Checkpointing and Recovery of Shared Memory Parallel Applications in a Cluster

Ramamurthy Badrinath[a*], Christine Morin[b], Geoffroy Vallée[c]
[a,b]IRISA/INRIA
[c]EDF R&D
INRIA PARIS Project-team, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
[a]badri@cse.iitkgp.ernet.in, {[b]cmorin, [c]gvallee}@irisa.fr

Abstract

This paper describes issues in the design and implementation of checkpointing and recovery modules for the Kerrighed DSM cluster system. Our design is for a DSM supporting the sequential consistency model. The mechanisms are general enough to be used in a number of different checkpointing and recovery protocols. It is designed to support common optimizations for performance suggested in literature, while staying light-weight during fault-free execution. We also present preliminary performance results of the current implementation.

1. Introduction

As cluster computing systems supporting the DSM programming model come to be more widespread in use, supporting fault tolerance in these systems has attracted a lot of interest. Several works have dealt with evaluating the overhead of various checkpointing protocols ([12] provides a survey); they have dealt with tracking dependencies[16, 2] and supporting some checkpoint optimizations[6, 8].

Several systems for checkpointing and recovery of applications running on DSM models have been described in literature. Yet, in practice, DSM systems over commodity hardware that support fault tolerance have been scarce. While [6, 8] deal implicitly with sequential consistency models, coordinated checkpointing and recovery by rollback, [14, 4, 13] and [3] use release consistency, uncoordinated checkpointing and recovery by replaying some log information. Further, several of these works including[7] also recognize the issues of reconstructing the state of locks or barriers, and the state of the DSM machine. In [6] the authors realize the importance of providing negative acknowledgments or *aborts* when it is detected that a DSM operation or a synchronization primitive cannot be completed in the presence of a fault.

In this paper we attempt to design fault tolerance for the kernel level DSM system called Kerrighed (formerly Gobelins), but limit ourselves in this presentation to the DSM aspects of Kerrighed. In Section 2 we give an introduction to the Kerrighed model for cluster computing supporting the DSM model, introducing the idea of containers, memory managers and synchronization services. In Section 3 we describe the fundamental fault tolerance mechanisms. In particular we describe the dependency tracking mechanisms. In Section 4 we describe what state we store for the process and the memory and how we restore them. We also discuss several important implementation issues in this section. Section 5 presents performance results from our initial implementation of coordinated checkpointing. Finally in Section 6 we summarize the effort and chart out the future course.

2. DSM Model

The Kerrighed cluster system is implemented as an extension to Linux and is composed of a set of distributed services (Figure 1) providing global management of different logical resources in the cluster. Gandalf module is in charge of global memory management, Aragorn module is in charge of global process management, Elrond module provides a set of synchronization tools for parallel applications (atomic counters, locks, barriers...). All these services are based on Gimli for their communications. Gimli is a service providing high performance reliable kernel to kernel communication inside the cluster.

2.1. Containers for Global Memory Management

In a cluster, each node executes its own operating system kernel (*host operating system*), which can be coarsely

[*] *On leave from IIT-Kharagpur, India.*

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

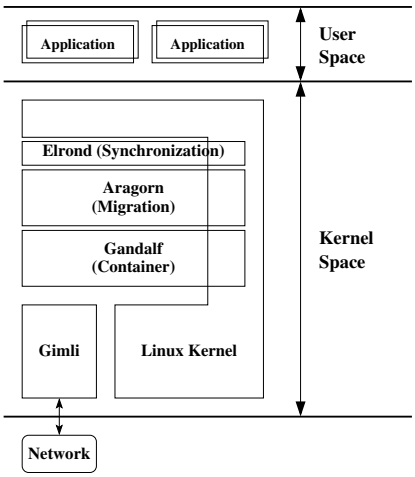

Figure 1. Kerrighed architecture

divided into two parts: (1) *system services* and (2) *device managers*. For global memory management, we propose a generic service inserted between the system services and the device managers layers called *container* [10]. Containers are integrated in the core kernel thanks to *linkers*, which are software pieces inserted between containers and existing device managers and system services. The key idea is that containers give the illusion to system services that the cluster physical memory is shared as in an SMP machine. We give in the remainder of this section a brief overview of container and linker mechanisms and how they can simply be used to provide a kernel level shared virtual memory system. A detailed version is in [11].

2.2. Container Definition

A container is a software object allowing to store and share data cluster wide. A container is a kernel level mechanism completely transparent to user level software. Data is stored in a container on demand by the host operating system and can be shared and accessed by the host kernel of other cluster nodes. Pages handled by a container are stored in page frames and can be used by the host kernel as any other page frame. Container pages can be mapped in a process address space (by appropriately linking it to regions of the virtual Memory Area, or VMA, of the process), be used as a file cache entry, etc.

By integrating this generic sharing mechanism within the host system, it is possible to extend to a cluster scale traditional services offered by a standard operating system (see figure 2). This allows to keep the OS interface known by users.

The memory model offered by containers is *sequential consistency* implemented with a write invalidation protocol similar to [9]. This model is the one offered by physically shared memory.

2.3. Linkers

Many mechanisms in a core kernel rely on the handling of physical pages. Linkers divert these mechanisms to ensure data sharing through containers. To each container is associated one or several high level linkers called *interface linkers* and a low level linker called *input/output linker*. The I/O linkers allow containers to access a device manager.

Connecting a Container to System Services An interface linker changes the interface of a container to make it compatible with the high level system services interface. This interface must give the illusion to these services that they communicate with traditional device managers. Thus, it is possible to "trick" the kernel and to divert device accesses to containers. It is possible to connect several system services to the same container. There is a *mapping* interface linker to allow mapping of container pages to the process address space.

Data Input/Output in Containers During the creation of a new container, an input/output linker is associated to it. The container then stops being a generic object and allows to share data coming from the device it is linked to. For each semantically different data to share, a new container is created. For instance, a new container is used for each memory segment to share or to be visible cluster wide.

Just after the creation of a container, it does not contain any page and no page frame contains data from this container. Page frames are allocated on demand during the first access to a page. Similarly, data can be removed from a container when it is destroyed or in order to release page frames when the physical memory of the cluster is saturated. These actions are performed by input/output linkers on a demand from the container. There is a memory I/O linker to deal with the memory data. A container linked with such a linker is called a *memory container*.

2.4. Shared Virtual Memory

The virtual memory service of an OS allows sharing of data between threads or between processes, through a system V segment, for instance.

A shared virtual memory allows several processes running on different nodes to share data through their address space. Providing this service in a cluster requires us to ensure three properties: (1) data sharing between nodes, (2) coherence of replicated data and (3) simple access to shared data thanks to processor read/write operations.

The container service ensures the two first properties. The third one is ensured by the *mapping* interface linker.

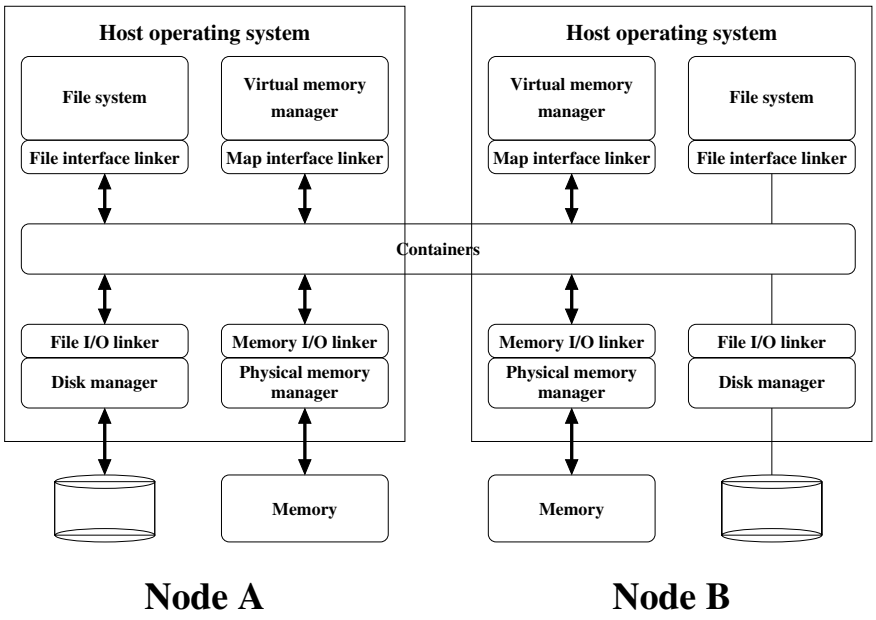

Figure 2. Integration of containers and linkers within the host operating system

Thus, mapping a memory container in the virtual address space of several processes via a mapping linker leads to a shared virtual memory.

When a process page fault occurs, the mapping interface linker diverts the fault to containers. The container mechanism places a copy of the page in local memory and ensures the coherence of data. Lastly, the mapping interface linker maps the local copy in the address space of the faulting process and changes virtual page access right according to the rights of the page in the container.

Finally, we note that the pages of the containers are distributed over nodes of the system. A *page manager* thread is a kernel server responsible for keeping information on the state and location of a set of pages. At any time the page may be located at any node, with that node possessing permissions to modify the page. That node would then be called the *owner* of the page. For the DSM system to function the page managers cooperate to exchange and update ownership and state information for pages.

3. Fault Tolerance Mechanism

From the description in the previous section the processes in an application in Kerrighed communicate using shared memory which is realized using containers. Inter process communication (IPC) in any form causes dependencies, among the tasks that are communicating, that must be honored when computing a globally consistent state for recovery. Hence in a system to provide fault tolerance and to support optimizations of fault tolerance protocols it is important to provide mechanisms to track dependencies. The dependencies that are generated are analyzed during the checkpoint or recovery phase and are an essential ingredient to computing recovery lines. For instance suppose p_1 writes a value to a variable which p_2 reads. If due to faults p_1 is rolled back to a state prior to the write then p_2 needs to be rolled back to a state prior to the read. p_1 and p_2 are respectively referred to as the source and destination of the dependency generated.

In practice these mechanisms need to be particularly light-weight since they may be activated whenever there is IPC adding to the overhead of IPC.

The actual use of the mechanisms described below depends on the checkpointing and recovery scheme implemented. In [5] one finds a description of various checkpointing and recovery strategies. [1] provides a detailed account of the dependencies generated by IPC discusses how the mechanisms may be used to support a variety of checkpointing and recovery protocols.

We now present a mechanisms to track dependencies. This is a generic mechanism and usable in coordinated as well as uncoordinated checkpointing protocol implementations, as well as support message passing, though we are mainly interested in DSM aspects for this presentation. In our implementation we propose to track direct dependencies between memory elements (pages) and tasks, and among tasks. Thus pages and tasks are the basic entities among which we track dependencies. We describe an outline of the mechanism below and refer the reader to [1] for details, with examples, of their usage with respect to reads and

writes to shared memory.

We assume each entity in the system (i.e., each page and each task) has a distinct identity. We associate with each entity the following attributes:

1. An integer called a *sequence number* (**sn**). This is initialized to 1 when the entity begins its existence. This is incremented only when the entity takes a checkpoint. Thus a sequence number can be used to distinguish intervals between checkpoints. If in an interval is the value of the sequence number is x, we will refer to it as interval number x. The sequence number of the source of a dependency is delivered to the destination of the dependency whenever a dependency is created. The local value of sn will always be the *checkpoint number* of the next checkpoint to be taken for the entity. We will denote by $c_{i,j}$ the j^{th} checkpoint taken by entity i.

2. A vector called the *direct dependency vector* (**DDV**). Over time, due to IPC, an entity receives a number of sequence numbers from other entities. These are stored in the DDV of the entity, thus recording dependencies. Initially for an entity j, $ddv[j]$ contains all zeros. If there is a dependency from entity i to j (for instance a message is sent from task i is received by task j), and the sequence number (of i) sent in the interaction is x, and $ddv[j]$ is the local dependency vector with entity j, then on the event at j, we execute the following code:
$$ddv[j][i] = \max\{ddv[j][i], x\}$$
Whenever we checkpoint an entity j we store along with the local checkpoint, the corresponding DDV, i.e., $ddv[j]$. This saved DDV is called the *timestamp* of the corresponding checkpoint. Whenever a checkpoint is taken at a node j we execute the following code:
$$ddv[j][j] = sn; sn++;$$
`save the timestamp` $ddv[j]$.

We will refer to the timestamp associated with the k^{th} checkpoint of entity j as $ddv_k[j]$. Note that $ddv_k[j][j] == k$. For completeness sake, the timestamp for the zero-th checkpoint for all entities has all zeros.

This mechanism essentially records direct dependencies between checkpoints of various entities. So for instance if entity i decides to rollback to checkpoint number n, then for entity j if $ddv[j][i] > n$, then clearly entity j needs to rollback to a checkpointed state for which $ddv[j][i] \leq n$. Of course this covers only direct dependencies. The recovery line computation is the responsibility of the recovery protocol. In [2, 16] it is shown that tracking direct dependencies suffices for computing the recovery line.

With this mechanism in place it is possible to detect all direct dependencies. Yet, in the case of shared memory, some optimization is possible. Consider a page that has not been changed between the two checkpoints $c_{i,j}$ and $c_{i,k}(k > j)$ for the page, then clearly any read after $c_{i,k}$ still refers to the version checkpointed at $c_{i,j}$ provided the page has not yet been modified before the read after $c_{i,k}$. Recording the newer sequence number by the reader results in an artificial dependency. Hence in a read of a page we may prefer to use an older value than the actual sequence number of the page. For this we introduce the following third attribute for each page:

3. An integer counter called the *last write number* (**lwn**). This maintains the interval number of the last interval during which the page was written to. On a read the *lwn* of the page is 'received' by the reading task rather than the *sn* of the page. Also this means that on write to a page, the page must update *lwn* to its current value of *sn*.

One uses the dependency information captured by these attributes during the analysis of the fault just prior to recovery (see Section 4.3). One can also use the *lwn* and *sn* to support incremental checkpointing. The value of the expression *sn* == *lwn* evaluates to `true` at a checkpoint if and only if the page has been modified since the last checkpoint.

4. Checkpoint and Recovery Implementation

In order to implement recovery from a fault by rollback recovery it is important to identify the items or information lost on a failure. In a DSM system we lose all the execution states of all the tasks running on that node. We also lose all pages residing on that node. We also lose the memory managers that are managing pages on that node. In addition we may lose information for distributed objects like locks and barriers, or these may enter invalid states (for instance a lock is considered held by a process which is now dead). With this in mind we design modules to handling checkpointing and recovery. These are described below.

4.1. Checkpointing

When we checkpoint tasks we need to checkpoint their private state including the VMA map information. Checkpointing when in kernel state (e.g., due to page fault or system call) is not attractive as we will then have to re-build kernel state information on recovery. Hence we propose to checkpoint tasks only when they are not in some kernel mode state already. Focusing on coordinated checkpointing, we can say that we wish to make sure all processes are in a *safe* state for checkpointing. There is a *thread checkpointing kernel server* running on each node which will respond to requests to checkpoint tasks on that node. In general this thread will receive a request to checkpoint a list

of tasks running on that node and will check for whether that task is in a safe state and initiate checkpointing only if it is in the safe state. In our design the 2-phase coordinated checkpoint commit protocol is executed in the kernel context of each process. In general we can have the code execute any protocol. We notice that the basic mechanism to checkpoint the private state of the process is much the same as one would use for migration. Hence we reuse the Aragorn mechanisms, which have been developed for migration, for much of this activity. Aragorn mechanisms are described in [15].

When we checkpoint pages we need to checkpoint the page contents as well as its meta information, in particular which container it belongs to. This is done by the owner of the page. There is a *page checkpointing kernel server* running on each node which will respond to requests to checkpoint pages which are owned by that node, given the container and page identification. An important issue here is the storage of the pages. There is a strong motivation here to store the checkpointed images of the pages in the memory in remote nodes rather than committing them to disk as the synchronous writing for each page to a disk would consume significant time. It is proposed therefore to implement remote copy storage as proposed in the ICARE[8] system. While saving checkpoints to disk is important especially for tolerating multiple faults, we may defer this (and do it asynchronously), thereby increasing the time in history that we may need to rollback to. This is a tradeoff between checkpointing overhead on the one hand and amount of lost computation on the other.

As for the DSM state, at recovery time one can always find out, by querying, which pages are currently being managed by whom and one can reassign page ownerships and thus reconstruct the DSM state. Thus no checkpointing of the DSM state itself is needed.

An interesting problem that remains is the state of some distributed IPC primitives. For instance at checkpoint time a lock may have been acquired by a task, which is currently executing in user mode. In some sense this is like a page that needs to be restored. We must store this information because when we rollback, we will need to recreate the state of holding that resource. One can of course chose not to checkpoint when a lock is being held, just as one may chose not to permit checkpoint when not in user mode. But in either case it is clear that for the checkpointing to be useful, the synchronization primitives being used by the process must support actions to read(checkpoint) and reset(recover) their state. In our design the Elrond module (see Section 2) will be engineered to support checkpointing and recovery of distributed locks.

4.2. Extending the Tracking Mechanisms

It is useful at this point to consider how the dependency tracking mechanism can be extended to support distributed locks and barriers. We propose to treat each of these as dependency causing entities and associate sequence numbers, checkpoints and DDVs with each of them. In particular each of the synchronization primitives generates a two way dependency between a task issuing the primitive and the entity (the lock or barrier, as the case may be). Since we do not checkpoint tasks in their kernel states, it is clear that we will not have a task checkpointed when it is waiting at a barrier. Similarly we will not checkpoint a task that is waiting on a lock.

4.3. Fault Analysis

While we can possibly be selective about checkpointing tasks only when in a safe state, we cannot be similarly choosy during faults - they happen when they happen. Fault analysis requires us to figure out what has been lost and needs to be restarted, what needs to be rolled back, and where to get the state information. We can find out what tasks and pages are where by querying some threads on the nodes. We can find out the state to recover to because we can see the timestamps stored in the checkpoints and the current DDVs for the entities that are alive, and use these in a standard analysis such as reachability analysis [5]. Then we need to initiate restart and rollback procedures for the entities identified.

4.4. Restart of Dead Entities

One can recover a dead entity to a previously stored checkpoint. It is fairly easy in our system to deal with process private state and memory, as the checkpoint has exactly all the information needed to rebuild this state. The container mechanism ensures that as long as we have stored the right VMA map with respect to the container and page IDs, we can restart at the same state. For pages too this is straight forward. For locks we go to a state where nobody is waiting on a lock, but someone may have acquired a lock, so we potentially need to reinitialize its state, from the checkpoint. At checkpoints barriers are always clean, because of our checkpointing constraint. So barriers just need a method to be used to reinitialize to an empty state.

4.5. Rollback of Live Entities

In the case of pages, locks and barriers, this is identical to a re-initialization of the internal state as for restart. In case of tasks much of the state restoration is similar to the restart procedure; but the task is sometimes in a state which makes

the rollback challenging. For instance the task may be at a barrier. If it is in kernel mode then it will be brought to a state in which it needs to participate in the rollback protocol. This involves for instance aborting any barrier or lock wait it may be performing. Thus several Elrond modules and page fault handlers may need to be aware of such exception situation.

4.6. Server States during Recovery

An important issue during the fault recovery phase arises from the fact that when a fault happens on one node another node may be in communication with that node - either for handling a page request or some synchronization request like a barrier. Or, unaware of a failure it may initiate a new communication with that node. Hence it is important to be able *to detect* and *to deal with* the internal states of kernel servers or kernel mode executions of certain tasks. Hence the kernel server modules that support page fault handling or processing of page requests or synchronization requests must be preemptible from their processing and forced to some passive state.

4.7. Performance Issues

An important factor that governs the success of any scheme for fault tolerance is the overhead on fault-free execution. It is clear from the description in Section 3 that the actions to be taken on each event that generates a dependency is itself small. Further, it is clear that we have to only execute these when generating a new dependency, rather than for every read or write operation. For instance, on getting a copy of a page to write, one can record a new dependency, but it does not have to be redone for every read or write after that in that interval to that page. The additional threads of the kernel servers or the synchronization primitives for handling faulty cases are only invoked in the case of faults and their existence is no overhead to the fault-free execution of the system.

The main component that consumes resources is likely to be related to storage of checkpoint images. We believe that an approach like that of ICARE[8] will be a reasonable method to achieve acceptable impact on fault free performance. The mechanisms not only reduce the time to checkpoint, but also provide redundancy which can potentially be exploited to decrease page access times.

5. Preliminary Results

In this section we report results obtained from the first version implementation of coordinated checkpointing. The recovery phase is still under development.

We implement coordinated checkpointing in the Kerrighed kernel and support system initiated incremental checkpointing. Our results are for a cluster with two to six nodes. Each node is a 200MHz Pentium III PC with 128MB RAM, 100Mbps Ethernet and a local hard disk.

The coordinated checkpointing is implemented using a kernel server on each node that coordinates with the local tasks to be checkpointed on the one hand and the coordinator on the other hand.

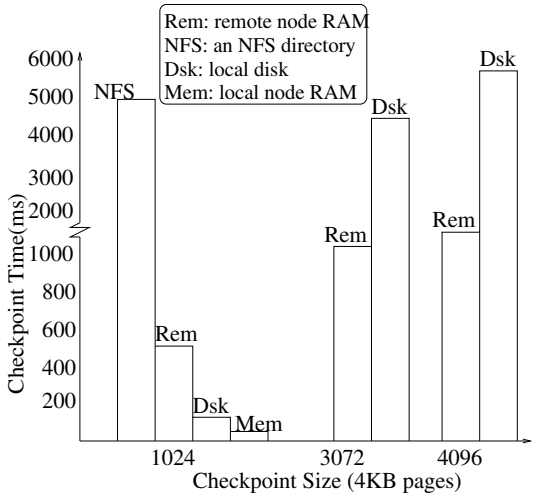

Figure 3. Effect of checkpoint destination

Figure 3 shows the effect of the choice of the location to save the checkpoint on the delay of checkpointing from the point of view of one of the nodes. It also shows how this time changes with the checkpoint size. These times were the first checkpoints for three applications instances: Gram-Schmidt on a 1024×1024 matrix, Jacobi on 1024×1024 matrix, and Gram-Schmidt on a 2048×2048 matrix, running on two nodes, thereby giving checkpoints on one node with size 1024, 3072 and 4096 pages (of 4KB each) respectively. In this figure the times are only for the shared DSM memory and do not include the cost of checkpointing the private task state. In the case of saving checkpoints on files (NFS or local disks) the times include the time to `write` as well as `sync` the data. In the case of checkpointing onto a remote node the times include the time to copy to a remote node and receive an acknowledgement from the remote. In each case we show the number of 4KB pages saved in the checkpoint. The first observation is that NFS is not a good choice for the location of the checkpoint. Further, as may be expected the time to make a copy to the local memory is the least. For small checkpoint sizes, the best option is to save the checkpoint on the local disk. It is interesting to note that as the checkpoint size increases, the time to checkpoint to the local disk increases greatly, but the time to checkpoint to a remote node does not increase at a similarly

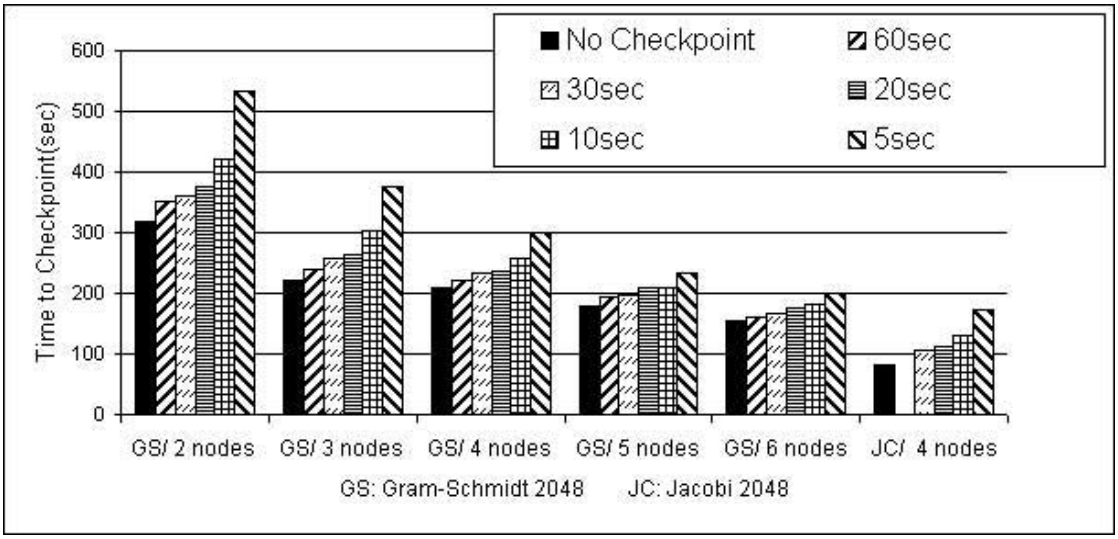

Figure 4. Effect of checkpoint interval

fast rate, hence making remote nodes the preferred choice for the checkpoint destination. In particular one could make one copy in the local memory and another on a remote node to provide the redundancy to help tolerate faults.

Figure 4 tries to demonstrate the scalability of the protocol, and give an idea of the fault free overhead on the running of the application. In the figure we have plotted the total execution time against the checkpoint frequency, for different cases. In this figure all the checkpoints are to local disk. In all except for one case we have the same application which is Gram-Schmidt for a 2048×2048 matrix of floating point numbers. The total amount of data checkpointed for this application is 8192 pages at the most (which is the first checkpoint). With 4KB pages this gives 32MB of data. But after the first checkpoint successive checkpoints are smaller, and therefore typically faster. In each case we note that the overhead of the checkpointing in terms of the fraction of increase in overall application execution time itself is quite low down to an interval of about 20 seconds. For example in the case of two nodes the overhead of checkpointing grows from 10% to 19% as we go from a checkpoint interval of 60 sec to an interval of 20 sec. In the case of six nodes this overhead grows from 5% to 14%. As the number of nodes increases, we have smaller amount of data to be checkpointed at each node, thereby reducing the absolute overhead of checkpointing.

In Figure 4 we have also shown one case of an another application namely Jacobi computation with matrices of 2048×2048 floating point numbers, on a four-node cluster. In this case the initial checkpoint is about thrice as large as the initial checkpoint of a 2048×2048 Gram-Schmidt, but successive checkpoints are of two third that size and stay that way up to the end of the computation. This is significantly different from the case of Gram-Schmidt where the checkpoint size decreases as the execution progresses. With the checkpoint interval of 30 seconds, total execution overhead is a significant 30%. This overhead should greatly reduce if one uses a faster means to place the checkpoint.

An interesting observation that does not show up in the above graphs is the overhead of checkpointing the task states (as opposed to the DSM). This itself tends to be quite small, and has nearly no impact on the checkpointing time. The other overhead that does not show up in the figures is the overhead of the checkpointing protocol itself. This is the overhead to synchronize the nodes in order to take a coordinated checkpoint. In our experiments with one thread on each node of the cluster, these two overheads together take nearly the same time for all cluster sizes studied and was found close to 20 milliseconds.

In summary, our experimental results show us that efficient coordinated checkpointing is possible using a version of the mechanisms outlined in this paper. Also, there is clear indication that the destination of the checkpoint will play a major roll in determining the fault free overhead.

6. Conclusion

In this paper we have presented the important aspects of our design of fault tolerance mechanisms for a DSM cluster system. In particular we have shown the utility of extending the dependency tracking scheme to cover objects such as locks and barriers. Further we have described herein specific support required from several kernel threads that the system comprises and the synchronization mechanisms. We have also presented preliminary performance results from our first implementation.

Our immediate goal is to implement recovery, and to augment support for synchronization primitives. Further in the future we would like to experiment with other protocols for checkpointing and recovery.

References

[1] R. Badrinath and C. Morin. Common Mechanisms For Supporting Fault Tolerance in DSM and Message Passing Systems. Technical Report RR-4613, INRIA, November 2002. http://www.inria.fr/rrrt/rr-4613.html.

[2] R. Baldoni, G. Cioffi, J. Helary, and M. Raynal. Direct dependency-based determination of consistent global checkpoints. *International Journal of Computer Systems Sciences and Engineering*, 16(1):43–49, 2001.

[3] R. Christodoulopoulou and A. Bilas. Dynamic data replication for tolerating single node failures in shared virtual memory clusters of workstations, June 2001.

[4] M. Costa, P. Guedes, M.Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *Operating Systems Design and Implementation*, pages 59–73, 1996.

[5] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[6] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Symposium on Reliable Distributed Systems*, pages 42–51, 1994.

[7] B. Janssens and W. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Symposium on Reliable Distributed Systems*, pages 34–41, 1994.

[8] A.-M. Kermarrec, C. Morin, and M. Bantre. Design, implementation and evaluation of ICARE: an efficicent recoverable DSM. *Software Practice and Experience*, 28(9):981–1010, July 1998.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computers*, 7(4):321–359, November 1989.

[10] R. Lottiaux and C. Morin. Containers: A sound basis for a true single system image. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[11] R. Lottiaux, C. Morin, and G. Vallée. Containers: An architecture for an efficient cluster operating system. Technical Report 1442, IRISA, February 2002.

[12] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transaction on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[13] F. Sultan, T. Nguyan, and L. Iftode. Scalable fault tolerant distributed shared memory. In *Supercomputing 2000 High Performance Networking and Computing Conference*, Nov. 2000.

[14] G. Suri, R. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *International Symposium on Fault-Tolerant Computing Systems*, pages 279–288, Pasadena, California, 1995.

[15] G. Vallée, C. Morin, J.-Y. Berthou, I. Malen, and R. Lottiaux. Process migration based on gobelins distributed shared memory. In *CCGRID 2002 - DSM 2002 Workshop*, pages 325–330, May 2002.

[16] Y. Wang, A. Lowry, and W. Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, 50:223–230, 1994.