Transparent Fault Tolerance for Parallel Applications on Networks of Workstations

Daniel J. Scales, DEC Western Research Laboratory
Monica S. Lam, Stanford University

To appear in the *Winter 1996 USENIX Technical Conference*, January, 1996

Transparent Fault Tolerance for Parallel Applications on Networks of Workstations

Daniel J. Scales and Monica S. Lam*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

This paper describes a new method for providing transparent fault tolerance for parallel applications on a network of workstations. We have designed our method in the context of shared object system called SAM, a portable run-time system which provides a global name space and automatic caching of shared data. SAM incorporates a novel design intended to address the problem of the high communication overheads in distributed memory environments and is implemented on a variety of distributed memory platforms. Our fundamental approach to providing fault tolerance is to ensure the replication of all data on more than one workstation using the dynamic caching already provided by SAM. The replicated data is accessible to the local processor like other cached data, making access to shared data faster and potentially offsetting some of the fault tolerance overhead. In addition, our method uses information available in SAM applications on how processes access shared data to enable several optimizations which reduce the fault-tolerance overhead.

We have built an implementation of our fault-tolerance method in SAM for heterogeneous networks of workstations running PVM3. In this paper, we present our fault-tolerance method and describe its implementation in detail. We give performance results and overhead numbers for several large SAM applications running on a cluster of Alpha workstations connected by an ATM network. Our method is successful in providing transparent fault tolerance for parallel applications running on a network of workstations and is unique in requiring no global synchronizations and no disk operations to a reliable file server.

*Author's current address: Digital Equipment Corporation Western Research Laboratory, 250 University Ave., Palo Alto, CA 94301.

This research was supported in part by DARPA contract DABT63-91-K-0003.

1 Introduction

Networks of workstations linked by high-speed interconnects such as ATM networks look quite attractive as platforms for running parallel applications, especially since many institutions already have numerous workstations on employees' desks or in common areas that are often idle. To date, networks of workstations have been used for only a limited range of parallel applications, because of issues of programmability and efficiency. The basic message-passing primitives provided by systems such as PVM [18] support only a very low-level programming interface and are difficult to use for programming applications with complex communication patterns. Systems that support a global shared-memory model in software greatly ease programming of parallel applications, but can encounter efficiency problems because of the high cost of communication in workstation environments.

In addition, programmers must deal with the possibility that individual processes or workstations participating in a parallel computation may fail. Workstations are often administered by the individual owners or reside in public areas. It is therefore not uncommon for workstations to be rebooted without notice. Because workstations are a shared resource, processes on a workstation may fail because they are explicitly killed by another user or exceed the resource limits of the workstation. An applications programmer using a cluster of workstations must either accept the loss in performance that results when an application run must be restarted after a failure, or attempt to deal explicitly with the possibility of failure in the application code.

SAM is a run-time system for distributed memory machines and networks of workstations that eases programming by providing a global name space and dynamic caching of shared data in software at the level of user-defined types (or *objects*) [15, 16]. SAM incorporates a novel design that retains many of the efficient properties of message passing and minimizes communication. The basic approach in SAM is to require the programmer to designate the way in which data will be accessed, thus allowing communication for synchronization and data access to be combined. Producer/consumer relationships are expressed by accesses to single-assignment values, and mutual exclusion constraints are represented by access to data items called accumulators. SAM currently runs on the CM-5, Intel Paragon, IBM SP2, and heterogeneous networks of workstations. Experience with SAM has shown that it significantly eases the programming of complex applications and allows the programmer to achieve good performance for these applications on a variety of distributed memory platforms.

This paper describes a method of providing fault tolerance in software for long-running parallel applications on networks of workstations written using SAM. Our method recovers from the common case of a small number of processors that fail by halting; it does not handle a global system failure or failures in which processors operate incorrectly. It is desirable that a fault-tolerance mechanism have transparency, low overhead, portability, and scalability. By transparency, we mean that the fault-tolerance mechanism can automatically be used for applications without any additional programming by the user. The fault-tolerance mechanism should have small overhead, so that the performance benefits of running an application across many workstations is not significantly reduced by the cost of providing fault tolerance. By portability, we mean that the fault-tolerance mechanism can be used on a variety of kinds of workstations without changes to the basic system software. Finally, the mechanism should be inherently scalable and must therefore minimize the need for global communication and synchronization.

Dynamic caching of shared data is an important part of the SAM design which makes it possible to exploit the data locality in parallel applications. Our fundamental method of providing fault tolerance is to use the dynamic caching functionality already provided by SAM to ensure, at checkpoints, that all shared data is replicated on more than one *host* (workstation). Our method thus provides fault tolerance without doing expensive writes to disk and does not require a common file server. Our integration of the fault tolerance method with the SAM system provides advantages of simplicity and efficiency. The implementation of the fault tolerance is simplified by using the existing caching mechanism. Because the replicated data is accessible to the local host like all other cached data, the apparent overhead of the fault tolerance method may be reduced by faster application access to shared data. In addition, though our method is applicable to any shared object system with caching, it uses information available about data accesses in SAM applications to reduce the fault tolerance overhead.

Our method runs on an arbitrary collection of workstations with standard operating systems and transparently recovers from the common case of a single (or a small number of) host failures without any user intervention. Only the process on the failed host must be restarted and redo some computation; other processes continue executing without any rollback. It also avoids global synchronizations among all the workstations. The main disadvantage of our method is that the amount of fault-tolerance overhead depends on the communication pattern of the application and can potentially be large. In addition, because of its technique of replicating data in the memory of other hosts, our method may increase the memory requirements of an application on each host.

We have implemented our method in the SAM system for heterogeneous collections of workstations, using PVM3 as the underlying message-passing layer. PVM3 provides portability to a wide variety of processor architectures and interconnects, and supports the basic functionality for detecting when process and host failures occur. Our implementation automatically provides transparent fault tolerance for all SAM applications in any such environment. In this paper, we describe our method in detail and provide performance results for several large parallel applications running on a network of workstations and using our fault tolerance method.

In the next section, we give a brief overview of the SAM system. We then describe the basic software approaches that have been taken for making parallel applications fault-tolerant. We describe the key ideas of our approach and its implementation in detail. We then provide performance results for several SAM applications running in parallel on workstations connected by an ATM network and analyze the overhead of our method for fault tolerance. Finally, we compare with other software distributed shared memory systems that have provided fault tolerance and conclude.

2 SAM Overview

In this section, we present a brief overview of SAM mainly via several examples. SAM is implemented as preprocessor and run-time system for C programs, but could be modified to work with FORTRAN-90 and C++ applications as well. A more detailed description is provided in [15, 16].

SAM is a software distributed shared memory system that provides a global name space for accessing shared

Shared Address Space		**Distributed Address Space + SAM**	
1) Mutual Exclusion			
wait(lock) a[1] = ...; a[34] = ...; signal(lock)		begin_update_accum(a) a[1] = ...; a[34] = ...; end_update_accum(a)	
2) Producer/consumer			
i=i+1 allocate a_i a_i = ... signal(flag$_i$)	j=j+1 wait(flag$_j$) ... = a_j	i=i+1 begin_create_value(a_i) a_i = ... end_create_value(a_i)	j=j+1 begin_use_value(a_j) ... = a_j end_use_value(a_j)
3) Finite buffer (size 4)			
i=i+1 mod 4 wait(buf$_i$) a_i = ... signal(flag$_i$)	j=j+1 mod 4 wait(flag$_j$) ... = a_j signal(buf$_j$)	i=i+1 begin_rename_value($a_{i-4} \rightarrow a_i$) a_i = ... end_rename_value(a_i)	j=j+1 begin_use_value(a_j) ... = a_j end_use_value(a_j) free_data(a_j)

Figure 1: Examples of Creating and Accessing Data

objects on distributed memory machines. In SAM, all shared data is communicated at the level of user-defined types (objects) and is represented by either a *value* or an *accumulator*. (SAM deals only with the management and communication of shared data; data that are completely local to a process can be managed by any appropriate method.) In Figure 1, we show several common idioms, as they would be expressed using semaphores on a shared address space machine and using SAM primitives. While the SAM primitives differ from the typical primitives on a shared memory machine, the complexity of expressing the examples using the two models is quite similar.

In the first example, mutual exclusion is required to protect updates to shared data. In SAM, an accumulator is used to represent a piece of data that is to be updated a number of times, and whose final value is independent of the order in which the updates occurs. SAM automatically migrates the accumulator between processes as necessary and ensures that a process does not access the accumulator until mutual exclusion is obtained. Updates to an accumulator must be encapsulated by the SAM primitives `begin_update_accum` and `end_update_accum`. The call to `begin_update_accum` returns a pointer by which the accumulator can be accessed. SAM supports the idiom of chaotic computation via primitives which provide read access to a "recent" value of the accumulator, which is not guaranteed to be the most current value of the accumulator. SAM maintains a cache in each process of versions of accumulators that have been recently accessed and may be able to satisfy the chaotic request locally without communication.

In the second example, a consumer (right column) accesses data created by a producer (left column). In SAM, a value provides producer/consumer synchronization. Values have a single-assignment semantics: a value is atomically created once its initial contents are set and is henceforth immutable. The code to create a value, which may include arbitrary updates to different components of the value, is encapsulated by a pair of primitives `begin_create_value` and `end_create_value`. Similarly, code accessing a value is encapsulated by the primitives `begin_use_value` and `end_use_value`. A process that attempts to access a value will automatically be suspended until the value is created and has been brought to the local process. Conversely, an access will succeed immediately if the value is already available in the local process, returning a pointer to the local copy.

SAM maintains copies of values fetched from remote processes in local main memory in the form of a cache. Because all values have distinct names and are immutable, there is no consistency problem associated with maintaining this cache. SAM automatically frees up local copies that are not in use in an LRU fashion when the cache memory becomes filled. SAM must ensure that at least one copy of a value is maintained in the system, until it can determine that there will not be any other processes that will need to access the value.

The SAM programmer provides this information by specifying the number of accesses to the value that will occur or explicitly indicating when all accesses to the value have occurred.

In the third example, a consumer accesses a sequence of values created by a producer through a limited-sized buffer. To avoid memory usage problems that are associated with single-assignment values, SAM allows one value to reuse the storage of another value via the `begin_rename_value` primitive. This primitive provides the necessary synchronization to ensure that the producer does not reuse the storage of a value before the consumer has accessed it. In this way, imperative data objects are easily represented in SAM via a sequence of values which can all share the same storage. SAM also allows a value to be converted to an accumulator and vice versa.

The creator of a value or an accumulator must specify the data type of the new object. With the help of a preprocessor, SAM uses this type information to allocate space for, pack (for sending in a message), unpack, and free the storage of the data. The preprocessor can handle complex C data types, including structures, arrays, and types that contain pointers and therefore are not necessarily stored in one contiguous block in memory. In heterogeneous environments, SAM also handles any necessary data conversion between dissimilar machines. Data is always transmitted in units of whole objects.

SAM provides two additional mechanisms for optimizing communication. First, it provides an operation for producers to *push* data to consumers. Second, it provides operations to do asynchronous fetches of objects, thus allowing optimizations such as prefetching in which communication is overlapped with computation or other communication. Each of these optimizations is well integrated into SAM's shared memory model.

3 Software Approaches for Tolerating Failures

In this section, we describe some of the common software methods for tolerating failures. We consider parallel applications in which a number of processes cooperate in doing a computation. To achieve parallelism, each process typically runs on a different host (workstation), so we will assume that there is one process on each host.[1] Each process may be described as going through a series of states during its lifetime, which is determined by its initial input parameters, the program code it is executing, and communications with other processes. Most software fault tolerance mechanisms operate by maintaining information about the state of each process in a parallel execution. When a process fails, the mechanism restarts the failed process (and potentially some of the other processes) in a previous state so that the parallel application can continue executing. The fault tolerance mechanism must ensure that the states of the restarted processes are consistent with each other and any remaining processes, so that the processes execute the remainder of the parallel application correctly (i.e. in the same way as some failure-free execution of the application.)

The most common mechanisms for allowing restoration of the state of a process are *checkpointing* and *logging*. A checkpointing mechanism saves essentially the entire state of the process at one point in time. A logging mechanism saves information about changes to the state of the process and may be viewed as an incremental form of checkpointing. The checkpointing or logging information is most commonly saved to disk.

Checkpointing methods for providing fault tolerance in software for parallel applications may be divided broadly into two classes: *consistent checkpointing* methods and *independent checkpointing* methods. We illustrate these methods in Figure 2, where the arrows indicate communication between processes P1, P2, and P3, and the heavy horizontal bars indicate checkpoints. Figure 2(a) shows *consistent checkpointing* [6, 11], in which all processes periodically synchronize and checkpoint their state simultaneously. This method requires a global synchronization that ensures that all processes are in a consistent state (in which all messages sent by one process have been received by the destination process). Then each process saves its entire state (processor registers and its entire virtual address space), typically to disk. If a failure occurs, then the computation is resumed by restoring *every* process to its state at the last checkpoint (as indicated by the dotted line) and restarting execution. The advantage of this approach is that it can recover from any number of host failures, if the checkpoint information is saved to reliable, nonvolatile storage. Also, when failures are very infrequent, the fault-tolerance overhead can be made very low by making the interval between checkpoints very large. The disadvantages are that global communication between all the hosts is required during a checkpoint, and the states of all processes are rolled back to the last checkpoint during a recovery.

Independent checkpointing methods avoid global synchronizations and allow the state of each process to be checkpointed or logged independently. To ensure that a process can be restarted from its last check-

[1] In a number of our applications, each process is actually executing several tasks sharing the same address space. Our fault-tolerance mechanism is unaffected by the use of these light-weight tasks, except for the existence of multiple stacks.

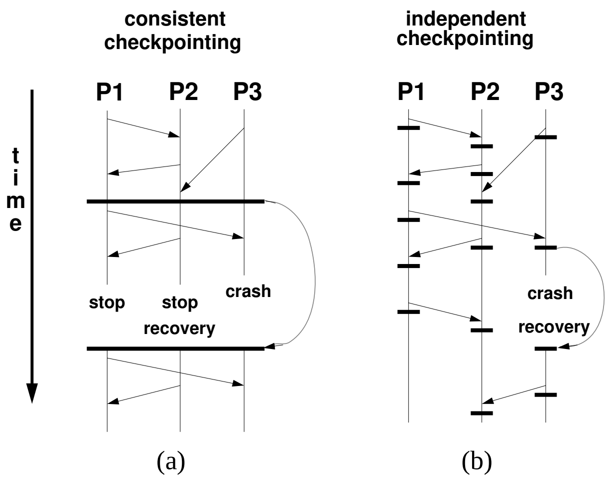


Figure 2: Consistent (a) and Independent (b) Checkpointing

point without affecting other processes, independent checkpointing methods generally require that each process saves its state whenever it communicates with other processes. The independent checkpointing methods divide into two classes, optimistic and pessimistic methods. In the pessimistic method [4], when a process communicates with another process, it atomically checkpoints or logs its incremental state change with the communication. It is then guaranteed that the process can be restarted from its last checkpoint or last set of logging records without requiring any retransmission of communications from other processes and without sending out a duplicate of any communication. The process can therefore be recovered without affecting any other processes. Figure 2(b) illustrates the use of pessimistic independent checkpointing to recover from the crash of a host. The advantage of this approach is that no global synchronization is required, and only the work of the failed process is rolled back. The disadvantage is that fault-tolerance overhead depends on the pattern of communication and can be large.

In the optimistic method [9, 17], a process also checkpoints or logs information when it communicates with another process, but the checkpointing or logging is not required to occur atomically with the communication. Because of this fact, it is not guaranteed that a process can be recovered without affecting the state of other processes. Instead, it may be necessary to rollback the state of other processes which have processed information sent by the process that failed. Dependence information is maintained by all processes to determine which processes need to be rolled back during a recovery.

4 Our Approach

Our approach applies the concepts of pessimistic, independent checkpointing to the SAM system. That is, when a process fails, the recovery procedure involves only restarting that process at a previous state (on the same host or a different host). All other processes proceed normally without rollback. The failed process appears to the other processes during the failure and recovery as if it has become very slow, but otherwise does not affect their execution. However, unlike a naive independent checkpointing approach, our method does not require a checkpoint every time a process is involved in a communication; based on information available about data accesses in SAM applications, it requires checkpoints for only a subset of the communications. In addition, we take the novel approach of checkpointing the state of a process by ensuring that its data is replicated to other processes using the caching mechanism of SAM, instead of saving its data to disk. Using information maintained by SAM, we reduce the amount of state that must be saved by only replicating the objects that have changed since the last checkpoint. In the following sections, we describe our approach in detail. We first describe when a process must do a checkpoint and what state information must be preserved at each checkpoint. We next describe the memory management of data that has been replicated for fault-tolerance purposes. Then we give step-by-step descriptions of the checkpointing procedure and the procedure for recovering from a failure.

4.1 Determining When To Checkpoint

During a parallel execution, each application process communicates with and affects the execution of the other processes. A process restarted after a failure must interact with the other processes in the same way as if the process never failed; otherwise, inappropriate messages sent by the restarted process will cause other processes to execute incorrectly. In a system such as SAM with a shared-memory model, one process can affect the execution of another process only by creating or modifying shared data which is accessed by the other process. A restarted process will therefore interact with other processes as if it never failed if it satisfies the following condition: any shared data produced by the original process that was communicated to other processes (and is still accessible) has the same contents in the restarted process. This condition is sufficient for ensuring a proper execution and is in general necessary, since processes may potentially access any data created or modified by the original process. Our fault-tolerance method must checkpoint sufficiently so that, at any point in the execution of a process, it can restart the process from a checkpointed state that satisfies the above condition.

The concept of a reexecutable operation is important in deciding when checkpoints are necessary. Suppose an operation is executed when a process is in a particular state and then the process fails. The operation is *reexecutable* if the operation is guaranteed to produce identical effects (on the current process and other processes) when executed from the same state in a restarted process. Any operation which only involves the local process is inherently reexecutable. However, operations that interact with sources external to the process, such as other processes or the operating system, may not be reexecutable. In parallel applications, most operations that are not reexecutable result from communication with other processes. For example, message sends in a message-passing program are not reexecutable, since the restarted process will send out a duplicate message. Similarly, message receives are not reexecutable, since the restarted process will wait for a message that the original process already received.

In the SAM system, a process only interacts with other processes by creating, modifying, or accessing shared data. The creation of a single-assignment value is reexecutable, because a value is not changed once created, and the restarted process will recreate the value with the same contents as in the original process. Similarly, accessing a value is a reexecutable operation, because any value that was accessed by the original process is guaranteed to have the same contents when the restarted process accesses the value again. Conversely, creating or updating an accumulator is not a reexecutable operation, because the accumulator may be modified by another process between the time it created or updated by the original process and the time when the restarted process repeats the operation.

If a process uses the results of an operation that is not reexecutable in creating or modifying a shared data object and there is no checkpoint between that operation and the creation of the data, then we classify the current contents of the shared object as *nonreproducible*. If the process is restarted from a checkpoint preceding the operation that is not reexecutable, then the resumed process may produce the shared object again with different contents than during the first execution. If the original contents of the object were communicated to another process, then the resumed process may produce data which is not consistent with the data that has already been sent to another process. Hence, the condition described above requires that we cause a checkpoint to occur *whenever nonreproducible data is sent to another process*. With this requirement, we know that the nonreproducible data will be recreated exactly from the checkpoint information if the process crashes. Conversely, if we communicate *reproducible data* to another process, we do not need to do a checkpoint. If the process crashes, the data will either be restored from the last checkpoint or it will be recreated with exactly the same contents as before when the process reexecutes from the last checkpoint state.

An important consequence of the above discussion is that an application that operates mainly with values will have few operations that are not reexecutable. The application will therefore mainly create reproducible data and processes will not usually have to checkpoint when sending data. In other distributed shared memory systems, any access to shared data is not a reexecutable operation, since the shared data may be modified by another process between the time of the access by the original process and the access by the restarted process. Any data that is modified is therefore not reproducible, and processes must checkpoint any time they transmit data that they modified to another process.

4.2 Information Preserved By a Checkpoint

The basic state of any process on a host may be divided into the shared data managed by the SAM system and the processor's private state. We can minimize the amount of state that must be preserved at each checkpoint by recognizing which data on a processor will be available from another process or can be reconstructed during a recovery.

An important observation about the shared data is that only one copy of each data object need be preserved by checkpointing, even though several hosts may be caching copies of the object. For simplicity, we therefore designate a *main copy* of each data object and only checkpoint the main copy of each object. If a process using a cached copy (other than the main copy) fails, then the process holding the main copy can send a copy of the object to the recovering process. (Section 4.3 describes our method of ensuring that the main copy is not freed while it might still be needed for a recovery.) In our implementation, the main copy of a value is the copy on the process that created the value. The main copy of an accumulator is the actual current contents of the accumulator, which migrates between processes. We define the *owner* of an accumulator or value as the process that currently holds the main copy.

The *private state* of a processor consists of the following:

- processor state (mainly registers)
- stacks of all active tasks in the process

- values of statically allocated global variables
- shared data objects in the process of being created or modified
- pending requests by the current process for data owned by other processes
- pending requests by other processes for data owned by the local process
- directory information about the location of shared objects owned by other processes

The first four items are the local state of the parallel application, and the remaining items are essentially the private state maintained by the SAM system on each processor. Each processor must save its private state at every checkpoint. However, pending requests for data from other processes need not be saved during a checkpoint, since the other processes can reissue the requests during the recovery process. Also, directory information on the location of shared objects owned by other processes does not have to be saved, since the owners of those objects can transmit that directory information during a recovery.

Instead of using checkpointing or logging to disk to preserve the state of a process, we achieve fault tolerance by ensuring the state is replicated in two or more processes. We accomplish the replication of the shared data objects using the caching mechanism already provided by SAM. The replicated data is therefore available for use by the local process, potentially offsetting some the cost of the fault tolerance scheme. We will refer to a cached copy of an object used for checkpointing purposes as a *checkpoint copy*. We also replicate the private state of a process to another process as well. We therefore entirely avoid expensive disk operations during checkpointing. We also avoid depending on the local disk of a host for holding checkpoint information and can restart a process on a new host when the old host has a permanent failure.

Much of the shared data owned by a process is likely not to change at every checkpoint. We therefore keep track of which accumulators or values have been created and which accumulators have been modified since the last checkpoint and only replicate those objects at the next checkpoint. Objects that have not changed have already been replicated in a previous checkpoint. We currently preserve the entire contents of the private state at every checkpoint; we could instead attempt to save only the incremental changes to the private state since the last checkpoint.

To guarantee that we can recover from the simultaneous or near-simultaneous failure of n hosts, we must ensure that the private state and shared data owned by a host is replicated at a minimum of n other hosts. In practice, because simultaneous failures are rare, we choose n to be 1, thereby minimizing the fault-tolerance overhead, but guaranteeing recovery only when one host failure occurs at a time. However, all of our techniques immediately generalize to the case where n is greater than one. To simplify the recovery procedure, we always replicate a particular object to a specific process which is determined directly from the name of the object. Similarly, we always replicate a process's private state to a specific process.

4.3 Memory Management of Replicated Data

Because the main copy and checkpoint copy of a data object are potentially needed to aid in a recovery, memory management of these copies must be handled specially. Normally, the SAM implementation can free the main copy when it determines (via user-provided information) that all accesses to the data have occurred. However, to allow for recovery from a failure, the SAM implementation must maintain the main copy somewhat longer, until every other process has done a checkpoint since its last access to that object. The reason is that some process which has accessed the object since its last checkpoint may crash. When the process is restarted from that checkpoint, the process will attempt to access the object. If the main copy has been freed up, then that access (hence the recovery procedure) will fail. Because the checkpoint copy is essentially a backup for the main copy (in case the process with the main copy is the one that fails), it can only be freed when the main copy is finally freed. These conditions on freeing the main and checkpoint copy are analogous to the conditions in other fault-tolerance methods on when checkpoint files and log records can be reclaimed.

We wish to avoid having to send any extra messages in order to determine when the main copy can be freed. Our approach is to mark the main copy as "freeable" at the point at which all accesses to it have occurred, but leave it in the shared object cache. At some point later, it will be replaced in the cache, because it is no longer being referenced. At that time, before actually freeing it, we must ensure that all processes have done a checkpoint since the copy was made freeable. The straightforward way of ensuring this fact is to send a message to each process and wait for a reply message indicating that it has done a checkpoint before freeing the copy. However, we have designed a technique which avoids these extra messages in almost all cases by piggybacking timestamp information on other fault-tolerance messages (message sent during checkpointing). The basic idea is to maintain information, in all processes, on the last known time at which each process checkpointed, and to keep this information updated by transmitting with each fault-tolerance message the time when the sender last checkpointed. However, because of the lack of a globally consistent time among the hosts, the actual method is more complex.

Our technique is as follows. Each process i maintains a "virtual time" counter which is incremented each time there is a checkpoint or a freeing of an object that it owns. Process i also maintains a time vector T_i of the last-known virtual times on each process. The i_{th} component of T_i is always the current virtual time on process i. The rest of the vector is kept relatively current by transmitting the sender's current time vector with each fault-tolerance message and using it to update the receiver's time vector. Each process i also maintains vectors $C_i = (c_{i1}, c_{i2}, ..., c_{i,n-1})$ and $D_i = (d_{i1}, d_{i2}, ..., d_{i,n-1})$. C_i is the value of the time vector T_i when the process last did a checkpoint. d_{ij} is the last known value of c_{ji} from process j and is maintained by transmitting the current value of c_{ji} with each fault-tolerance message from process j to process i. If the current value of c_{ji} is c, then process j has done a checkpoint since the time on process i was c. Therefore, if the current value of d_{ij} on process i is d, then process j has done a checkpoint since the time on processor i was d. We can thus use the D_i vector to determine if each process has checkpointed since a particular virtual time on process i.

Suppose that we are about to remove the main copy of an object from the cache, and it was first marked freeable at time f on process i. If all elements of D_i are greater than f, we can free the copy immediately. If not, we must delay freeing the copy and send a "force-checkpoint" message to each process j for which $d_{ij} < f$. If process j receives a force-checkpoint message and its value of c_{ji} is less than f, then it does a checkpoint, which ensures that c_{ji} becomes greater than or equal to f. If its value of c_{ji} is already greater or equal to f, then process j does not need to checkpoint. In either case, process j replies to process i with its current values of T_j and c_{ji}. Process i can free the main copy when it receives replies from all the processes to which it sent force-checkpoint messages. It also informs the process holding the checkpoint copy that the checkpoint copy can be freed. We will present statistics in Section 5 showing that these force-checkpoint messages and forced checkpoints are required very infrequently in our applications.

4.4 Checkpointing Procedure

We can now describe the process of checkpointing in detail. Checkpointing is a transaction in which we atomically save the private state of the process, replicate all the necessary data, and execute the operation that originally started the checkpoint. The transaction is accomplished by sending checkpoint copies to other processes initially in an "inactive" state, which makes the data inaccessible until the checkpoint commits. The steps of a checkpoint are as follows:

- Send a copy of the private state to a designated process.
- For each nonreproducible data object[2] owned by the current process which has not yet been checkpointed, send a checkpoint copy to a designated process in an "inactive" state.
- For each reproducible object owned by the current process which has not yet been checkpointed, send a checkpoint copy to a designated process (in an active state).
- Send the requested object (that caused the checkpoint) to the requesting process in an inactive state if it has not yet been sent to that process as a checkpoint copy.
- Wait for acknowledgements from the processes that received the private state or any inactive objects. Then "activate" all of the inactive objects by sending out a message to all of the processes with inactive objects.

While the local process is doing a checkpoint, it continues serving incoming requests for reproducible data, receiving data it has requested from other processes, and receiving replicated data sent by another process that is checkpointing. However, to ensure that each checkpoint is consistent, a process delays serving requests for nonreproducible data and responding to "activate messages" from other processes that have just completed a checkpoint. After the checkpoint has been committed, all of these delayed operations are executed, possibly causing another checkpoint in the process.

4.5 Recovering from a Failure

The goal of the recovery procedure is to restore the necessary state of the failed process for the SAM application to continue normally and to restore the necessary checkpoint information for the system to tolerate further failures. The state that must be restored includes

[2] In our current implementation, we do not detect operations other than access to accumulators that are not reexecutable. However, we could easily modify system libraries to catch the other kinds of operations, such as system calls, that are not reexecutable. None of the applications described in Section 5 perform these kinds of operations except for reading input files at startup.

all of the private state of the process, and all shared objects that were in use at the last checkpoint or whose main copy was on the failed process. The checkpoint information that must be restored are all the checkpoint copies that were maintained on the failed process.

Our system depends on the underlying message-passing layer to detect the failure of a process or host. Our current implementation runs on PVM3, which provides a notification message to surviving processes when another process fails. The recovery procedure when process p fails is as follows:

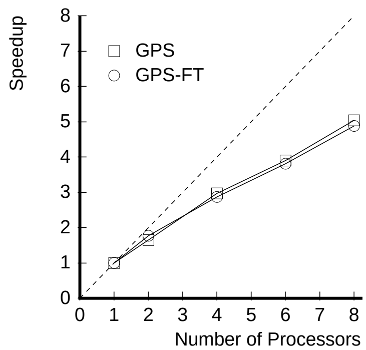
- One or more processes get a notification that process p failed. They send a message to a distinguished process d (process 0, or process 1 if process 0 is the process that failed) to coordinate the recovery.
- Process d receives the failure message and starts the recovery process, ignoring further messages indicating that process p has failed.
- Process d chooses a host on which to restart the failed process. This can be the same host, if it is still running or has been restarted, or a completely different host. Process d starts up a process in recovery mode on the new host.
- Process d sends a message to all other processes indicating that a recovery is in progress and including the PVM id of the new process. The new process will not receive any messages sent to the old process, since it has a different PVM id.
- The process holding the copy of the private state of the failed process sends that state to the new process.
- Each process besides the failed process aborts and restarts any checkpoint it has started that involves process p (since process p will have lost all information about that partial checkpoint).
- Each process besides the failed process also sends copies of data to the new process as follows:
  - If a process has a checkpoint copy of a data object whose main copy was at process p, it sends a copy of the data to process p, which will again hold the main copy.
  - If a process holds the main copy of a data object that was being accessed at the time of the last checkpoint by process p, it sends a copy of the data to process p.
  - If a process holds the main copy of an object whose checkpoint copy was on process p, it sends a checkpoint copy to process p.
  - If a process holds the main copy of a data item and the directory information for that data was on process p, the process informs process p that it holds the main copy.
- Each process reissues any requests for data that have been made to the failed process and have not yet been fulfilled.
- When the recovering process has received all of the data from other processes, it resumes its computation with its restored registers, stacks, etc.

5 Performance Results

We have implemented our fault tolerance method for SAM applications running on clusters of workstations using PVM3. When linked with our fault-tolerant SAM library, a parallel SAM application recovers transparently when we kill one of the processes involved in the computation. In this section we give performance results for three long-running applications–GPS, Water, and Barnes-Hut–executing on a cluster of eight 225 MHz Alpha workstations connected by an ATM network. The workstations are connected by a 155 Mbit/sec AN2 ATM network [1, 19] developed at DEC SRC, and use a version of PVM3 that is highly tuned for the AN2 network. The maximum achievable bandwidth under PVM3 is about 14.6 Mbytes/sec and the latency for sending a message from one host to another is about 90 μs. The timings for different runs of the applications are very consistent; the numbers below are from averaging the results for several runs, excluding the initialization phases, when there were no other users of the cluster. For each of the applications below, recovery from a failed process only takes on the order of a few seconds.

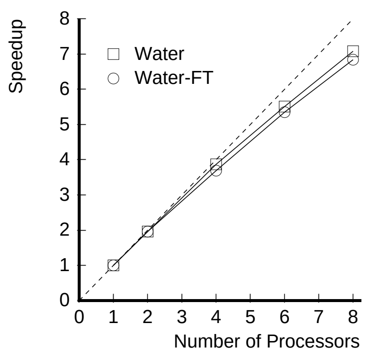
GPS is an application which attempts to determine a useful formula that predicts the degree of exposure to solvent of amino acids via a technique called genetic programming [7]. Genetic programming applications attempt to evolve useful formulas by emulating evolution and typically require large amounts of computer time. They build an initial population of individuals, which are candidate formulas, and then evolve the population over many generations using techniques analogous to genetic recombination, mutation, and survival of the fittest.

Figure 3 gives the speedup curves for the evolution of a population of 1000 individuals, both when not using fault tolerance (labeled GPS) and when using fault tolerance (labeled GPS-FT). The first three rows in the table to the right give the percent change in performance when fault tolerance is used in the runs, the average



	2 proc	4 proc	6 proc	8 proc
% change in performance	+10.8	-3.3	-2.3	-3.1
checkpoints per second	0.84	1.07	1.08	0.97
% sends causing checkpoints	7.0	24.3	25.0	24.9
force-ckpt. messages per sec.	0.00	0.00	0.00	0.00
forced checkpoints per sec.	0.00	0.00	0.00	0.00
shared data miss rate (%)				
without fault tolerance	18.4	35.4	39.7	46.9
with fault tolerance	2.5	23.5	32.8	41.1

Figure 3: Performance and Fault-Tolerance Statistics for GPS



	2 proc	4 proc	6 proc	8 proc
% change in performance	-0.8	-4.5	-3.0	-3.3
checkpoints per second	0.13	0.34	0.49	0.63
% sends causing checkpoints	100.0	100.0	100.0	100.0
force-ckpt. messages per sec.	0.00	0.02	0.04	0.05
forced checkpoints per sec.	0.00	0.02	0.04	0.05
shared data miss rate (%)				
without fault tolerance	9.0	17.1	20.7	22.7
with fault tolerance	7.6	15.2	18.8	20.8

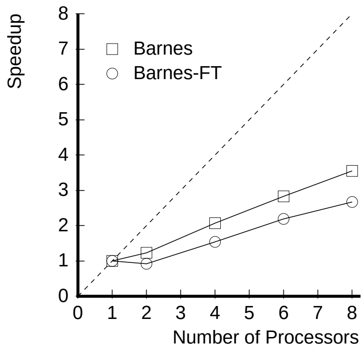
Figure 4: Performance and Fault-Tolerance Statistics for Water

number of checkpoints executed on each processor per second, and the percentage of sends of shared objects that cause checkpoints (equivalently, the percentage of sends of nonreproducible data). The low percentages indicate that many checkpoints are avoided and that the fault-tolerance overhead is much lower than it would be for systems which must assume that all data accesses are not reexecutable. The next two rows give the average number of “force-checkpoint” messages sent out on each processor per second and the average number of forced checkpoints on each processor per second. These operations, which are explained in Section 4.3, are never required for GPS runs. The final row gives the average miss rate on shared data both without and with fault tolerance. The miss rate is actually lower with fault tolerance because the replication of data during checkpoints can eliminate later misses to that data.

GPS is a relatively coarse-grained application, since there is much computation per individual in determining its fitness. In addition, the individuals of the population are evenly distributed across the processes, so the overhead of checkpointing the individuals is distributed well across processes. The fault-tolerance overhead is therefore quite low. GPS actually runs slightly faster on two hosts when using fault tolerance, because all the individuals created by one host are replicated to the other host and hence are immediately available when that host wants them.

The Water application evaluates forces and potentials in a system of water molecules in the liquid state. Water is derived from the Perfect Club benchmark MDG [3] and performs the same computation. Water is implemented using Jade [14], which is a parallel language implemented entirely in SAM. Since we have added fault tolerance to the SAM system, all Jade applications are also automatically fault-tolerant.

Figure 4 gives speedup curves for the simulation of 1728 particles, both when not using fault tolerance and when using fault tolerance, and with the same statistics as before. In Water, the main process collects all the data at each time step, so it must checkpoint most of the shared state of the system, which is quite large. The main process can therefore become a bottleneck with increasing processors, but the actual overheads remain quite small. Because of Jade’s load-balancing functionality, the distribution of tasks to processors involves an operation which is not reexecutable on the receiving process. Since tasks cause checkpoints only upon completion when they communicate their results, all data produced by these tasks is considered nonreproducible. Some force-checkpoint messages and forced checkpoints occur in Water, but the number is



	2 proc	4 proc	6 proc	8 proc
% change in performance	-25.0	-25.9	-22.4	-24.6
checkpoints per second	0.66	1.44	1.40	2.24
% sends causing checkpoints	16.3	6.8	5.9	5.3
force-ckpt messages per sec.	0.38	1.55	0.00	0.00
forced checkpoints per sec.	0.09	0.08	0.00	0.00
shared data miss rate (%)				
without fault tolerance	0.051	0.132	0.186	0.305
with fault tolerance	0.013	0.123	0.193	0.267

Figure 5: Performance and Fault-Tolerance Statistics for Barnes-Hut

very small. Again, the miss rate is actually lower with fault tolerance because the replication of objects during checkpoints eliminates some misses to these objects.

Barnes-Hut [2] is an application that simulates the evolution of an n-body system using a tree data structure to compute the forces between bodies in O($nlogn$) time. The SAM version of the code is based on the original serial version of the code and is written in a shared-memory style. All of the processes participate in the building and processing of the distributed tree data structure, and communication between processes is much more fine-grain than in GPS and Water.[3] The force computation is partitioned across processes so that each process has extensive locality in accessing the distributed tree, which is exploited by the dynamic caching provided by SAM.

Figure 5 gives speedup curves for a simulation of 8000 bodies. There is significant software overhead in implementing the Barnes-Hut algorithm in a shared-memory style on distributed memory machines, but this overhead only reduces the overall performance by a fixed percentage and performance scales well. Because of the fine-grain nature of the application and the large tree data structure that is built, the fault-tolerance overhead is quite high. The communication in building and modifying the tree causes many checkpoints, and the checkpointing process adds significantly to the latency of communication. If the Barnes-Hut application did not have strong locality which reduces the amount of fetching of remote data, the number of checkpoints would be even higher. The number of checkpoints is also greatly reduced by the high percentage of sends of reproducible data. Two conflicting factors affect the shared data miss rate when fault tolerance is used. As described above, the miss rate can be reduced if processors which have received checkpoint copies of an object then access that object. However, because of the extra checkpoint copies, the shared data cache may be utilized less efficiently for other objects. Because of these conflicting effects, the shared data miss rate decreases for some runs when fault tolerance is used, but increases for others.

Our performance numbers show that our fault-tolerance method has low overhead and is quite suitable for the coarse-grain applications that would typically be run on networks of workstations. The overhead of our method is larger for the finer-grain Barnes-Hut application, but the parallel performance still scales well despite the overhead.

[3]The average time between shared data misses on each process in 8-processor non-fault-tolerant runs is 109 ms for GPS, 484 ms for Water, and 5 ms for Barnes-Hut.

6 Related Work

In this section, we compare with recent work that have provided fault tolerance in the context of software distributed shared memory.

Like our work, Kaashoek et al. [10] implement fault tolerance for a shared object system called Orca. However, the methods for achieving fault tolerance are completely different. Kaashoek et al. use global checkpoints at periodic intervals to ensure that the system can be restarted from a previous state if there is a host failure. Checkpointing the state of the entire system consistently is fairly simple, because their implementation of Orca is based on reliable ordered broadcasting of all messages. Like our system, they provide fault tolerance transparently without any extra effort by the Orca programmer. Because they use global checkpointing, they can recover from failures of many hosts simultaneously (assuming that none of the disks used for checkpointing have permanent failures). However, their fault tolerance method depends completely on their use of reliable ordered broadcast, whose performance does not scale well for large numbers of hosts or networks with variable delays.

MOM [5] is a fault-tolerant implementation of a Linda-like programming model. The programmer uses the basic operations of Linda, but must sometimes specify additional information that is necessary only to support fault tolerance. In addition, the programmer is constrained to a model of parallel computation in which each task reads some input tuples and produces some output tuples, but otherwise does not interact with other tasks. Xu and Liskov [21] describe a design for a distributed implementation of Linda in which the tuple space is made highly available by replicating it across several processors. However, they do not consider the problem of restarting Linda tasks that were running on hosts that have failed.

Wu and Fuchs [20] describe techniques for making a page-based shared virtual memory system fault-tolerant. Like our work, Wu and Fuchs implement asynchronous pessimistic checkpointing by forcing a checkpoint whenever a host sends a page to another host in response to a request. They propose using two copies of each virtual memory page to allow checkpointing of pages to disk to proceed quickly. Janssens and Fuchs [8] apply this method to a DSM system with a relaxed memory consistency model. The expected checkpointing overhead is greatly reduced because a host need only checkpoint when it holds a synchronization variable that another host attempts to acquire. Richard and Singhal [13] describe a related approach in which a processor logs to disk the pages that it has accessed via read operations whenever it sends a modified page to another processor, instead of doing a full checkpoint. A processor must periodically checkpoint (independently of other processors) in order to reduce the size of the log.

Neves et al. [12] describe a method of providing fault-tolerance for an entry-consistent DSM system called DiSOM. Each process logs the contents of an object in its own local volatile memory, whenever it releases a write lock on the object. If another process acquires the object and later fails, then the data can be recovered from the process that holds the log entry. This technique is similar to our approach of not freeing data objects until we can guarantee that all other processes have checkpointed since their last access to the data. Each process must checkpoint to disk frequently to reclaim the memory used by the log entries.

Of the methods described above, only those associated with Orca and MOM have been implemented.

7 Conclusion

We have described a method of providing transparent fault tolerance for long-running parallel applications on networks of workstations in the context of the SAM shared object system. Our fundamental approach is to use the dynamic caching functionality provided by SAM to ensure the replication of all data on more than one host at checkpoints. Our method thus avoids expensive writes to disk and does not require a common file server. Each process checkpoints independently and only when sending data which is not reproducible to another process. The SAM design, which provides extra information on how data is accessed, is quite useful in reducing the number of checkpoints.

Our method works on heterogeneous clusters of workstations (which have a common message-passing layer such as PVM3), without changes to the system software and without doing any global synchronizations among the hosts. When there is a failure, only the work of the failed process must be redone from the last checkpoint; other processes continue without rollback. Interestingly, in a task-based application, checkpoints naturally occur at task boundaries when the results of tasks are transferred between processes, thus ensuring that only the work of the currently executing task must be redone during a recovery. We are investigating further optimizations to reduce the checkpoint overhead, such as using a more sophisticated scheme for determining where to replicate data or using available information to decrease the amount of state saved during a checkpoint.

It is interesting to note that our method is potentially useful for fast process migration. We can move a process in a parallel application off a particular host very quickly by killing it and restarting it on another host from its last checkpoint. Such fast process migration is useful when a parallel application is running on idle workstations that are on individuals' desks and an owner begins to use one of the workstations again.

We have implemented our method in SAM for workstation clusters using PVM3. Our method successfully recovers from process and host failures during long-running SAM applications. The amount of overhead is dependent on the communication patterns of an application and may be significant for applications with finer-grain communication such as Barnes-Hut. For the coarse-grain applications that are more likely to be run on networks of workstations, the overhead is quite low.

Acknowledgments

We thank Chandu Thekkath, Mike Burrows, and DEC SRC for allowing us to use the AN2 cluster and for providing a fast version of PVM3 for the cluster. We thank Simon Handley for the use of his GPS code and Martin Rinard for providing the Water code. We also thank the referees for comments that helped improve the paper.

Availability: The SAM system is available via anonymous FTP at `suif.stanford.edu` in `/pub/sam` and via the World Wide Web at `http://suif.stanford.edu`.

References

[1] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, Nov. 1993.

[2] J. E. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[3] M. Berry and et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[4] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 90–99, Oct. 1993.

[5] S. R. Cannon. Adding Fault-tolerant Transaction Processing to Linda. *Software – Practice and Experience*, 24(5):449–466, May 1994.

[6] K. M. Chandy and L. Lamport. Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[7] S. Handley. The Prediction of the Degree of Exposure to Solvent of Amino Acid Residues via Genetic Programming. In *Second International Conference on Intelligent Systems for Molecular Biology*, pages 156–160, Stanford, CA, 1994.

[8] B. Janssens and W. K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. In *Proceedings of the Twenty-third International Symposium on Fault-Tolerant Computing*, pages 155–163, July 1993.

[9] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Aug. 1988.

[10] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent Fault-Tolerance in Parallel Orca Programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 297–311, Mar. 1992.

[11] K. Li, , J. F. Naughton, and J. S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 2–11, Sept. 1991.

[12] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 121–129, Aug. 1994.

[13] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proceedings of the Twelfth Symposium on Reliable Distributed Systems*, pages 58–67, 1993.

[14] M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, Nov. 1992.

[15] D. J. Scales and M. S. Lam. An Efficient Shared Memory System for Distributed Memory Machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[16] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 101–114, Nov. 1994.

[17] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, Aug. 1985.

[18] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[19] C. P. Thacker and M. D. Schroeder. AN2 Switch Overview. In preparation.

[20] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, Apr. 1990.

[21] A. S. Xu and B. Liskov. A Design for a Fault-tolerant, Distributed Implementation of Linda. In *The Nineteenth International Symposium on Fault-Tolerant Computing*, pages 199–206, June 1989.

Author Information

Dan Scales is a researcher at Digital Equipment Corporation's Western Research Laboratory. His research interests include parallel languages and runtime systems, distributed systems, and software development environments. He received a BS in electrical engineering and computer science from Princeton University and a PhD in computer science from Stanford University in 1995. He may be contacted at `scales@pa.dec.com`.

Monica Lam is an associate professor in the Computer Science Department at Stanford University. Her research interests are in compilers, computer architectures, and parallel computing, and she currently leads the SUIF (Stanford University Intermediate Format) parallelizing compiler research project at Stanford. She received a BS degree from the University of British Columbia and a PhD in computer science from Carnegie Mellon University in 1987. She is the recipient of a National Science Foundation 1992 Young Investigator Award. She may be contacted at `lam@cs.stanford.edu`.