Further more, the design and architectural complexity of each scheme is also part of the cost equation and should also be taken into consideration.

3.2 API of Checkpointing

The essence of checkpointing API is to present the following two decisions to the implementation layer: a) what to checkpointing and b) who make the call. There are a range of possibilities, but here we discuss the two extremes: *programmer controlled* and *user parameterized*. The API of these two approached locate at different levels, the first being at the library call level invoked by the programmer at the time of coding, while the second is invoked by the user at the OS level.

In the first model, the programmer will insert checkpoint library calls that are bracketed by barriers. For example:

```
barrier ( chk_bar );
checkpoint( x );
checkpoint( y );
...
barrier ( chk_bar );
```

The advantage of this approach is being economic in using the checkpointing bandwidth: the programmers (supposedly) know exactly what data they need to checkpoint. However, this is not a familiar syntax to think about and can be quite a burden for the programmers: the checkpointing data must be a complete set to which a rollback will be able to re-execute completely. Another disadvantages is that it is hard for the programmers to get an idea as how much performance degradation they are going to expect, which depends on the checkpoint frequency, the size of the checkpoint data and the checkpointing bandwidth the system provides. This is not an easy equation to solve.

Both of the disadvantages in the above approach will be and can be challenged if time comes for people to realize that they must write applications that is *HA-aware*. Ideally, the task for the programmers can be simplified to as little as only giving out checkpointing hints at the place they would like to have the data checkpointed. It may be relatively an easier job for the compiler to figure out what data is alive at this point and what is not [12]. Checkpointing calls can be inserted for the first class of data, while the second class should only be flushed to stable storage once for all. Front-end profiling can provide the programmer with an idea of how frequent the checkpoint

process will be invoked and thus how much the performance impact they should expect, and subsequently guide further checkpoint fine-tuning. Of course, the compiler can not be as precise as the programmer can, thus some unnecessary checkpoint data size increase will be seen.

The *user-parameterized* approach provides an API at the OS level. The only thing the user needs to do is to specify two parameters, which directly relate to the two HA performance requirements that we discussed earlier: the performance degradation and the checkpoint resolution. A possible UNIX command, *ckp*, and its syntax is outlined in the following example:

```
ckp -resolution 5m -degrade 10 -no_log foo
```

Which runs the program *foo* with HA support. In this case, the user is willing to lost 5 minutes of useful work in case of a fault, the performance degradation due to checkpointing is bound to 10%, and this application is deterministic and thus no logging of input messages are necessary.

It is always possible to satisfy user's expectation if only one of the HA performance requirement is specified. For example, if only the resolution is specified, then the OS can preempt the application and make a checkpoint after so much user work has been done. Likewise, if only the performance degradation is specified, the OS can perform the checkpointing after it calculates the ratio of how long it will take to checkpoint the active state versus how long the application has executed since the last checkpoint. If, however, that both resolution and the performance degradation are specified, then we might not always be able to meet the goal if the underlying system does not have enough checkpointing bandwidth. In this case, we assume the user prefers one goal to the other.

While this approach has relieved the user from the burden of understanding the checkpointing process, it can place tremendous pressure on the checkpointing bandwidth of the system. The situation is especially bad when the application itself might be using lots of scratch space that does not need to be checkpointed at all.

An optimal approach can be reached if the application is garbage-collection enabled. In this case, the garbage collection, whether actually run or not, will output what portion of the active data does *not* need to be checkpointed, and thus reduce the checkpoint bandwidth requirement. This observation has been made in the past, as pointed out by [9].

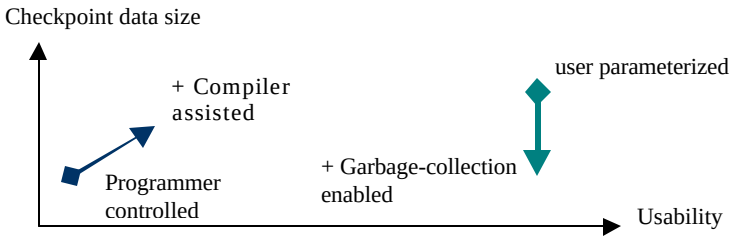


Figure 5 Four API Strategies for Checkpointing

The four API strategies are summarized in Figure 5. From this point onwards, we will treat when and what to checkpoint as a given. What we will focus on is to understand which implementation is the most cost-effective under which situation.

3.3 The Space of Checkpointing Alternatives

There are many ways of dividing the space of checkpointing alternatives. We will follow two axes that divide the space. The first being the raw checkpoint bandwidth, and the second being how checkpoint image is established/committed. Throughout this report, we assume that the active image is kept in volatile memory and is thus open to faults. One would think that if the active image is protected by fault-resilient memory system on top of mirroring or RAIM the solution will be more expensive. It turns out not so. Cost-effective HA solution that relies not on the availability of warm (meaning in memory) checkpoint data, but rather the capability of rebuilt it quickly, are the themes of two other reports [8, 13].

The raw checkpoint bandwidth into the reliable storage almost single-handedly defines the cost-performance factor. A larger bandwidth will enable finer checkpoint resolution and as little performance degradation as possible. From this standing, we can broadly classify different strategies into three categories, depends on how much reliable memory is as part of the reliable storage. At one end of the spectrum, there are schemes using the low cost (in terms of dollars per megabytes) disks only, obviously they can not deliver much bandwidth. We call them ***Zero-Memory Checkpointing***. At the other end, people have proposed schemes capable of checkpointing all application memory, such scheme we call ***Symmetric-Memory Checkpointing***. An immediate advantage of such system is the fast recovery time, since at the recovery the whole checkpoint image is warm in the memory. Somewhere in between, we can have schemes that use less memory by keeping only a fraction of the checkpoint data. They are called as ***Asymmetric-Memory Checkpoint***.

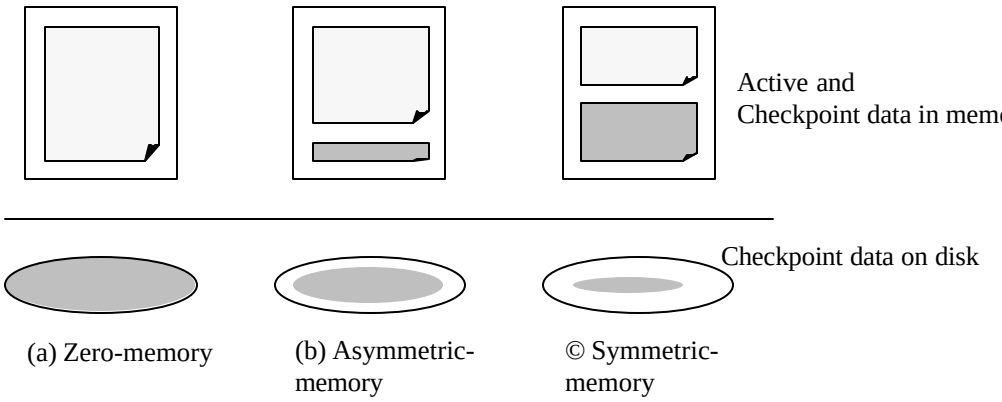


Figure 6 Zero-Memory, Asymmetric-Memory and Symmetric-Memory Checkpointing

The second ax that we use to differentiate various schemes is the way that the checkpoint is established and committed. The simplest form will stall the application and writes the checkpoint into reliable storage, and then continue. Borrowing interconnect technology, we call this the ***stop-and-forward***. The second form simply marks the modified data to be *unchangeable* and continue the execution regardless. A daemon will pull the data into the reliable storage at the background. If the process ever intend to modify the unchangeable data, they will be halted until the data made its way to the reliable storage. We call this approach ***copy-on-write***. Following these two, it is obvious that there exist one more possibility in which checkpoint image is built on the fly *before* the checkpoint. The checkpoint does nothing other than making an official statement that the checkpoint image built is ready to commit. We name this as ***built-on-the-fly*** (better naming suggestions welcome!).

The stop-and-forward approach is the simplest among all. But since it exposes the full checkpoint latency to the application, it has the worst performance. The other two choices aim at hiding the checkpoint latency along with application execution. Copy-on-write is more complex and performs better, but it also depends heavily on the application – if the application repeatedly modify the same pages (and do it soon enough), the frequent stall to flush the unchangeable back to reliable storage simply can make it behave as many small stop-and-forwards. The added overhead of system interruptions might very well bring worse performance. It has the added disadvantage that if a fault occurs before all the unchangeable data is flushed, we need to rollback to one checkpoint earlier than the end user expected. Built-on-the-fly is very attractive, but relies on hardware support and is not always practical. As always, there can exist hybrid schemes that use the combination of the three approaches.

The two-dimensional space, one ax being the fraction of reliable-memory as a fraction of reliable storage, the other being the tactic that establish/commit the checkpoint, is the one that we will discuss in turn.

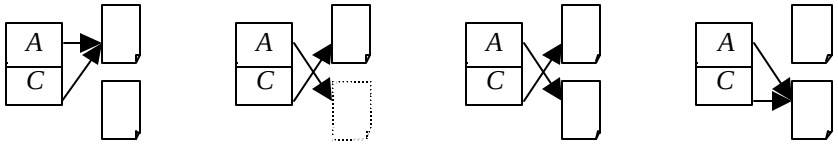
4. Zero-Memory Checkpointing

Zero-memory checkpointing always store the checkpoint image onto the disk. This is the simplest and the most economic form of checkpointing. The problem, of course, is the low checkpoint bandwidth that the system can offer, leading to poor application performance. Consequently, zero-memory checkpoint with stop-and-forward option is only practical in a multiprogramming environment where throughput is much stronger a demand than latency. In addition, the applic ations must be able to hide the checkpoint latency from each other, and they do not need excessive IO bandwidth themselves.

The copy-on-write approach, when checkpointing, simply marks all the pages and then allows the processes to keep on execution. A daemon then runs in the background pulling data into the disk and reset the marks on the fly. If the processes attempt to modify any marked data, they are stalled and the data is flushed to disk. As pointed out earlier, it is possible that the checkpoint process to be interrupted by a fault and one has to roll back to the previous checkpoint, violating the resolution requirement filled out by the end user. Conversely, to reach the resolution requirement user specified it is possible that the actual checkpoint frequency might need to be higher. Another caveat is that the performance is heavily application dependent. If the active data overlaps with the checkpoint data and the overlap occurs fast enough, we may very well end up with same amount of stall time due to checkpointing, if not more (considering all the overhead involved in trapping the processes).

The built-on-the-fly version does not appear as a serious candidate at the first glance. However, it plays a vital rule to ensure the correctness of even the base zero-memory checkpoint scheme. If a dirty page is swapped out to the disk, either as a result of paging or an exercise of this option, the page has already safely made to the disk. Presumably, if the page is not to be modified again, that version of page should be taken by the checkpoint. However, when the page is written to the disk, caution must be taken that we do not wipe out the checkpoint version of the page. The problem is how should we efficiently manage the coexistence of the checkpoint and active pages on the disk, which is more general than we thought. Shadowing (twin paging) is a nature solution. We discuss one implementation at the OS level in the next section. A solution at the disk level is discussed in [14].

4.1 Manage Active and Checkpoint Data on Disk



(a) Checkpointed (b) Page dirty in memory (c) Dirty page written back (d) New checkpoint

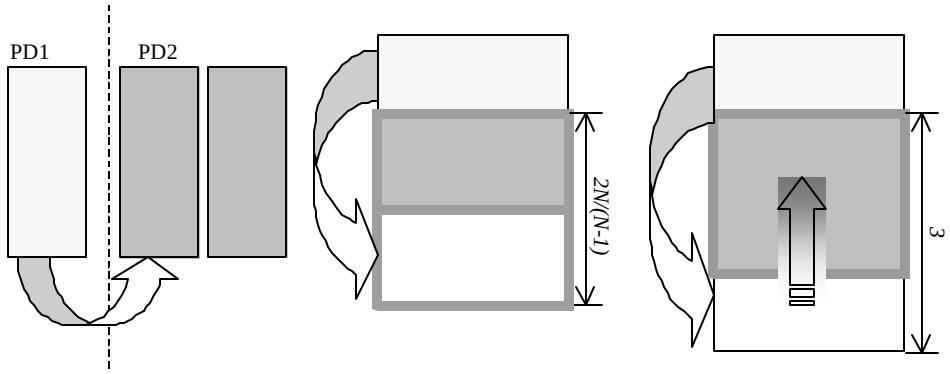
Figure 7 Manage the Coexistence of Active Data and Checkpoint Data on Disk

Because active data and checkpoint data coexist on the disk, each entry in the page table needs two pointers. One pointer *A*, pointing to an active page while the other one *C* points to a checkpoint page, as shown in Figure 7. We now describe the algorithm with which these pointers are manipulated.

Without loss of generality, we start with the case in which processes have just completed the checkpointing (Figure 7-(a)). In this case, both pointers point to the same page. Once process starts execution and pages are modified in the memory, the active pointer points to the other page which holds stale data anyway (in fact, as process goes along, that's the *previous* checkpoint data) (Figure 7-(b)). The dirty page maybe written back to memory before the new checkpoint begins, in this case they are written to where the *A* pointers point at (Figure 7-(c)). At the moment of checkpointing, modified pages are either dirty in memory, or have been written back to disk. The establish phase of the checkpointing process simply flushes all dirty pages back to disk, writing to where the *A* pointers pointed at. When this phase successfully completed, the commit phase simply change the *C* pointers to align with the *A* pointers. Assume the disk is already HA-enabled, faults occurs during either phase can be handled without any problem.

5. Symmetric-Memory Checkpointing

Symmetric-memory checkpointing, where the *complete* image of the last checkpoint is available *immediately* upon recovery, can be implemented in several different ways. We will describe them in turn.



(a) Single copy (b)RAIM © Dual Copy

Figure 8 Alternatives of Symmetric-Memory Checkpointing

5.1 Single-Copy Checkpointing

The single-copy checkpointing scheme is shown in Figure 8-(a). The checkpoint data is stored in volatile memory and is therefore itself *not* fault-resilient. However, the active data and checkpoint image is separated by a HA boundary, meaning that no faults are going to simultaneously take both of them down. Obviously, this approach limits the scale of the HA solution to one protection domain.

Theoretically, we must allocate two memory frames, each is large enough to take one checkpoint image, in the backup domain. This is dictated by the atomic principle of checkpointing. One frame of the memory is updated with the checkpoint data, while a complete undisturbed checkpoint image of the last checkpoint interval must be stored in another frame to prepare for any fault might occur during the checkpoint process itself. Once a checkpoint image is committed, the backup frame will be freed to load the next checkpoint of the next interval. In other words, the two frames alternate their roles as the application running along.

Handling Partial Checkpoint Image Update. For incremental checkpointing, however, typically only a fraction of the total memory needs to be copied to the backup. Alternation of the two frames will not work if we only update part of the checkpoint image. We can combine the two frames together to form a stack, assigning one of them to be the primary. The primary always receives the new checkpoint and the old content it has is pushed to the second frame. This is an elegant and practical approach.

The above operations are described as a stop-and-forward version. It is straightforward to adopt it to either built-on-the-fly or copy-on-write approaches. In fact, Mariposa family of HA solutions can be considered as variations of the built-on-the-fly approach [8, 13, 15], where writes are constantly reflected to the backup domain.

The problem of built-on-the-fly approach is that there may be multiple writes to the same line within one checkpoint interval, all being reflected to the backup domain. We can, on the other hand, mark all the lines that need to be checkpointed and then proceed into the next interval. No modification to a marked line is allowed (such as a write ownership grant) until we have sent it to the backup domain. This will be the copy-on-write version and it avoids the same-line-writes problem mentioned above. However, the difficulty is that we must make sure that all marked lines, not only those to whom modifications are attempted in this interval, should be checkpointed. A daemon reflecting these lines seems to be the only, not very attractive solution.

5.2 RAIM Checkpointing

The organization of RAIM symmetric-memory checkpointing is shown in Figure 8-(b). The checkpoint data is kept fault-resilient using the RAIM technique and, unlike the single-copy scheme, puts no restriction on where the checkpoint data must reside and how it is mixed up with the active data. Logically the RAIM range is divided into two identically sized buffers; each is large enough to host one checkpoint image. The operations for various options on the two buffers are almost identical to the case of single-copy.

The symmetric-checkpointing using RAIM has memory overhead rate of $2N/(N-1)$, closer to that of single copy for large N.

5.3 *Dual-Copy Checkpointing*

The dual-copy symmetric-memory checkpointing is shown in Figure 8-(c). Checkpoint data is made fault-resilient by keeping two copies in two distinctive nodes. As in the case of RAIM checkpoint, there is no restriction on where the checkpoint data should be placed.

The establish phase of the checkpoint process first makes another copy for each piece of the active data. This ensures that the active data is made fault-resilient before overwriting the checkpoint data. If this phase successfully finishes, the commit phase then goes ahead overwrites one copy of the old checkpoint data. The total memory overhead is thus 3.

Can we do better with less memory overhead, say 2? We now prove that this is generally not possible. Considering the active data *X*, whose checkpoint data is on node *A* and *B*. Now we overwrite the copy on *A* with the active data *X*. At this instance, if node *B* fails *and* there maybe an active data *Y* on it that has not yet made a new copy, we have lost the *only* old copy of *X* and the new copy of *Y*. If such event indeed occurs, we can roll neither forward nor backwards because the atomic property has been violated. In other words, the checkpoint of active data in one node is dependent on another node, if the action of copying active data in the first node leaves a subset of mixture of the only old and new copies in the second node. If all nodes depend on some other nodes, we say a *deadlock* has occurred. Such an example is shown in Figure 9-(a), the copying of the page 1 into node 2 clearly depends on the availability of node 3, which keeps the only new copy of page 3.

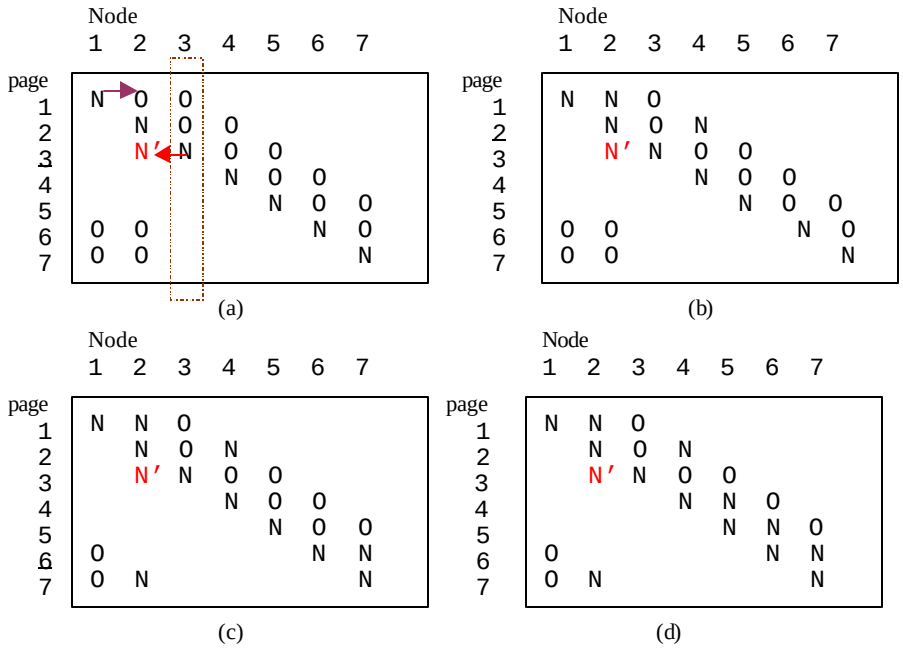


Figure 9 Reduce the Memory Overhead in Dual-Copy Checkpointing

The same proof we have given above suggests how we can attack the problem in general. Whenever we have a deadlock, we can always break the dependence cycle by making extra copies. In Figure 9-(b), we break the dependency by making a new copy of page 3. We can then proceed until another deadlock is reached, if any. We can derive a greedy algorithm that picks up the node to break the dependency based on a) how many dependencies can be broken and b) how many

extra copies are made. This algorithm will do much better than the overhead up-bound of 3. For this particular example we use only 1 extra copy and all the deadlocks are broken as a result of chain reaction. The process is described in Figure 9, where the pages that finished checkpointing are underlined in each step. We need yet to understand what is the lower bound of the overhead. Another problem is that the algorithm is sequential in nature, although copy operations between deadlock stages are parallel. In this example, the process has to be finished in four steps. One way of achieving more parallelism is to break more dependencies on several nodes, instead of one, whenever deadlocked.

The above discussion assumes a stop-and-forward approach. The dual-copy does not allow an efficient and complete implementation for built-on-the-fly or copy-on-write approach. We can only furnish a partial built-on-the-fly solution, even with 3 buffers. What we do is to reflect the writes to the 3rd buffer before checkpoint. At the time of checkpoint, the establish phase is thus completed already. We then select one of the remaining buffers and copy into it the modified pages from the 3rd buffer. This is the process of the commit phase. This concludes one checkpoint interval. Note that the application must be stalled during this copy duration. As the execution proceeds into the next checkpoint interval, the idle buffer that was not selected can be freed to host the reflecting writes. In other words, the three buffers can rotate their roles as the execution continues.

6. Asymmetric-Memory Checkpointing

As shown in the previous section, requiring that the *complete* checkpoint image be *immediately* available upon recovery requires excessive memory space. The approaches that relax this demand by rebuilding checkpoint image quickly at the speed of memory is one attractive solution and is the topic of the two subsequent reports [8, 13]. Asymmetric-memory checkpointing explores the possibility of keeping only part of the checkpoint image in the memory by utilizing the locality property the application may exhibit.

In asymmetric-memory checkpointing, the fault-tolerant memory system is only large enough to keep a fraction of the total application data. The first goal is therefore to maximize the overlap of the newly modified data with those kept in the memory. By doing so we will be able to minimize the chance of running out memory buffer to store the checkpoint and hence having to flushing to the disk while checkpointing. Notice that it's impossible to reach an optimal solution in this regard. One heuristic procedure may be as follows:

> Whenever a new modified page is *P* generated, the replacement algorithm will check if its last checkpoint page *P'* is in the memory. If so it goes ahead pins that page. If not, it reserves a free page if there is any, or replaces an unpinned page using LRU algorithm. As the number of modified pages increases, old checkpoint pages that do not overlap with the current working set gradually gets flushed to the disk.

This is not all. To take into account of recovery, we should retain in the buffer those pages that are not modified but *referenced*, because these are the pages that the application will reference if they fall back to the checkpoint. The replacement algorithm must be designed carefully to balance the above two requests.

The above discussion assumes the stop-and-forward implementation. In fact, that's about all the asymmetric checkpointing can do. The fact is that since the fault-resilient memory system now carries only a subset of the pages, it is very hard to establish a one-to-one hardware mapping between the active page to any page in the fault-tolerant buffer. The absence of a simple, one-to-

one mapping relationship between the active data and the buffer makes it impractical to implement built-on-the-fly solution, which has very low checkpoint overhead. Same holds for the copy-on-write approach. Asymmetric-memory checkpointing is a seemingly promising solution but apparently fall short on delivering much at the end.

7. Related Works and Conclusions

In this report, we discuss the recovery of memory and process on the platform of shared-memory DSM system. We divide the problem into recovery of unaffected memory (RUM), and recovery of affected processes (RAP). We point out that specially designed fault-tolerant memory is neither sufficient nor necessary to solve the problem of RUM. We discussed several ways of implementing a fault-tolerant memory system using plain memory. The proposed techniques laid the foundation for other HA solutions, in addition to attack the problem of RUM. We surveyed the space of transparent rollback recovery alternatives as our means to target RAP. There are two axes that constitute the space. The first ax is the fraction of fault-tolerant memory system as part of the reliable storage. This in many ways determines the cost of the system as well as the checkpoint bandwidth. The second ax is how and when the checkpoint image is established and committed. The three options, built-on-the-fly, stop-and-forward and copy-on-write, have different performances and implications on the system complexity.

The rollback recovery technology has been the focus of a great number of researchers. Some of the interesting surveys can be found in [9-11]. The method employed in this report to divide the alternative space is however new to this author's knowledge. N+1 parity has been proposed to implement diskless checkpoint schemes [16] in message-passing platforms. Using RAIM technology to implement a general-purpose fault-tolerant memory system in shared-memory multiprocessor is a novel concept.

8. Acknowledgement

I thank Manu Thapar for the crucial management support and encouragement he gave me to continue my study on HA even if a previous report [7] was shown not as fruitful as I have wanted. The selection of this topic owes a great deal to Bart Sears' suggestion. John Jinakiraman taught me a great deal of recovery issues, I appreciate his precious time. Tom Rokicki was kind enough to go over a draft version of this report and gave encouragement, I am grateful for his comments.

References

[1] M. Wu and W. Zwaenepoel, "A Non-Volatile, Main Memory Storage System," presented at the 6th Architectural Support for Programming Language and Operating Systems, 1994.

[2] M. E. Staknis, "Sheaved Memory: Architectural Support for State Saving and Restoration in Paged System," presented at the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, 1989.

[3] M. Stumm and S. Zhou, "Fault Tolerant Distributed Memory Algorithms," presented at 2nd IEEE Symposium on Parallel and Distributed Processing, 1990.

[4] A. Kermarrec, G. Cabillc, A. Gefflaut, C. Morin, and I. Puaut, “A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability,” presented at the 25th International Symposium on Fault-Tolerant Computing Systems, 1995.

[5] C. Morin, A. Gefflaut, M. Banatre, and A. Kernarrec, “COMA: an Opportunity for Building Fault-Tolerant Scalable Shared Memory Multiprocessors,” presented at the 23rd Annual International Symposium on Computer Architecture, 1996.

[6] G. A. Gibson, “Redundant Disk Arrays: Reliable, Parallel Secondary Storage,” : University of California, Berkeley, 1990.

[7] Z. Zhang, “High Availability Issues in DSM Systems: Research Opportunities,” Hewlett-Packard Laboratories, Technical Report 1997.

[8] Z. Zhang, “Single System HA Solutions,” Hewlett-Packard Laboratories, Techinical Report 1997.

[9] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang, “A Survey of Rollback-Recovery Protocols in Message-Passing Systems,” School of Computer Science, Cargegie Mellon University, Technical Report CMU-CS-96-181, October 3 1996.

[10] C. Morin and I. Puaut, “A Survey of Recoverable Distributed Shared Memory System,” IRISA, Technical Report 975, December 1995.

[11] D. K. Pradhan, *Fault Tolerant Computer System Design*: Prentice Hall, 1996.

[12] C. C. Li and W. K. Fuchs, “CATCH: Compiler-assisted Techniques for Checkpointing,” presented at 20th International Symposium on Fault-Tolerant Computing, 1990.

[13] Z. Zhang, “Mariposa+: Some Thoughts on Further Optimization for Mariposa,” Hewlett-Packard Laboratories, Technical Report 1997.

[14] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wikes, “Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees,” Hewlett-Packard Laboratories, Technical Report HPL-CSP-92-9, March 1992.

[15] M. Ziegler, “"Mariposa" -- Adapting the Sequoia "Fulcrum" Technology for Use in the Yosemite CEC,” System Architecture and Design Laboratory, Hewlett-Packard Company, Technical Report number, February 7 1997.

[16] J. S. Plank and K. Li, “Faster Checkpointing with N+1 Parity,” presented at the 24th Annual International Symposium on Fault-Tolerant Computing, 1994.