TRANSPARENT FAULT-TOLERANCE IN PARALLEL ORCA PROGRAMS

M. Frans Kaashoek (kaashoek@cs.vu.nl)
Raymond Michiels (raymond@cs.vu.nl)
Henri E. Bal (bal@cs.vu.nl)
Andrew S. Tanenbaum (ast@cs.vu.nl)

Vrije Universiteit, Amsterdam
The Netherlands

ABSTRACT

With the advent of large-scale parallel computing systems, making parallel programs fault-tolerant becomes an important problem, because the probability of a failure increases with the number of processors. In this paper, we describe a very simple scheme for rendering a class of parallel Orca programs fault-tolerant. Also, we discuss our experience with implementing this scheme on Amoeba.

Our approach works for parallel applications that are not interactive. The approach is based on making a globally consistent checkpoint from time to time and rolling back to the last checkpoint when a processor fails. Making a consistent global checkpoint is easy in Orca, because its implementation is based on reliable broadcast. The advantages of our approach are its simplicity, ease of implementation, low overhead, and transparency to the Orca programmer.

1. INTRODUCTION

Designers of parallel languages frequently ignore fault tolerance. If one of the processors on which a parallel program runs crashes, the entire program fails and must be run again. For a small-scale parallel system with few processors, this may be acceptable, since processor crashes are unlikely. The execution-time overhead of fault tolerance and its implementation costs may not be worth the extra convenience. After all, the goal of parallelizing a program is to reduce the execution time.

Consider, however, a parallel program that runs on a thousand CPUs for 12 hours to predict tomorrow's weather. If the mean-time-between-failure of a CPU is a few years, the chances of one of them crashing can no longer be neglected. Moreover, if a processor crashes when the computation is almost finished, the whole program has to be started all over again, thus doubling its execution time. In general, the larger the scale of the parallel system, the more important fault tolerance becomes. Future large-scale parallel systems will have to take failures into account.

An obvious question is: Who will deal with processor crashes? There are two options. One is to have the programmer deal with them explicitly. Unfortunately, parallel programming is hard enough as it is, and fault tolerance will certainly add even more complexity. The alternative is to let the system (i.e., compiler, language run time system, and operating system) make programs fault-tolerant automatically, in a way transparent to programmers. The latter option is clearly preferable, but, in general, it is also hard to implement efficiently.

In this paper, we will discuss how transparent fault tolerance has been implemented in the Orca parallel language [Bal 1990]. Orca is a language for running parallel programs on distributed systems, such as the Amoeba system [Tanenbaum et al. 1990]. The failures that we consider are transient failures such as hardware errors.

The problem we have tried to solve is modest: to avoid having to restart long-running parallel Orca programs from scratch after each crash. The goal is to do this without bothering the programmer and without incurring significant overhead. We have not tried to solve the more general class of problems of making all distributed systems fault-tolerant. Instead, we consider only the class of parallel programs that take some input, compute for a long time, and then yield a result. Such programs, for example, do not interact with users or update files.

Our solution is simple: make global checkpoints of the global state of the parallel program using reliable broadcasting. If one of the processors crashes, the whole parallel program is restarted from the checkpoint rather than from the beginning. Users can specify the frequency of the checkpoints, but otherwise are relieved from any details in making their programs fault-tolerant.

The key issue is how to make a global checkpoint that is *consistent*, preferably without temporarily halting the entire program. It turns out that this problem can be solved in a surprisingly simple way in Orca.

The outline of the rest of the paper is as follows. We first give some information about the Orca language and its implementation on Amoeba. Next, Section 3 describes the design of a fault-tolerant Orca run time system. Section 4 gives the implementation details and the problems we encountered with Amoeba. In Section 5, we give some initial performance measurements. In particular, we show how much time it takes to make a checkpoint and to restart a program after a crash. In addition, we measure the overhead of checkpoints on example Orca programs. Section 6 compares our method with related approaches, such as explicit fault-tolerant parallel programming, message logging, and others. Finally, in Section 7, we present some conclusions and see how our work can be applied to other systems.

2. THE ORCA LANGUAGE AND ITS IMPLEMENTATION

In this section we give a brief description of the Orca language and its implementation. The goal is just to give enough detail to make the rest of the paper understandable. More detailed descriptions are given elsewhere [Bal 1990; Bal et al. 1990, 1992].

2.1. Orca

Orca is a language for running parallel programs on distributed systems. Although Orca is intended for systems without physical shared memory, its programming model is based on shared data. Processes in Orca communicate through *shared data-objects*, which are variables of abstract data types. Processes can share objects even if they run on different machines. The objects are accessed solely through the operations defined by the abstract data type.

Initially, an Orca program consists of a single process, but new processes can be created explicitly through a *fork* statement. The parent process can pass any of its data-objects as a shared parameter to the child. In this case, the data-object will be shared between the parent and the child. The parent and child can communicate through this shared object, by executing the operations defined by the object's type.

The semantics of the model are very simple. All operations are applied to single objects, and all operations are executed indivisibly. The latter property simplifies programming, since users do not have to worry about what happens if two operations are applied simultaneously to the same object. In other words, mutual exclusion synchronization is done automatically.

The first property is a restriction, since it rules out atomic operations on collections of objects. This restriction, however, makes the model efficient to implement, because no complicated two-phase update protocols are needed. As it turns out, parallel applications seldom need atomic operations on multiple objects [Bal 1990]. If needed, however, they can be constructed in Orca, by building them as sequences of simple operations. In this case, the programmer must explicitly deal with synchronization.

Orca is perhaps best thought of as a programming language approach to Distributed Shared Memory (DSM). Other systems (e.g., IVY [Li and Hudak 1989]) try to simulate physical shared memory on a distributed system and provide the same operations (read/write words) as real memory. Orca provides a DSM programming model, but the operations on the shared memory are defined by the programmer through abstract data types. Also note that Orca is *not* an object-oriented language; it is a procedural language with abstract data types. Unlike in concurrent object-oriented languages, objects in Orca are purely passive (hence the name data-objects). Furthermore Orca does not support inheritance.

2.2. A distributed implementation of Orca

Orca can be implemented efficiently on a distributed system using a run time system (RTS) that replicates shared objects in the local memories of the processors [Bal et al. 1989]. If a processor has a local copy of an object, it can do *read-operations* locally, without doing any communication. A read-operation is an operation that does not modify the object's local data; read-operations are distinguished from write-operations by the Orca compiler.

After a write-operation, the copies of the object will no longer be up-to-date. There are many ways of dealing with this situation. In the implementation described here, write-operations are broadcast to all nodes containing a copy. All these nodes update their copy by applying the write-operation to the copy.

A key problem is how to update all the copies of an object in a consistent way. We solve this problem using an *indivisible* reliable broadcast protocol. With such a protocol, all messages are delivered reliably at all receivers. In addition, the protocol guarantees that, if two processes P1 and P2 simultaneously try to broadcast two messages (say M1 and M2), then either all receivers get M1 first or all receivers get M2 first. Because the broadcast is indivisible, it is not possible that some processes will see M1 first while others get M2 first. Our protocol in fact assigns consecutive *sequence numbers* to the broadcast messages; each receiver handles the incoming messages exactly in the order of their sequence numbers. Due to space limit the protocol, its implementation, and its performance cannot be discussed in detail here, but readers are referred to [Kaashoek and Tanenbaum 1991; Tanenbaum et al. 1992].

2.3. Running Orca programs on Amoeba

The Orca implementation runs on the Amoeba distributed operating system [Tanenbaum et al. 1990; Mullender et al. 1990]. Amoeba is based on the processor pool model. Each user has his or her own workstation, but the real computing power is located in a pool of processors shared among all users. These processors are connected through a LAN and are allocated on demand. Processes can communicate with each other using at-most-once Remote Procedure Call (RPC) [Birrell and Nelson 1984] or using group communication [Kaashoek and Tanenbaum 1991]. The group communication primitives implement the broadcast protocol mentioned above.

Parallel Orca programs are run on the processor pool. An Orca application is started by the program *gax* (Group Amoeba eXecute). *Gax* asks the directory server for the capability for the program, and with the capability it fetches the Orca executable (Orca program linked with the Orca RTS) from the file server. Next, *gax* allocates the requested number of processors in the processor pool and starts the executable program (see Fig 1) on each processor. These Orca processes together form one process group. A message sent to the group is received in the same order by all processes (including the sending process), even in the presence of communication failures.

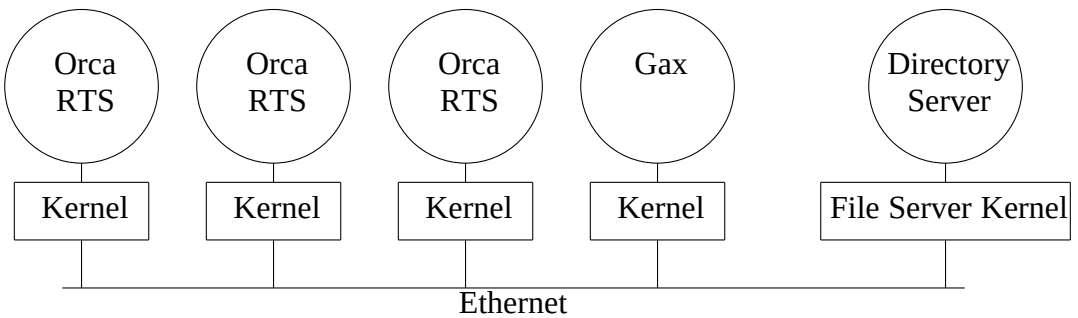

Fig. 1. An Amoeba system with an Orca application running on 3 processors. The Orca processes together form one process group.

3. DESIGN OF A FAULT-TOLERANT ORCA RUN TIME SYSTEM

There are many ways of making Orca programs fault-tolerant. One approach is to ask a special server to keep an eye on Orca applications and start them again if they fail. (In Amoeba such a server is available and is called the *boot server*). Unfortunately, this does not win much, since the application would have to start all over. For applications that have a deadline this is not appropriate.

Another approach is to use a message logging and playback scheme, such as optimistic recovery [Strom and Yemini 1985; Johnson 1989]. Message logging is designed for general distributed applications rather than just parallel applications, and may be too expensive for parallel applications. Optimistic recovery, for example, can deal with interactive programs. Programs using optimistic recovery will not ask for the same input twice or produce the same output twice. While this property is useful, it is not essential for most parallel programs, which frequently are not interactive. For those programs, the message logging solution is overkill. A cheaper and simpler form of fault tolerance is desired.

The method we use is to periodically make a global checkpoint of all the Orca processes. After a crash, all processes continue from the last checkpoint. Since parts of the program may be executed multiple times, the method cannot be used for interactive programs.

The most important design issue is how to obtain a *consistent* global checkpoint. As an example, assume a process P1 makes a checkpoint at time T1 and then sends a message to process P2. (Although Orca programs do not send messages, their run time systems do.) Assume that process P2 receives this message and then makes a checkpoint, at time T2. Obviously, the two checkpoints are not consistent. If both processes are set back to their state of their checkpoint, P1 would again send the message, but P2 would be in a state where it had already accepted the message. So, P2 would receive the message twice.

As explained in [Koo and Toueg 1987], checkpointing should be done consistently relative to communication. It certainly is not necessary to have all processors make checkpoints at exactly the same time, but messages must not cross checkpoints.

Based on this observation, one could envision making a consistent checkpoint by first telling each processor to stop sending messages. When the whole system is quiet, each processor is instructed to make its local checkpoint; all these checkpoints will then be consistent, since no interprocess communication takes place during checkpointing. We will not go into the details of how such a design might work. Suffice it to say that this solution would not be very attractive. The reason is that it may take a lot of time and overhead to bring the whole (distributed) system to a halt. Also, the more processors there are, the more time is wasted.

In the Orca run time system based on reliable broadcasting it is almost trivial to make consistent checkpoints. As we explained in Section 2, the Orca run time systems communicate through reliable indivisible broadcasting. All processes receive all broadcast messages in the same order. Therefore, to make the global checkpoint consistent, all that is needed is to broadcast one *make-checkpoint* message. This message will be inserted somewhere in the global order of broadcast messages. Assume that the

make-checkpoint message gets sequence number *N* in this global ordering. Then, at the time a process makes its local checkpoint, it will have received and handled messages 1 to *N*–1, but not messages *N*+1 and higher. This applies to *all* processes. Therefore the checkpoints are all consistent.

To recover from a processor crash all processes synchronously roll back to their last checkpoint. No process will send a message before all processes have been rolled back. Thus, when a process sends a message, all processes will receive it with the same sequence number. Because all processes roll back and synchronize before they continue running, they will start in a consistent state.

For some applications checkpoints clearly are not going to be cheap. Each process must save its local data (or, at the very least, the difference with the previous checkpoint). For some processes, this may easily involve writing several hundred kilobytes to a remote file server. Even with the Amoeba Bullet file server [Van Renesse et al. 1989], this may take a substantial fraction of a second.

With multiple processes, things will get worse. The time to make a global checkpoint will depend on the configuration of the distributed system and on the network. Clearly, having a single centralized Bullet server for a thousand pool processors will not be a good idea. However, the Bullet server can be (and is) replicated, so different pool processors can use different instantiations of this server.

The main advantage of our algorithm is that it is extremely simple and easy to implement. It is not an optimal algorithm, but it is still useful. In particular, for long-running parallel applications, the overhead of making a checkpoint every few minutes will be quite acceptable. An advantage of our scheme is that the frequency of the checkpointing can easily be changed. Using a lower frequency will decrease the overhead, but increases the average amount of re-execution due to crashes. We will get back to this performance issue in Section 5, where we give initial performance results.

4. IMPLEMENTATION ON AMOEBA

In this section, we will describe the problems encountered with implementing a fault-tolerant run time system for Orca on Amoeba. To understand the implementation we have to take a closer look at how process management is done in Amoeba. Each Amoeba kernel contains a simple process server. When *gax* starts an Orca application on a processor, it sends to the processor's process server a descriptor containing capabilities for the text, data, and stack segment. The process server fetches the segments from the file server, builds a process from the segments, and starts the process. The capability for the new process, called the *owner capability*, is returned to *gax*.

To checkpoint a single process, *gax* sends a *stun* signal to the process server that manages the process. The process server stops the signaled process when it is in a safe state, that is, when it is not in the middle of an RPC. (Interrupting a process while doing a RPC would break the at-most-once semantics.) When the process has stopped, the process server sends the process descriptor to the owner (*gax* in this case) and asks it what do to with the process. Using the capabilities in the processor descriptor, *gax* can copy the process state to the file server. After having copied the state, *gax* tells the process server to resume the process.

Making a global checkpoint of the complete Orca application now works in the following way. Every *s* seconds, a thread in the RTS of one of the machines broadcasts

a *make-checkpoint* message to all other processes in the application. When a process receives this globally ordered message, it asks *gax* to make a checkpoint of it, as described above. When all processes of the application have made a checkpoint, *gax* stores the capabilities for the checkpoints with the directory server. Other Amoeba servers will make replicas on multiple file servers using the capabilities stored with the directory server.

Rolling back to a previous checkpoint works as follows. If a member of the group that runs the Orca application crashes, the group communication primitives return an error after some period of time. When a process sees such an error, it asks *gax* to roll the application back. *Gax* starts by killing any surviving members and then starts the application again. Instead of using the original executable, it uses the checkpoints of the processes.

The actual implementation is more complicated due to a number of problems. The first problem is that by using *gax* we have introduced a single point of failure: if *gax* crashes, the Orca application cannot make any checkpoints or recover. To prevent this from happening, *gax* is registered with the boot service. The boot service checks at regular intervals whether *gax* is still there. If it is not there, the boot service starts *gax* over. (When *gax* starts running again, it kills the remaining processes and rolls the application back to the last checkpoint.) The boot service itself consists of multiple servers that check each other. As long as the number of failures at any point in time is smaller than the number of boot servers, the Orca application will continue running.

A second problem is that the checkpoints made by the process server do not contain all the state information about the parallel program. In particular, the kernel state information about process groups is not saved.

As an example, suppose the RTS initiates a global checkpoint by broadcasting a *make-checkpoint* message. At about the same time, a user process executes a write-operation on a shared object. As a result, its local RTS will send an *update* broadcast message and block the Orca process until this message has been handled. Assume that the broadcast protocol orders the *update* message just after the *make-checkpoint* message. The broadcast protocol will buffer this message in the kernel and it will be delivered after the checkpoint is made. If after a crash a process has been rolled back to this checkpoint, all the kernel information about the group is gone, including the buffered message.

Fortunately, detecting that a message has been sent and not received by any process when the checkpoint was made is easy. After a thread has sent a broadcast message, it is blocked until the message is received and processed by another thread in the same process. If after recovery there are any threads blocked waiting for such events, they are unblocked and send the message again.

The problem that the kernel information about the group is not included in a checkpoint is harder. We have solved this problem by having *gax* maintain a state file, in which it keeps track of additional status information. This file contains the current members of the process group, as well as the port names to which the restart messages (discussed below) are to be sent, the number of checkpoints made so far, and other miscellaneous information.

As a consequence of this approach, *gax* must read the status file during recovery and rebuild the process group. To rebuild the group, *gax* needs the help of the

processes that are being rolled back. These processes must actively join the newly formed process group. Clearly, all this activity is only needed during recovery and not after making a checkpoint. The problem is, it is difficult for the processes to distinguish between these two cases (i.e., resuming after making a checkpoint and resuming after recovery). After all, the state of the parallel program after recovery is the same as the state of the latest checkpoint.

Our solution to this problem is as follows. After making a checkpoint, a checkpoint server does not continue the process immediately, but it first waits for a *continue* message from *gax*. If it receives this message, it simply continues. On the other hand, if a processor has crashed and the program has just recovered, *gax* sends a *restart* message instead of the usual *continue*. If the checkpoint server receives a *restart*, it first participates in rebuilding the group state, before continuing the application.

Yet another performance issue concerns the text segment of a process. It is not necessary to dump the text (code) segment of each process, since text segments do not change and can be obtained by reading the executable file containing the process's image. At the expense of writing some more code, our prototype implementation avoids saving the text segment of a process after the first checkpoint.

Another important issue is the scalability of our implementation. The cost of broadcasting the *make-checkpoint* message is almost independent of the number of receivers [Tanenbaum et al. 1992], and therefore scales well. However, *gax* and the bullet server are likely to become a bottleneck as the number of processors increases. This could be avoided by using a distributed algorithm for making checkpoints, for example, by sending the checkpoints to a neighbour instead of to the bullet service.

Although the implementation is more complicated than we expected, only minor modifications were required to existing software. We added 324 lines of C-code (including 84 lines of comments) to the Orca RTS, bringing the total number of lines of C-code for the RTS at 6196. To the sources of *gax* we added 779 lines. No changes were made to the Amoeba kernel.

5. PERFORMANCE

We have measured the performance of the implementation described in the previous section. The Orca programs run on a collection of MC68030s. The directory server and the file server are duplicated. They run on Sun 3/60 and use a SCSI-3 controller and a WREN IV SCSI disk. All processors are connected by a 10 Mbit/s Ethernet. Given this environment, an Orca program running on n pool processors can tolerate 1 failure of the directory server or file server and $n-1$ concurrent processor pool failures. If both directory or both file servers crash, the Orca program will block until one of each is back up. If n concurrent failures happen, the Orca program will be restarted from the latest checkpoint by the boot server. Because all processors are connected by one network, the system will not operate if the network fails. If the system would have contained multiple networks we could have tolerated network failures, because the Amoeba routing protocol is based on dynamic routing tables [Kaashoek et al. 1991].

We have measured the overhead of checkpointing and recovery for a toy Orca program called *pingpong*. Pingpong consists of processes running on different processors and sharing one integer object. Each process in turn increments the shared integer and blocks until it is its turn to increment it again. This program is interesting because

it sends a large number of messages: for each increment of the shared integer the RTS sends one message. If this program were to run on a system based on message logging, it would experience a large overhead.

We ran pingpong with and without checkpointing and computed the time for making one checkpoint. The results are given in Fig. 2. Each process has 7 segments: one text segment of 91 Kbytes, one data segment of 56 Kbytes, and 5 stack segments of 11 Kbytes each. The state file is 3670 bytes large. The first checkpoint with text segment consists of 202 Kbytes and subsequent checkpoints (without text segment) are 111 Kbytes. If pingpong is running on 10 processors, taking a global checkpoint will involve writing 1.11 Mbytes to the file server. With 8 or more processors, the bullet service becomes a bottleneck, because it is only duplicated.

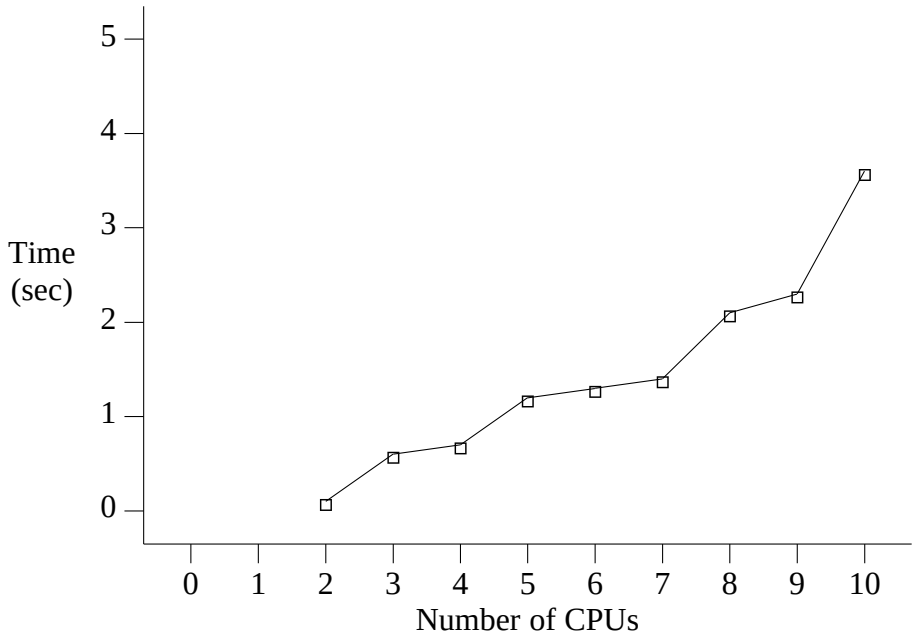

Fig. 2. The cost of making a checkpoint.

The time to recover from a processor crash is equal to the time to detect the processor crash plus the time to start *n* processes. The time to detect a processor crash is tunable by the user. The user can specify how often the other members in the group should be checked. The time to start a new process on Amoeba is 58 msec [Douglis et al. 1992]. Thus, most of the time for making a global checkpoint is spent in transferring segments from each processor to the file server.

For any application the total overhead (the difference between execution time with and without checkpointing) introduced by our scheme depends on 3 factors:

1) The cost of making a checkpoint (dependent on the number of processors and the size of the process's image);
2) The cost of a roll back;

3) The mean time to failure (MTTF) of the system (hardware and software).

If one wants to minimize the average run time of the application, then given the numbers for these factors, one can compute the optimal computing interval (the time between two checkpoints such that the average run time is minimized) and thus the overhead introduced by checkpointing (see Appendix):

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{(T_{comp} + T_{cp}) / 2 + T_{rb}}{T_{MTTF}},$$

where T_{comp} is the computing interval, T_{cp} is the mean time to make a checkpoint, T_{rb} is the mean time to recover from a crash.

If, for example, an application runs for 24 hours using 32 processors, the cost for a checkpoint is 15 seconds, time to roll back is 115 seconds, and the MTTF is 24 hours, then the optimal checkpoint interval is 27 minutes. In this case the overhead is 1.6 percent. If the MTTF is 7 days, then the optimal checkpoint interval is 71 minutes and the overhead is 0.7 percent.

6. COMPARISON

Although considerable research has been done both on parallel programming and on fault tolerance, few researchers have looked at fault-tolerant parallel programming. In this section, we will look at some of this work and also at more general techniques for fault tolerance.

6.1. Fault-tolerant parallel programming

An alternative to our approach is to let programmers deal with processor crashes. We have described our own experiences with explicit fault-tolerant parallel programming in a separate paper [Bal 1992]. The language used for these experiments was Argus [Liskov 1988]. Below we will first compare our work on Argus and Orca.

The Argus model is based on guardians, remote procedure calls (RPC), atomic transactions, and stable storage. A guardian is a collection of processes and data located on the same processor. Processes communicate through RPC. Programmers can define atomic objects, which are manipulated in atomic transactions (consisting of multiple RPC calls, possibly involving many different guardians). If a transaction commits, the new state of the atomic objects changed by the transaction is saved on stable storage. After a guardian crashes, it is recovered (possibly on a different processor) by first restoring its atomic objects and then executing a user-defined recovery procedure.

Our parallel Argus programs use multiple guardians, typically one per processor. Each guardian does part of the work and all guardians run in parallel. Guardians occasionally checkpoint important status information, by storing this information in atomic objects.

An important advantage of letting the programmer deal with fault tolerance is the increased efficiency. Our Argus programs, for example, only save those bits of state information that are essential for recovering the program after a failure. They do not checkpoint data structures that the programmer knows will not change, nor do they

save temporary (e.g., intermediate) results. In addition, it is frequently possible to recover only the guardian that failed, rather than all guardians (and processes) in the program. Finally, the programmer can control *when* checkpoints are made. For example, if a process has just computed important information that will be needed by other processes, it can write this information to stable storage immediately.

In Orca, programmers do not have these options. On the other hand, programming in Orca is much simpler, because fault tolerance is handled transparently by the system. For the parallel Argus programs, the extra programming effort varied significantly between applications. Some applications were very easy to handle. For example, a program using replicated worker style parallelism [Carriero et al. 1986] merely needs to write the jobs (tasks) of the workers to stable storage. If a worker crashes, the job it was working on is simply given to someone else, similar to the scheme described in [Bakken and Schlichting 1991]. Other applications, however, require much more effort. For parallel sorting, for example, a lot of coordination among the parallel processes is needed to obtain fault tolerance [Bal 1992].

Of course, there are many other language constructs that could be used for fault-tolerant parallel programming. Examples are: exception handlers, fault-tolerant Tuple Space [Xu 1988] and atomic broadcasts. The Amoeba broadcast protocol, for example, can tolerate processor crashes, so it can be used for building fault-tolerant applications [Kaashoek and Tanenbaum 1990]. Little experience in using these mechanisms for parallel programs is reported in the literature, however.

6.2. Other mechanisms providing transparent fault tolerance

Several other systems provide fault tolerance in a transparent way [Strom and Yemini 1985; Sistla and Welch 1989; Johnson 1989; Koo and Toueg 1987]. Most of these schemes are based on message logging and playback, usually in combination with periodic checkpoints. They are much more general than the method we described, in that they can also handle interactive distributed programs and sometimes can even give real-time guarantees about the system.

We compare our approach in more detail with one of the message logging approaches. We have chosen to compare it with Johnson's work [Johnson 1989], because it is implemented on the V system [Cheriton 1988], a system comparable to Amoeba, and he gives performance figures of his implementation. Johnson's approach is much more general than our approach (it can deal with interactions with the outside world) but is also much more complicated and harder to implement. It requires, for example, extensive modifications to the V kernel. Furthermore, it logs every message. This increases the cost for communication substantially (between 22 and 36 percent) and this introduces a fixed overhead for all applications. Using our scheme the overhead depends on the frequency of making checkpoints and is independent of the number of messages that an application sends. For parallel programs, this property is important, since such programs usually communicate a lot.

An interesting system related to ours is that of Kai Li [Li et al. 1990]. This system also makes periodic global checkpoints. Unlike ours, however, it does not delay the processes until the checkpoint is finished. Rather, it makes clever use of the Memory Management Unit. The idea is to make all pages that have to be dumped *read-only*. After this has been done, the application program is resumed and a *copier*

process is started in parallel, which writes the pages to disk. Since all pages are read-only to the application, there is no danger of losing consistency. If the application wants to modify a page, it gets a page-fault. The page-fault handler asks the copier process to checkpoint this page first. After this has been done, the page is made writable and the application is resumed. In this way, much of the checkpointing can overlap with the application.

In principle, we could have used this idea for making a checkpoint of a single process, but it would require extensive modifications to the sources of the memory management code and the way checkpoints are made in Amoeba. As we wanted to keep our implementation as simple as possible, we were not prepared to implement this optimization.

Another related approach is that of [Wu and Fuchs 1990]. In this paper, the authors describe a method to make a page based shared virtual memory fault-tolerant. Like our method is their method transparent to the programmer and is intended for long running parallel computation. In their method, the owner process of a modified page takes a checkpoint before sending the page to the process that requests it. Therefore, unlike our method, the frequency of checkpointing is determined by the patterns of data sharing. Frequent checkpointing occures if two process alternately write the same page.

7. CONCLUSION

We have described a very simple method for making parallel Orca programs fault-tolerant. The method is fully transparent, and works for parallel programs that take input, compute, and then yield a result. The method is not intended for interactive programs, nor for real-time applications.

Our method makes use of the fact that processes in the Orca implementation communicate through indivisible reliable broadcasting. In a system based on point-to-point message passing, it would be much harder to make a consistent checkpoint of the global state of the program. One approach might be to simulate indivisible broadcasting, for example by using the algorithm described in [Bal and Tanenbaum 1991]. This algorithm includes timestamp vectors in each message being sent, which are used in determining a consistent ordering. Another method might be to freeze the whole system before a checkpoint is made, but this introduces a performance penalty.

The paper also describes an actual implementation of our system, on top of the Amoeba distributed operating system. We have identified a number of problems with some Amoeba services, in particular the process server. We managed to get around these problems, but the implementation would have been much simpler if certain restrictions in Amoeba were removed. Finally, we have given initial performance results of our system, using a simple parallel application.

ACKNOWLEDGEMENTS

The authors would like to thank Dave Bakken, Leendert van Doorn, Fred Douglis, Elmootazbellah Nabil Elnozahy, Dick Grune, Cees Verstoep, and Cees Visser for their careful reading of the paper.

REFERENCES

Bakken, D. E. and Schlichting, R. D., "Tolerating Failures in the Bag-of-Tasks Programming Paradigm," *Proc. of the 21st International Symposium on Fault-Tolerant Computing*, pp. 248-255, Montreal, Canada, June 1991.

Bal, H. E., "Programming Distributed Systems," Silicon Press, Summit, NJ, 1990.

Bal, H. E., "Fault-tolerant Parallel Programming in Argus," *Concurrency Practice & Experience*, 1992. (accepted for publication)

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "A Distributed Implementation of the Shared Data-Object Model," *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL, Oct. 1989.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Experience with Distributed Programming in Orca," *IEEE CS Int. Conference on Computer Languages*, pp. 79-89, New Orleans, LA, Mar. 1990.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Orca: A language for Parallel Programming of Distributed Systems," *IEEE Transaction on Software Engineering*, 1992. (accepted for publication)

Bal, H. E. and Tanenbaum, A. S., "Distributed Programming with Shared Data," *Comp. Lang.*, Vol. 16, No. 2, pp. 129-146, 1991.

Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Call," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Carriero, N., Gelernter, D., and Leichter, J., "Distributed Data Structures in Linda," *Proc. 13th ACM Symp. Princ. Progr. Lang.*, pp. 236-242, St. Petersburg, FL, Jan. 1986.

Cheriton, D., "The V Distributed System," *Commun. ACM*, Vol. 31, No. 3, pp. 314-333, Mar. 1988.

Douglis, F., Kaashoek, M. F., Tanenbaum, A. S., and Ousterhout, J. K., "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, 1992. (accepted for publication)

Johnson, D. B., "Distributed System Fault Tolerance Using Message Logging and Checkpointing," TR89-101 (Ph.D. thesis), Rice University, Dec. 1989.

Kaashoek, M. F. and Tanenbaum, A. S., "Fault Tolerance Using Group Communication," *Proc. of 4th ACM SIGOPS European Workshop*, Bologna, Italy, Sep. 1990. (Also published in SIGOPS OSR, Vol. 25, No. 2)

Kaashoek, M. F. and Tanenbaum, A. S., "Group Communication in the Amoeba Distributed Operating System," *Proc. The 11th Internatinal Conference on Distributed Computer Systems*, pp. 222-230, IEEE Computer Society, Arlington, TA, May 1991.

Kaashoek, M. F., Van Renesse, R., Van Staveren, H., and Tanenbaum, A. S., "FLIP: an Internetwork Protocol for Supporting Distributed Systems," IR-251, Vrije Universiteit, Dept. of Math. and Comp. Sci., Amsterdam, June 1991. (submitted for publication)

Koo, R. and Toueg, S., "Checkpointing and Roll-back Recovery for Distributed Systems," *Trans. Soft. Eng.*, Vol. SE-13, No. 1, pp. 23-31, Jan. 1987.

Li, K. and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Syst.*, Vol. 7, No. 4, pp. 321-359, Nov. 1989.

Li, K., Naughton, J. F., and Plank, J. S., "Real-Time, Concurrent Checkpoint for Parallel Programs," *Proc. 2nd Symposium on Principles and Practice of Parallel Programming*, pp. 79-88, Seattle, WA, Mar. 1990.

Liskov, B., "Distributed Programming in Argus," *Comm. ACM*, Vol. 31, No. 3, pp. 300-312, Mar. 1988.

Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staveren, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23, No. 5, pp. 44-53, May 1990.

Sistla, A. P. and Welch, J. L., "Efficient Distributed Recovery Using Message Logging," *Proc. 8th ACM Symp. Princ. of Distr. Comp.*, pp. 223-238, Edmonton, Alberta, Aug. 1989.

Strom, R. and Yemini, S., "Optimistic Recovery in Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 3, No. 3, pp. 204-226, Aug. 1985.

Tanenbaum, A. S., Kaashoek, M. F., and Bal, H. E., "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, 1992. (accepted for publication)

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S., Jansen, A., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," *Commun. ACM*, Vol. 33, No. 12, pp. 46-63, Dec. 1990.

Van Renesse, R., Tanenbaum, A. S., and Wilschut, A., "The Design of a High-Performance File Server," *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, pp. 22-27, Newport Beach, CA, June 1989.

Wu, K-L. and Fuchs, W. K., "Recoverable Distributed Shared Virtual Memory," *IEEE Trans. on Comp.*, Vol. 39, No. 4, pp. 460-469, Apr. 1990.

Xu, A. S., "A Fault-tolerant Network Kernel for Linda," TR-424, M.I.T., Cambridge, Mass., Aug. 1988.

APPENDIX

In this section we will deduce the formula to compute the overhead of checkpointing and the optimal computing interval given the time to make a checkpoint, the time to recover from a crash, and the MTTF of the system.

For the derivation we introduce the following variables:

T_{tot} is the time needed to run the application without checkpointing;
T_{comp} is the time between two checkpoints;
T_{cp} is the mean time to make a global checkpoint;
T_{rb} is the mean time to roll back;
T_{MTTF} is the mean time to failure.

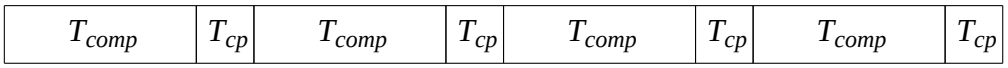

Fig. 3. An Orca execution without failures.

An execution of an Orca application without any failures consists of a sequence of alternating computing intervals of length T_{comp} and checkpoint intervals of length T_{cp} (see Fig. 3). If a failure happens during a computing interval or a checkpoint interval, an extra interval with duration equal to the recover time plus the time wasted before the crash is inserted (see Fig. 4). The mean time wasted can be estimated by $(T_{comp} + T_{cp}) / 2$.

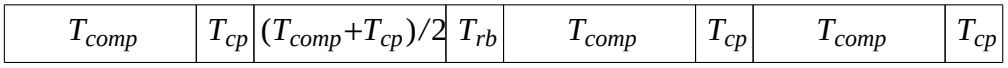

Fig. 4. An Orca execution with a failure.

Let T_{tot} be the total computation time needed to finish the application. Then, the number of computing intervals is equal to T_{tot} / T_{comp} and the number of failures during the total run time is approximately T_{tot} / T_{MTTF} (assuming $T_{cp} << T_{tot}$ and $T_{rb} << T_{tot}$). Thus the average run time is:

$$\frac{T_{tot}}{T_{comp}} (T_{comp} + T_{cp}) + \frac{T_{tot}}{T_{MTTF}} (\frac{T_{comp} + T_{cp}}{2} + T_{rb}).$$

The overhead for checkpointing is the average run time divided by T_{tot} minus 1:

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{\frac{T_{comp} + T_{cp}}{2} + T_{rb}}{T_{MTTF}}.$$

Minimizing the overhead function gives:

$$optimal\ computing\ interval = \sqrt{2T_{cp}T_{MTTF}}$$