J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

PROGRAM DIAGNOSTICS

This article presents a key topic in program diagnostics: *checkpointing.* Checkpointing is the act of saving the state of a running program so that it may be reconstructed later in time. It is an important basic functionality in computing systems that paves the way for powerful diagnostic tools in many fields of computer science. This article provides a comprehensive overview of checkpointing in uniprocessor and parallel processing systems, including definitions, uses of checkpointing, and implementation details. Also included in this overview is a brief discussion of checkpoint consistency, which is a major concern in parallel processing systems, and a thorough discussion of issues related to the performance of checkpointing. It is intended that the reader of this article should receive a thorough grounding in checkpointing, with enough detail to implement an efficient checkpointer if so desired.

Overview of Checkpointing

A *checkpointer* is a tool that performs checkpointing: it saves the state of a running program. This facilitates the implementation of many diagnostic tools; they are listed in Table 1. Unless otherwise specified, we assume that checkpoints are saved on stable storage, such as a magnetic disk. There are three levels where checkpointing can be implemented. They differ in the level of user/programmer involvement:

(1) Operating System (*OS*) Checkpointing Here, checkpointing is performed by the operating system (*OS*). Typically, any program can be checkpointed by the operating system without any effort on the part of the programmer or user. For example, standard process pre-emption (i.e., making a process relinquish the central processing unit (*CPU*) and putting it on the ready queue) can be viewed as a simple form of OS checkpointing. Most operating systems do not implement checkpointing beyond process scheduling. There are a few notable exceptions, such as Unicos (1), KeyKOS (2) and fault-tolerant Mach (3), which implement rollback recovery, and Sprite (4), which is a distributed operating system that includes process migration as a primitive operation.

(2) User-Level, Transparent Checkpointing Here, checkpointing is performed by the program itself. Although other methods are possible, such as rewriting executable files (5), transparency is usually achieved by compiling the application program with a special checkpointing library. Because checkpointing is performed on top of, rather than in the operating system, the recoverability of the operating system state is an important issue. For example, most operating systems assign process ids in a manner that is unrecoverable. In other words, a recovering process cannot ask to be assigned a certain process id. Moreover, most operating systems do now allow user programs to checkpoint the state of the file system. These issues are often handled by restricting the program being checkpointed, so that it relies only on the recoverable system state. For example, programs that use read-only, write-only, or sequentially written read-write files can be checkpointed by storing the names and seek pointers of all open files at the time of checkpointing (6). Moreover, programs must assume that their process ids may change over time as the result of being checkpointed and restored.

Table 1. Functionalities Enabled by Checkpointing

Field	Functionality Facilitated by Checkpointing
Fault-tolerance	Rollback recovery
Debugging	Post-mortem and replay debugging
Parallel processing	Process migration and job-swapping
Software engineering	Elimination of boundary condition errors
Simulation	Virtual time

Table 2. Examples of Transparent Checkpointers

Name	Functionality	Computing Platform
Libckpt (9)	Fault-tolerance	Uniprocessors
Libckp (26)	Fault-tolerance	Uniprocessors
Condor (27)	Process migration	Uniprocessors
Igor (21)	Debugging	Uniprocessors
Manetho (28)	Fault-tolerance	Message-passing distributed systems
MIST/MPVM (29)	Fault-tolerance/migration	Message-passing distributed systems
CoCheck (30)	Fault-tolerance/migration	Message-passing distributed systems
(31)	Fault-tolerance	Distributed shared-memory systems
Ickp (16)	Fault-tolerance	Intel iPSC/860
CLIP (32)	Fault-tolerance	Intel Paragon

The restrictions do not affect the majority of applications that need checkpointing. For more discussion on the restrictions imposed on user-level checkpointers, see papers by Litzkow and Livny (7), Plank (8), and Plank et al. (9). There have been many transparent, user-level checkpointers written for various computing platforms. Examples are listed in Table 2.

(3) User-Level, Nontransparent Checkpointing Here, programmers actively incorporate checkpointing into their programs, often with the help of libraries and preprocessors. Nontransparent checkpointing obviously places a much larger burden on the programmer. This includes the burden of correctness, a liability that the transparent checkpointers do not possess. The tradeoff is in performance and flexibility. Programmers can specify the exact information that is needed for recovery, and thus checkpoint less information than transparent checkpointers. Moreover, with nontransparent checkpointing, programmers can save the checkpoints in a machine-independent format, which allows the checkpoints to be restored on machines of different architectures. This is impossible with current transparent checkpointers. Examples of nontransparent checkpointing systems are given in Table 3.

In this article, the focus is on transparent user-level checkpointers, although most of the discussion applies to the other checkpointers as well.

Table 3. Examples of Nontransparent Checkpointers

Name	Functionality	Computing Platform
Libft (26)	Fault-tolerance	Distributed systems
Dome (33)	Fault-tolerance/load balancing	Distributed systems
PUL (34)	Fault-tolerance	Distributed systems
Calypso (35)	Fault-tolerance/load balancing	Distributed systems
COSMOS (36)	Fault-tolerance	Distributed systems

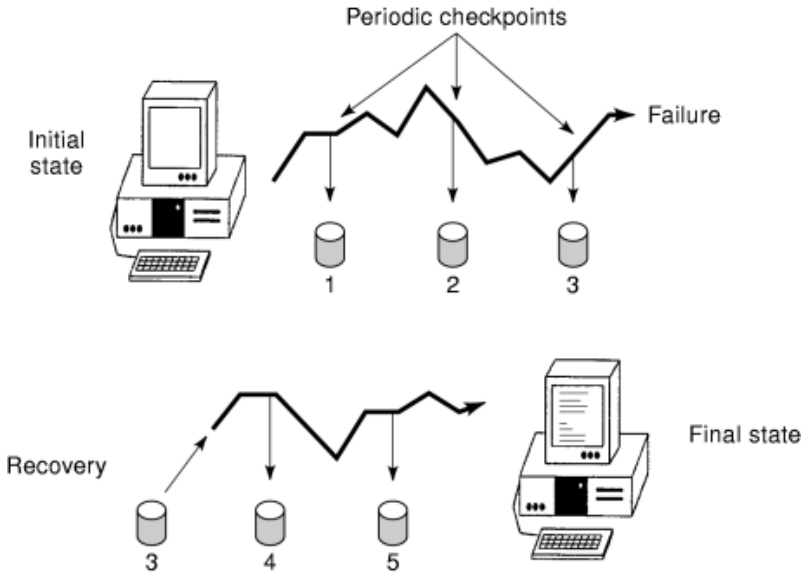

Fig. 1. Checkpointing for fault-tolerance. Periodic checkpoints are saved on stable storage to limit the amount of recomputation that must be performed upon recovery.

Uses of Checkpointing

Checkpointing provides the backbone for many tools, enumerated in the following text. This list is not exhaustive. For more uses of checkpointing, see Ref. 10.

Fault-Tolerance (Rollback Recovery). The major use for checkpointing is fault-tolerance (Fig. 1). This is typically called checkpointing with *rollback recovery*. At a periodic interval, the application stores checkpoints to disk. If a failure occurs that causes the application to be terminated prematurely, the application can restart from its most recent checkpoint, losing at most an interval's worth of computation.

As a fault-tolerant method, checkpointing is very powerful because it makes no assumptions about the type of failures that may occur. As long as the checkpointed state is failure free, it can tolerate hardware, software, and even power failures.

Migration. Process migration is another use of checkpointing. Instead of storing a checkpoint to disk, the checkpointing processor sends its checkpoint to another processor, which begins its computation from this checkpointed state. After sending the checkpoint, the initial processor terminates the application. Process

migration is useful for load balancing, in which a heavily loaded processor migrates its computation to a lightly loaded processor, thereby completing the computation more quickly.

Job Swapping. With job swapping, a processor stores a checkpoint to disk and then terminates the application. With job-swapping, a user may execute a long-running, computation-intensive program in short time intervals in order to share the computer with other users. A typical example is when a user executes an application only at night. In the morning, the application is checkpointed, and the computer is freed for other users.The next evening, the application is restarted from the checkpoint.

Post-Mortem and Replay Debugging. Checkpoints may be stored for purposes of debugging a program. For example, most debuggers have tools for examining checkpoints (called *core files*), which are created when a program exits abnormally. Often this examination allows the programmer to identify the offending bug, or at least to narrow the bug-finding search to a few suspicious subroutines.

Replay debugging is a more powerful functionality. Here, an application takes periodic checkpoints, and to aid in debugging, the program may be rolled back and replayed to *any* previous state. Checkpointing and replay may also be employed to track the state of variables and even complex data structures over time. The main drawback with replay debugging is the overhead induced by checkpointing.

Elimination of Boundary Condition Errors. A frequent number of programming errors in distributed systems occur when the system reaches a synchronization state that the programmer never envisioned. These states occur infrequently, and often as a result of *boundary conditions* that do not repeat themselves often. In such situations, checkpointing and rollback recovery can be employed to eliminate the effect of the bug (11). As in a fault-tolerant execution, checkpoints are taken periodically, and if an error occurs due to one of these boundary conditions, the system is rolled back to the previous checkpoint. It is statistically unlikely that the same boundary conditions will occur, and thus that the software will get into the same state that caused the bug. Thus, rollback recovery serves to eliminate the bug.

Virtual Time. Many simulation systems are modeled as a distributed set of objects that communicate by sending "event" messages. Objects process event messages, and communicate their output by sending more event messages. Unlike structured scientific programs, simulation systems have more nondeterministic message-sending patterns. At many points in time, an object may be faced with a dilemma—if there are no events coming in, then it should perform some computation. If there is an event on its way, then it should wait for the event and perform some different computation.

One solution to this dilemma is to use checkpointing. The object checkpoints itself and performs some computation. If an event then comes in that should have been processed before performing that computation, the object rolls back to the previous checkpoint and processes the event instead. The "Time Warp" simulation system works in this manner, and has been shown to be quite efficient for a variety of simulations (12,13).

Implementation Details

Conceptually, taking a transparent checkpoint of a process on a typical uniprocessor is simple (see Fig. 2). When a program is executing, its state is composed of the values in memory, the CPU registers, and the state of the operating system (including the file system). Usually, the memory is divided into four parts: executable code; global variables; heap; and stack. Of these, the global variables, heap, and stack need to be stored along with the registers in a checkpoint. Typically, the code is unchanged from the program's executable file, and thus may be restored from there in the event of a failure.

If the checkpointer is implemented in the operating system, then enough information can be stored with the checkpoint to restore the view that the program had to the operating system at the time of the checkpoint. If the checkpointer is implemented at user level, then it does not have the privileges to restore the operating system, but instead can attempt to make the operating system appear as it was at the time of the checkpoint. For example, the seek pointers of all open files may be stored with the checkpoint, and restored on recovery.

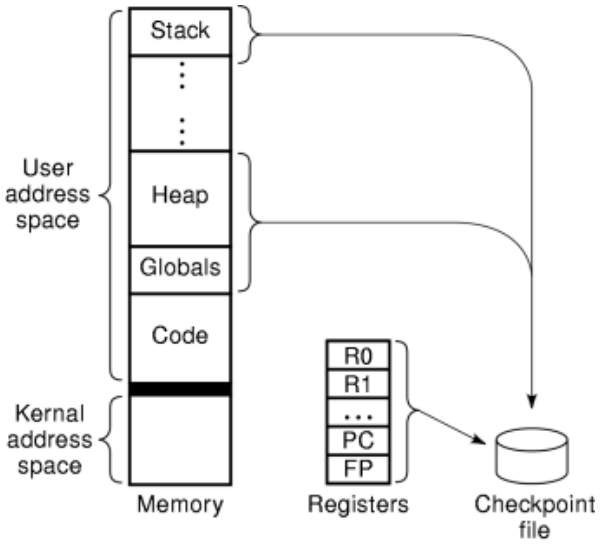

Fig. 2. Checkpointing a typical uniprocessor transparently. The portion of the user's address space that may be modified by the program is saved on stable storage.

Thus, even though the operating system's data structures for files may be different upon recovery, the view that the application has to the files is the same.

Part of the challenge in writing a transparent checkpointer is to be able to rebuild as much state that is external to the checkpointing process as possible. Besides operating systems internals and the file system, other external states that can be reconstructed include the window system and the state of external servers.

Most nontransparent checkpointers give the programmer primitives for storing and recovering data that is in the globals, stack, and heap. However, the programmer is responsible for restoring the execution state of the program (e.g., the call stack) upon recovery. Although this places a greater burden on the programmer, it can enable functionalities such as restoring the checkpoint on a machine of differing architecture from the checkpointing machine. This is because machine-dependent details such as memory layout and the definition of stack frames are not part of the checkpoint.

Processors with multiple CPUs attached to the same memory can be checkpointed in the same manner, except the register sets from each CPU must be included in the checkpoint. Moreover, the CPUs must all be frozen so that the register state of each CPU corresponds to the single checkpointed state of memory.

Checkpointing systems with multiple CPUs and multiple memories are more complex, because either there are no convenient ways to synchronize all the CPUs, or such synchronization is expensive. Moreover, in the absence of synchronization, the *checkpoint consistency* problem arises. This problem is important enough to warrant an in-depth discussion.

Checkpoint Consistency

Suppose a computing system is composed of n processors, each with their own memories, connected by an interconnection network. The only way that the processors can communicate is by sending and receiving messages over the network. Such is the case with standard distributed systems, and with distributed memory multiprocessors such as the Intel Paragon or IBM SP.

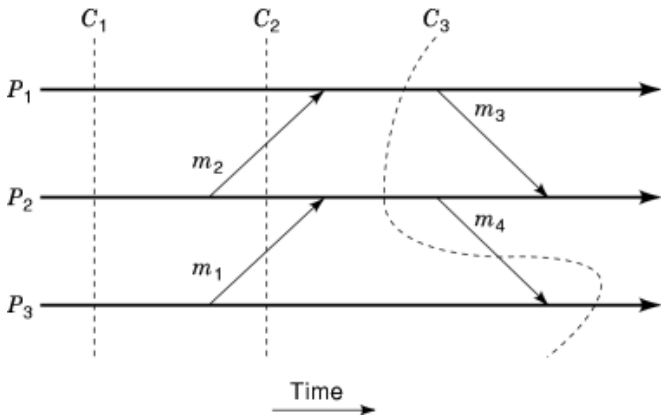

Fig. 3. A sample distributed system with three cuts. Cuts C_1 and C_2 are consistent. Message m_4 makes cut C_3 inconsistent.

The standard way of visualizing a distributed group of processors is to use horizontal lines that denote the relative progress of the processors over time as in Fig. 3. A message between processors is denoted by an arrow from the point at which the sending processor sends the message to the point at which the receiving processor receives it.

A *cut* is represented by a dotted line that crosses each process's time line exactly once. A message is said to cross the cut line from left to right if its sending point is to the left of the cut line, and its receiving point is to the right of the cut line. Crossing from right to left is defined in a similar manner. A *consistent* cut is a cut in which no messages cross the cut from right to left. Other cuts are called *inconsistent.*

Figure 3 shows a distributed system with three processors, four messages, and three cuts. The first two cuts (C_1 and C_2) are consistent as no messages cross them from right to left. The last (C_3) is inconsistent due to message m_4.

In order for a checkpointing system to operate correctly, it must ensure that *the system is always in a consistent state,* meaning that taken as a whole, the collection of states of all processors must compose a consistent cut. If the system is not in a consistent state, then upon recovery, there will be at least one processor whose state has been derived from the receipt of a message that has not been sent. There is no guarantee that the processor that is supposed to send such a message will ever actually send the message, due to nondeterminism either in the program or in the processor's interaction with other processors. Thus, inconsistent states are to be avoided.

During the normal operation of a distributed system, the processors are always in a consistent state because messages cannot be sent backwards in time. However, when failures occur and processors need to be rolled back, the checkpointing system must ensure that a processor rollback does not result in an inconsistent state.

For example, suppose each processor in Fig. 3 has taken checkpoints at the points where C_1, C_2 and C_3 cross its time line. If processor P_3 fails, it may be rolled back to its most recent checkpoint without affecting the other processors because the union of their states is a consistent cut. However, suppose processor P_1 fails. If it is rolled back to its most recent checkpoint, then the resulting state is inconsistent because m_3 crosses the cut line from right to left. To fix this problem, processor P_2 can be rolled back to its most recent checkpoint, but then m_4 renders the cut inconsistent. To roll the system back to its most recent consistent cut, P_3 additionally must be rolled back to its checkpoint from C_2.

This example illustrates a problem with *independent checkpointing,* where processors in a distributed system checkpoint themselves periodically without any coordination. The problem is that processors may have

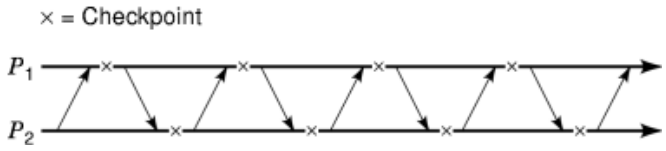

Fig. 4. The domino effect. A scenario like this one will force the program to restart from the beginning in the event of a failure, rendering all checkpoints useless.

to store multiple checkpoints because differing failure scenarios may require multiple rollbacks. A well-known pathological case has been called the *domino effect,* because each attempt to roll a processor back to a previous checkpoint forces another processor to roll back even farther. An example is in Fig. 4, where a failure of either processor forces the system to restart from the beginning, because no combination of checkpoints composes a consistent cut.

In general, there are two approaches to checkpointing distributed systems: coordinated checkpointing; and checkpointing with message logging. An overview of each is presented here. For further study on consistent checkpointing, including more detail on independent checkpointing, coordinated checkpointing, message logging and their interaction, see the survey by Elnozahy et al. (14).

Coordinated Checkpointing. With coordinated checkpointing, all processors cooperate to store a set of checkpoints that comprise a consistent cut. Moreover, any messages that cross the cut (from left to right) must be logged in stable storage. In the event of a failure, all processors roll back to the most recent checkpoint and the messages are re-sent from the log.

There are many advantages to coordinated checkpointing. First, as all errors force processors to roll back to the most recent checkpoint, only one checkpoint needs to be stored by each processor. There are times when a processor may hold two checkpoints, for example when it has finished checkpointing but other processors have not. However, once all processors have committed their checkpoints, all other checkpoints may be deleted. This is a great advantage over other checkpointing techniques. Second, recovery from a failure is simple. All processors must roll back. Many other checkpointing techniques require complex algorithms to determine the recovery state. Finally, any number of failures may be tolerated.

A simple technique for creating coordinated checkpoints is to use a two-phased commit protocol. This technique is sometimes called *sync-and-stop.* A master processor starts checkpointing by broadcasting a CHECKPOINT-BEGIN message to all other processors. Upon receiving this message, each processor halts computation and determines whether all messages that it has sent have been received. This can be done with special system calls or with acknowledgments. When a processor has determined that it has no outstanding messages in the network, it sends a CHECKPOINT-READY message back to the master. Upon receiving CHECKPOINT-READY messages from all processors, the master broadcasts a COMMIT-CHECKPOINT message, which instructs all processors to commit their checkpoints to disk. When a processor has committed its checkpoint, it sends a CHECKPOINT-COMMITTED message back to the master and resumes computation. When the master receives these messages from all processors, the checkpoint is finished and it notifies the processors that they may delete previous checkpoints.

The sync-and-stop technique guarantees a consistent cut because it ensures that there are no messages in the system at the time of checkpointing. Therefore no messages can cross the cut, either from left to right or right to left. As with coordinated checkpointing, the sync-and-stop technique is useful for its simplicity. It is straightforward to implement and requires no message logging or resending. Its major drawback is its synchronous nature, requiring all processors to freeze before checkpointing begins.

A less synchronous coordinated checkpointing technique is the well-known *Chandy–Lamport* algorithm (15). In this algorithm, each processor has a distinct set of neighbor processors. To communicate to a non-neighbor, a processor must send a message to a neighbor, which forwards the message onward. In the Chandy–

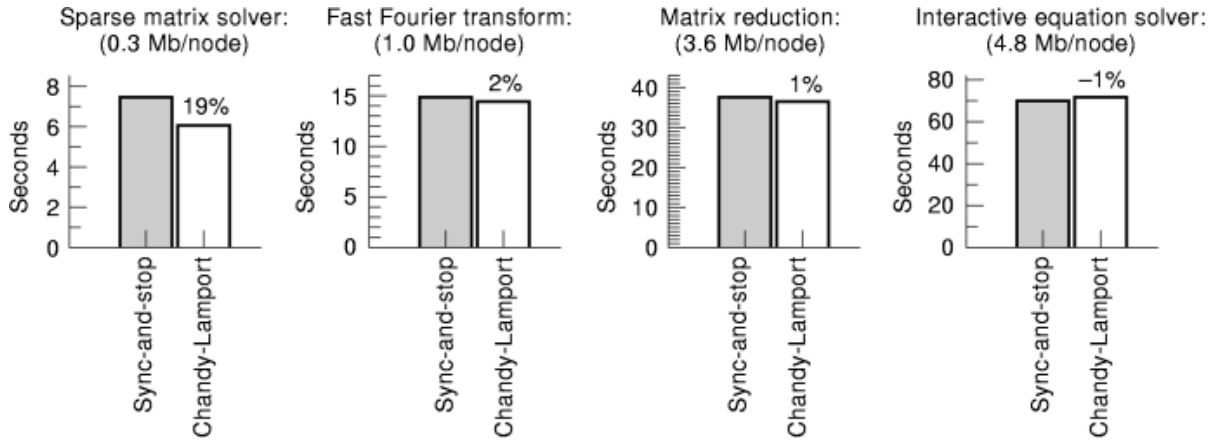

Fig. 5. Overhead of sync-and-stop vs. Chandy-Lamport on the Intel iPSC/860. Although sync-and-stop is more primitive and freezes all the processors, it does not degrade performance significantly in most applications.

Lamport algorithm, a master processor starts the algorithm by broadcasting a *marker* message to all its neighbors and then committing its checkpoint. All other processors begin the algorithm upon receipt of a marker message. They too broadcast a marker message to all their neighbors, and then commit a checkpoint. For each neighbor, a processor logs messages received from that neighbor from the time the processor starts checkpointing until the time a marker message is received from the neighbor. When markers have been received from all neighbors, no more messages have to be logged.

As with the sync-and-stop technique, the Chandy–Lamport algorithm ensures that checkpoints compose a consistent cut. Moreover, all messages that cross the cut are logged by the receiving processor. Upon failure, all processors roll back to their checkpoints and replay messages from their logs. Note that in a fully connected network with n processors, a total of $2n(n-1)$ marker messages are sent per checkpoint.

The Chandy–Lamport algorithm is more decentralized than the sync-and-stop algorithm, and appears to be preferable. However, in many distributed computing environments, the cost of writing checkpoints to stable storage is often much larger than that of coordinated message passing so that the consistency algorithm is immaterial. For example, a user-level, transparent checkpointer called ickp was written for the Intel iPSC/860, a hypercube-based multicomputer. Ickp implements both coordinated checkpointing techniques, and has been tested on many long-running scientific programs. Figure 5 shows the overhead of checkpointing four of these programs on 32 processors. Checkpoints are stored on the iPSC/860's special file system consisting of four disks connected to I/O processors. The overheads shown are averaged per checkpoint. In this and all other graphs, the percentages above the rightmost bars depict the percentage reduction in overhead versus the leftmost bars.

In this environment, the dominant cost of checkpointing is committing the checkpoints to disk, and as Fig. 5 shows, the coordinated checkpointing algorithm is largely immaterial. Full results of these tests are reported in Ref. 16 and similar results have been reported in Ref. 17. There are applications and computing environments where the Chandy–Lamport and other, more complex coordinated checkpointing techniques exhibit significantly better performance than the sync-and-stop technique (8). The reader is directed to Ref. 14 for further exploration into coordinated checkpointing.

Message Logging. Message logging algorithms assume that the programs executing on each processor are *piecewise deterministic*. This means that given an initial state and an ordered collection of incoming messages, the program always behaves the same way. With piecewise determinism, as long as a processor logs all the messages that it has received since a checkpoint along with the order in which these messages have

been received, then it can reconstruct *any state* following the checkpoint. Message logging algorithms make use of this fact to allow processors to checkpoint independently. As long as messages and message order are logged, a failing processor may be restored to any state up to the time of failure.

Message logging algorithms fall into two classes: *pessimistic* and *optimistic.* With pessimistic logging, messages are logged, either to stable storage or to a backup processor, before the receiving processor processes the message. This makes recovery simple and fast, affecting only the recovering processor. Care must be taken so that the processor and the logging site both receive messages in the same order.

Optimistic message logging attempts to reduce the failure-free overhead of message logging by pushing more complexity into recovery. For example, instead of logging actual messages, the receipt order may be logged, and the sending processor may either store the message in its log, or use a checkpoint of its own to regenerate the message. Optimistic message logging is often complex, necessitating each message to be piggybacked with dependency information. However, it usually shows better performance than pessimistic message logging; in addition, for applications where processors should not roll back past output operations (e.g., if a bank-teller program has directed one of its devices to output money, it is undesirable to have that program roll back and output the money again), optimistic message logging is the best method for fault-tolerance. As with coordinated checkpointing, Ref. 14 provides a good starting point for further exploration into message logging.

The Performance of Checkpointing

Obviously, correctness is the single most important aspect of a checkpointer. However, the second most important aspect is performance. There are many metrics by which performance can be measured, including checkpoint overhead, checkpoint latency, recovery time, and storage space. Of these, the most important is *overhead,* defined to be the time added to the running time of the application as a result of checkpointing.

It is desirable to have the overhead of checkpointing be as low as possible. Informally, if the overhead is too high, a user would rather risk failure than endure the performance penalties of checkpointing. Formally, if the failure rate of the computing system is known, then it is possible to compute the ideal interval of checkpointing for fault tolerance, given the overhead of checkpointing. Using simple assumptions, this interval is proportional to the square root of the overhead (18). More complicated assumptions and analyses can be used to hone this calculation (see Ref. 19), but the fact remains that lowering the overhead of checkpointing improves the fault-tolerant behavior of the application.

In situations other than fault-tolerance, the overhead of checkpointing is also important. For example, in debugging applications, reducing the overhead allows the granularity of checkpointing to be minimized, thus improving both the bug-free and tracing performance of the program. In simulation systems, reducing the overhead of checkpointing allows the system to be more aggressive in attempting to squeeze more parallelism out of the application.

There are many methods that can be used to improve upon the simple checkpointing implementing described in the preceding. These methods revolve around two simple concepts: *latency hiding;* and *checkpoint size reduction.* Opportunities for latency hiding arise because often the most time consuming portions of checkpointing, for example writing a checkpoint to disk, do not require use of the CPU. Thus, the CPU can be used to execute the application program concurrently with the writing of the checkpoint to disk.

Checkpoint size reduction revolves around the concept that smaller checkpoints take less time to store. Therefore, if checkpoints can be made smaller, then storing them to disk or to another processor should induce less overhead. In the text that follows, several checkpointing optimizations are detailed, along with some examples of the performance improvements that are possible.

Checkpoint Buffering. The simplest method to reduce the overhead of checkpointing is *checkpoint buffering.* This is a simple application of standard buffering to checkpointing systems and a latency hiding optimization. When a processor checkpoints, instead of freezing the application for the duration of the check-

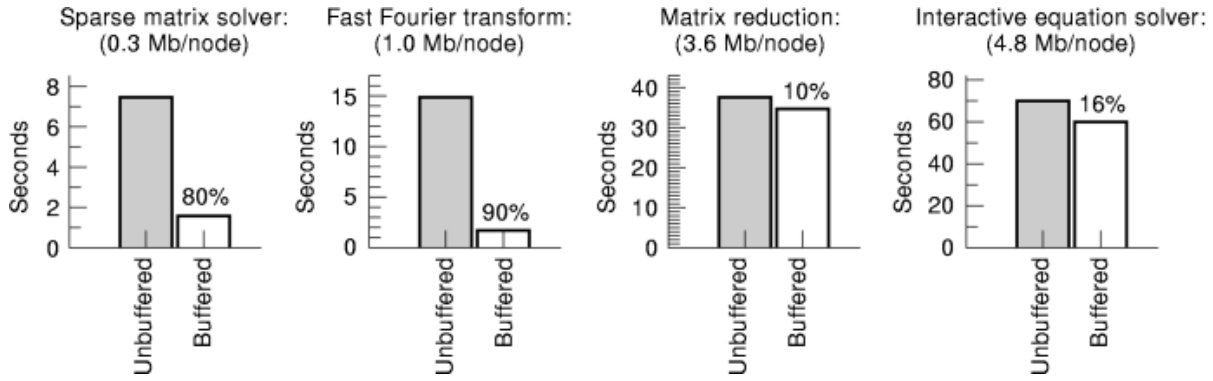

Fig. 6. Overhead of checkpoint buffering on the Intel iPSC/860. Buffering can improve greatly the overhead of checkpointing by allowing the application to continue while the checkpoint is written to disk.

point, the application is only frozen while a copy of the checkpoint is created in main memory. Once the copy is completed, the application may resume while the copy is stored on whichever external device (disk or network) is desired. Storing the checkpoint makes use of the direct memory access (*DMA*) primitives of most computers, and thus involves CPU intervention only at the beginning and the end. When the checkpoint is committed, the buffer may be discarded. If there is not enough physical memory to hold a complete checkpoint in memory, then as much of the checkpoint as possible is buffered to physical memory and the rest must be stored while the application is frozen.

With checkpoint buffering, the time to store a checkpoint is increased because a copy of the checkpoint must be made. However, the overhead can be decreased dramatically. For example, in Fig. 6, the overhead of the same four programs on the iPSC/860 (16) is plotted with and without checkpoint buffering. When buffering is enabled, a buffer of 1 Mbyte/node is employed. In the cases where the entire checkpoint fits into the buffer, the overhead is reduced dramatically. In the other cases, the improvement is not quite so dramatic, but is improvement nonetheless.

Copy-On-Write Buffering. A major inefficiency with checkpoint buffering is that if a region of memory does not change between the time it is copied to the buffer and the time the buffer is checkpointed, then the copying was not really necessary. This source of overhead is eliminated by copy-on-write buffering. Copy-on-write makes use of primitives for paged virtual memory. Specifically, on most computer systems, memory is partitioned into fixed sized *pages,* usually a power of two between 512 and 8192 bytes. Each page may have its protection set to be read-write, read-only or no-access. If a write operation is attempted on a memory location in a read-only page, or any memory operation is attempted on a location in a no-access page, then an *access violation* occurs, generating a hardware interrupt. With copy-on-write, it is assumed that the application is allowed to set the protection of its pages in memory, and that it may process the interrupts that result from access violations.

With copy-on-write buffering, the protection of all pages in memory is set to read-only at checkpoint time. Then the application is resumed, and a separate thread of control starts storing pages to the checkpoint. After a page is stored, its protection is reset to read-write. If the application generates an access violation, then the offending page is copied to a buffer, and the protection is set to read-write so that the application may continue. When the checkpointing thread goes to checkpoint such a page, it uses the copy from the buffer, and deallocates that page from the buffer when it has been stored in the checkpoint.

Copy-on-write buffering is the most consistently successful general-purpose optimization to reduce the overhead of checkpointing. Figure 7 shows the results of transparent OS checkpointing of distributed programs on a network of 16 diskless Sun 3/60 workstations (20). These graphs show that the improvements available

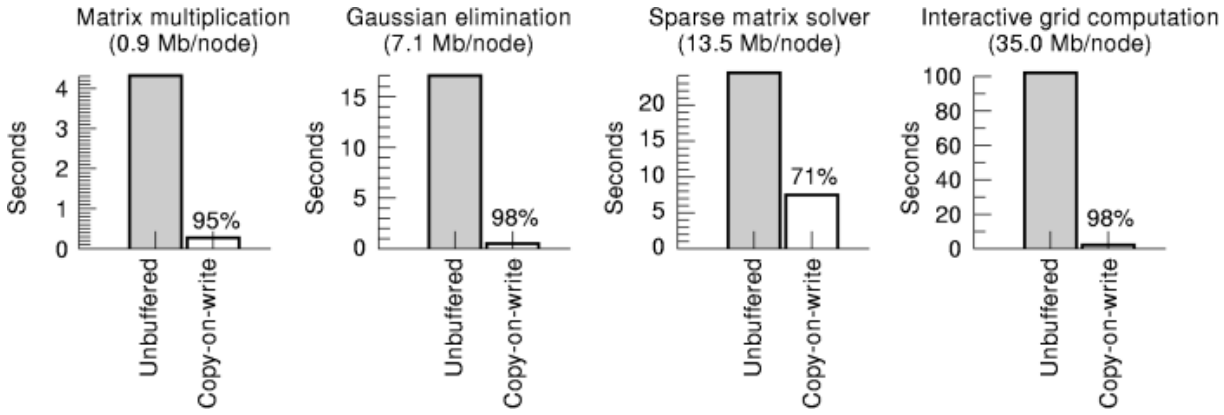

Fig. 7. Overhead of copy-on-write buffering on a network of Sun 3/60 workstations. Copy-on-write buffering improves performance in a manner similar to checkpoint buffering, but with fewer demands on physical memory.

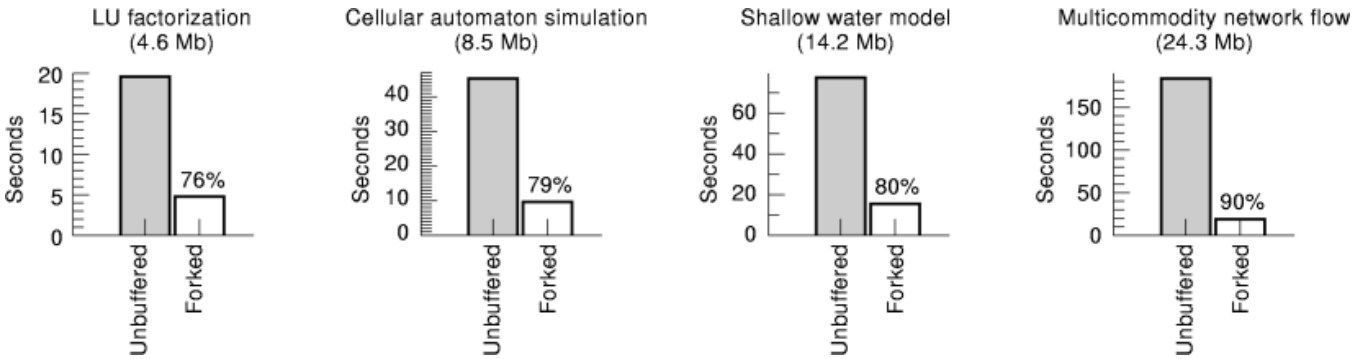

Fig. 8. Overhead of forked checkpointing on a Sun Sparc-2 workstation. Forked checkpointing takes advantage of the Unix fork() system call, which is implemented using copy-on-write in the operating system.

using copy-on-write can be quite dramatic. The degree of improvement is dependent on the memory access pattern of the program. Programs that cause many access violations show less improvement than those that cause few access violations.

User-level checkpointers often do not have access to the proper virtual memory primitives to implement copy-on-write checkpointing. In such cases, an alternative is to use the process cloning primitives provided by many operating systems. For example, in Unix systems the system call fork() performs this function. In most operating systems, process cloning is implemented using copy-on-write. Thus a user-level checkpointer may simply clone the application process and have the clone checkpoint itself while the original process continues the application. This is sometimes called "forked" checkpointing and is a very simple and portable implementation of copy-on-write checkpointing, yielding similar performance improvements.

For example, Fig. 8 shows the checkpoint overhead of a user-level checkpointer running on a Sun Sparc-2 workstation (9). Although the improvements are not as great as the copy-on-write examples in Fig. 7, they are still quite dramatic.

Memory Exclusion. Memory exclusion is a size reduction technique. There are two circumstances in which a variable may be excluded from a checkpoint: when the variable is *dead;* and when it is *read-only.*

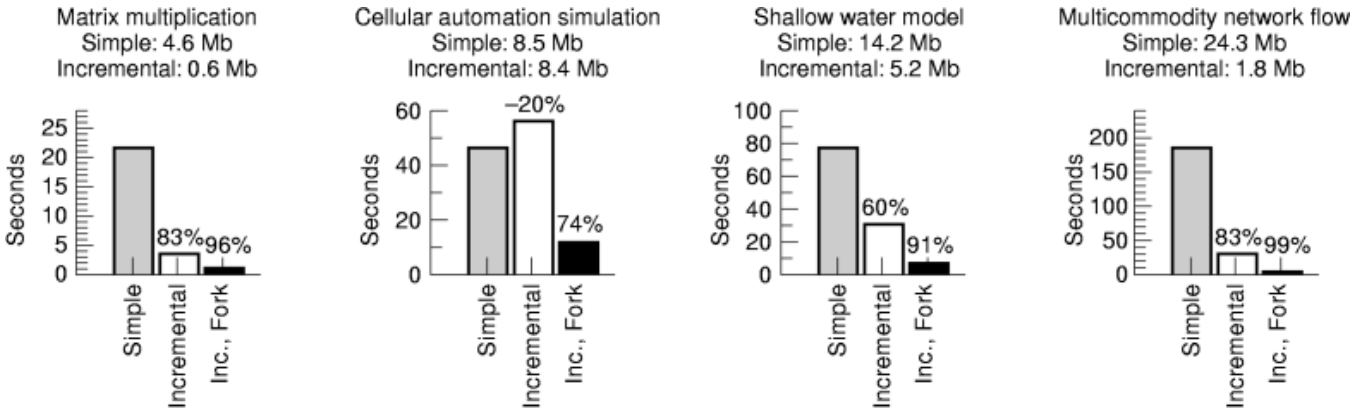

Fig. 9. Overhead of incremental checkpointing on a Sun Sparc-2 workstation. When an application modifies a fraction of its address space between checkpoints, incremental checkpointing can improve checkpointing performance.

A dead variable is one whose current value will not be used by the program. Either the program is done with that variable, and it will not be accessed again, or it will be overwritten before it is read. Dead variables do not need to be stored in checkpoint files, nor do they need to be restored in the event of a failure. Thus, a checkpointer may improve its performance by identifying dead variables at checkpoint time, and not storing them as part of a checkpoint.

A read-only variable is one whose value has not changed since the previous checkpoint (or since the beginning of the program). If previous checkpoints are not deleted, read-only variables do not need to be included in checkpoint files, because they may be restored from the previous checkpoints. Thus, a checkpointer may improve its performance by identifying read-only variables since the previous checkpoint, and not storing them as part of the next checkpoint.

As will be shown, memory exclusion has the potential to reduce the overhead of checkpointing. The challenge for the checkpointing system is to identify opportunities for memory exclusion transparently and efficiently. In the sections that follow, several techniques for memory exclusion are outlined and examples of the improvements in performance that they exhibit are shown.

Incremental Checkpointing. Incremental checkpointing is a technique for automating read-only memory exclusion using virtual memory primitives (21). With incremental checkpointing, following a checkpoint, all pages in memory are set to be read-only. When the program attempts to write a read-only page, an access violation occurs, and the checkpointer processes the resulting interrupt by storing the identity of the offending page in a list and resetting its protection to read-write. When it is time to take the next checkpoint, only pages that have caused access violations (those stored in the list) are checkpointed. The other pages have not been modified since the previous checkpoint, and are therefore composed solely of read-only variables.

Incremental checkpointing improves performance if the savings achieved by checkpointing fewer pages are greater than the penalty for setting the page protections and processing access violations. Unless almost all of memory is altered between checkpoints, this is usually the case. For example, Fig. 9 shows the results of incremental checkpointing on a transparent user-level checkpointer on a Sparc-2 workstation. Results of incremental checkpointing, and of combining incremental and forked checkpointing are presented. The average size of incremental checkpoints is reported as well.

In all the applications of Fig. 9 except one, incremental checkpointing greatly reduces the checkpoint size and overhead. In the cellular automaton program, each page in memory is updated between checkpoints, causing incremental checkpointing to increase the overhead because of the processing of access violations.

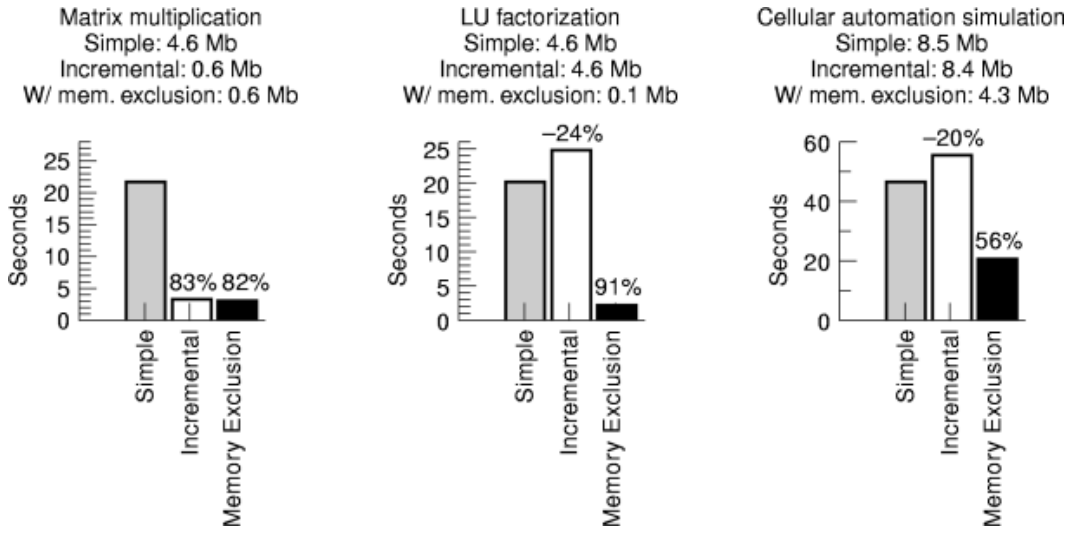

Fig. 10. Overhead of checkpointing with programmer-directed memory exclusion on a Sun Sparc-2 workstation. With a few hints from the programmer, memory exclusion can improve the performance of checkpointing significantly.

As Fig. 9 shows, incremental checkpointing can be combined with forked (or copy-on-write) checkpointing to reduce overhead even further.

Programmer-Directed Memory Exclusion. A simple, yet effective way to exclude memory is to allow the programmer to direct the checkpointer. The *libckpt* checkpointer (9) implements a subroutine called exclude_bytes() that allows the programmer to exclude variables explicitly from checkpoints because they are read-only or dead. To cancel the effects of exclude_bytes(), a second subroutine called include_bytes() is also supported. At checkpoint time, the checkpointer uses the programmer information to exclude memory from the checkpoint.

Programmer-directed memory exclusion can be combined with synchronous checkpointing to improve performance even further. With synchronous checkpointing, the programmer may insert checkpoint_here() procedure calls, which force the checkpointer to checkpoint at specific code locations. Synchronous checkpointing can be advantageous because there may be certain code locations where the amount of dead variables is maximized. For example, in the cellular automaton simulation program, there are code locations where almost half of memory is dead. With synchronous checkpointing, the programmer can force checkpointing to occur at these code locations, thereby halving the overhead of checkpointing.

Figure 10 shows three example applications where synchronous checkpointing combined with programmer-directed memory exclusion results in reduced checkpointing overhead on a Sparc-2 workstation. In two of the three examples, these are improvements that cannot be achieved by incremental checkpointing because they exploit dead variable exclusion. Although not shown in Fig. 10, forked/copy-on-write checkpointing may be combined with programmer-directed memory exclusion to lower overhead even more. For additional information about programmer directed memory exclusion, see Ref. 9.

One concern with programmer directed memory exclusion is that the programmer may err in identifying dead and read-only variables. For example, a “live” variable may be excluded, or a variable that has changed since the previous checkpoint may be marked as read-only. In such cases, the resulting checkpoints are incorrect, and are useless for recovery. One way to combat this problem is to use compiler assistance in identifying variables to exclude. While the compiler cannot identify all dead and read-only variables, it can use data flow analysis to identify many such variables. The details are beyond the scope of this article—Ref. 22

presents further details. Compiler assistance can also be useful for placing synchronous checkpointing calls in advantageous places (23).

Checkpoint Compression. A final optimization technique is checkpoint compression, which falls into the category of size reduction. A straightforward way to compress checkpoints is to use a standard compression algorithm such as *LZW* (Lempel–Ziv–Welch) (24). Compression only lowers the overhead of checkpointing if the extra processing time that it takes to perform the compression is smaller than the savings that result from writing a smaller file to disk. Analytically, this is when:

$$\frac{S}{C} + \frac{(1-f)S}{D} < \frac{S}{C}$$

where S is the size of the checkpoint, C is the compression speed, f is the compression factor defined as (uncompressed size − compressed size)/(uncompressed size) and D is the speed of disk writes. Factoring out S and solving for f, we see that compression is beneficial when:

$$f > \frac{D}{C}$$

For most uniprocessor systems, compression cannot improve the performance of checkpointing because $D/C > 1$. For example, in experiments on a Sun 3/50, Li and Fuchs reported values of $C = 0.034$ Mb/s and $D = 0.100$ Mb/s (23). Thus, no amount of compression can improve the performance of checkpointing.

On systems where multiple processors contend for disk storage, the value of C increases relative to D to the point where compression can be beneficial. For example, on a 32-processor iPSC/860, values of $C = 11.0$ Mb/s and $D = 2.16$ Mb/s were reported, meaning that compression is beneficial when $f > 0.193$.

Figure 11 shows the overhead of checkpointing on the iPSC/860 when compression is employed, compared to when it is not employed. In three of the four applications, the compression factor is high enough to lower the overhead of checkpointing, and indeed this is the case. In the fast Fourier transform, the entire memory space consists of essentially random floating point numbers, which are notoriously hard to compress. Full details on these experiments are in Ref. 16.

Other Performance Considerations

There are other metrics for checkpointing performance, such as checkpoint latency, recovery time, and space overhead. Checkpoint latency is defined to be the time it takes to commit a checkpoint from start to finish. Note that with no optimizations, checkpoint latency is equal to overhead, but when optimizations such as copy-on-write are employed, latency can be far greater than overhead. It has been shown that in fault-tolerant systems, checkpoint latency is minimally important compared to overhead. In other words, the performance of the system is affected far more by improvements in overhead than by improvements in latency (25).

In job-swapping applications, checkpoint latency is the most important metric. Because the application is terminated after the checkpoint is committed, there is no point in lowering failure-free overhead. For similar reasons, latency is more important than overhead for process migration as well.

In playback debugging applications, all checkpoints must be retained because any arbitrary previous state of the program may be desired. Thus, both space overhead and checkpoint overhead become valid concerns. Some checkpointing techniques based on executable rewriting have been developed that exhibit relatively high overheads (factor of two) so that checkpoint size can be minimized (5). Improving this further is an open area of research.

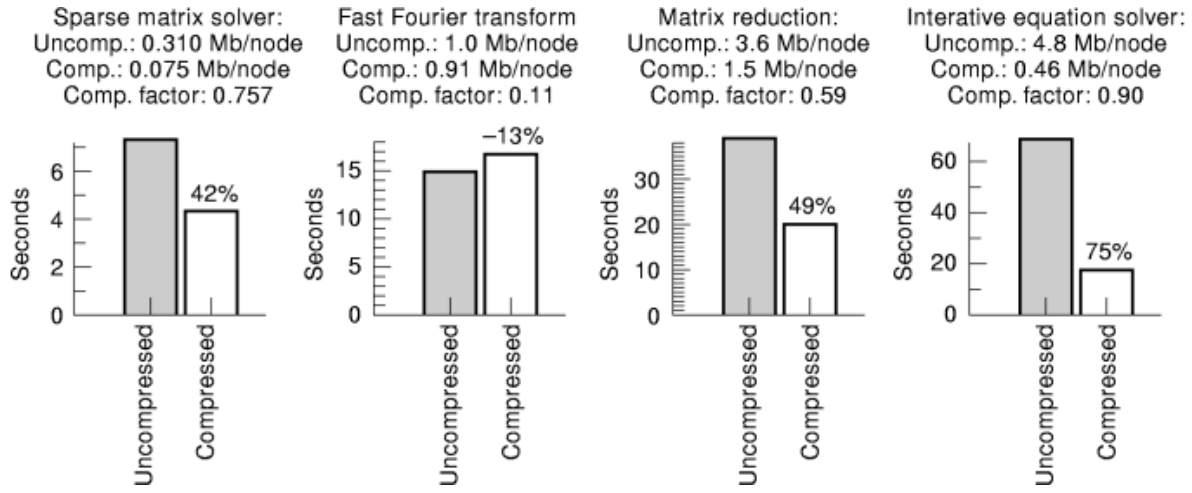

Fig. 11. Overhead of checkpointing with compression on the Intel iPSC/860. In order for compression to improve checkpointing performance, the CPU overhead of compressing the checkpoints must be lesser than the savings gained by writing smaller checkpoint files. On the iPSC/860, the stable storage bottleneck combined with the fact that the processors compress in parallel allowed compression to be beneficial.

Conclusion

Checkpointing is an extremely important functionality in program diagnostics. While there has been much research on checkpointing combined with many experimental implementations, popular operating systems still do not provide support for checkpointing. This has led to a proliferation of user-level checkpointing packages and application specific checkpointing implementations. This article has discussed the basic issues in checkpointing, including details on implementation, checkpoint consistency, and performance. The author is hopeful that future operating systems and applications will be designed with checkpointing as a central feature, so that users may benefit from its many functionalities.

BIBLIOGRAPHY

1. B. A. Kingsbury J. T. Kline Job and process recovery in a UNIX-based operating system, *Conf. Proc., Usenix Winter 1989 Tech. Conf.*, San Diego, CA, pp. 355–364, 1989.
2. C. R. Landau The checkpoint mechanism in keykos, *Proc. 2nd Inter.Workshop Object Orientat. Oper. Syst.*, pp. 86–91, 1992.
3. M. Russinovich Z. Segall Fault-tolerance for off-the-shelf applications and hardware, *25th Inter. Symp. Fault-Tolerant Comput.*, Pasadena, CA, pp. 67–71, 1995.
4. J. K. Ousterhout *et al.* The sprite network operating system, *IEEE Comput.*, **21** (2): 23–36, 1988.
5. R. H. B. Netzer M. H. Weaver Optimal tracing and incremental reexecution for debugging long-running programs, *ACM SIGPLAN '94 Conf. Program. Lang. Des. Implement.*, Orlando, FL, pp. 313–325, 1994.
6. J. Long W. K. Fuchs J. A. Abraham Implementing forward recovery using checkpointing in distributed systems, *2nd IFIP Working Conf. Dependable Comput. Crit. Appl.*, pp. 20–27, 1991.
7. M. J. Litzkow M. Livny Making workstations a friendly environment for batch jobs, *3rd Workshop Workst. Oper. Syst.*, 1992.
8. J. S. Plank *Efficient checkpointing on MIMD architectures*, Ph.D. thesis, Princeton University, Princeton, NJ, 1993.

9. J. S. Plank *et al. Libckpt:* Transparent checkpointing under unix, *Conf. Proc., Usenix Winter 1995 Tech. Conf.*, pp. 213–223, 1995.
10. Y.-M. Wang *et al.* Checkpointing and its applications, *25th Int. Symp. Fault-Tolerant Comput.*, Pasadena, CA, pp. 22–31, 1995.
11. Y. M. Wang Y. Huang W. K. Fuchs Progressive retry for software error recovery in distributed system, *23rd Int. Symp. Fault-Tolerant Comput.*, pp. 138–144, 1993.
12. D. R. Jefferson Virtual time, *ACM Trans. Program. Lang. Syst.*,7 (3): 404–425, 1985.
13. R. Fujimoto Parallel discrete event simulation, *Commun. ACM*,33 (10): 1990.
14. E. N. Elnozahy D. B. Johnson Y. M. Wang *A Survey of Rollback-Recovery Protocols in Message-Passing System, Techn. Rep. CMU-CS-96-181*, Pittsburgh, PA: Carnegie Mellon University, 1996.
15. K. M. Chandy L. Lamport Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Syst.*,3 (1): 63–75, 1985.
16. J. S. Plank K. Li Ickp—a consistent checkpointer for multicomputers, *IEEE Parallel Distrib. Technol.*,2 (2): 62–67, 1994.
17. E. N. Elnozahy W. Zwaenepoel On the use and implementation of message logging, *24th Int. Symp. Fault-Tolerant Comput.*, Austin, TX, pp. 298–307, 1994.
18. J. S. Young A first order approximation to the optimum checkpoint interval, *Commun. ACM*,17 (9): 530–531, 1974.
19. P. Jalote *Fault Tolerance in Distributed Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1994.
20. E. N. Elnozahy D. B. Johnson W. Zwaenepoel The performance of consistent checkpointing, *11th Symp. Reliable Distrib. Syst.*, pp. 39–47, 1992.
21. S. I. Feldman C. B. Brown Igor: A system for program debugging via reversible execution, *ACM SIGPLAN Notices, Workshop Parallel Distrib. Debugging*,24 (1): 112–123, 1989.
22. J. S. Plank M. Beck G. Kingsley Compiler-assisted memory exclusion for fast checkpointing, *IEEE Tech. Comm. Oper. Syst. Appl. Environ.*,7 (4): 10–14, 1995.
23. C.-C. J. Li W. K. Fuchs CATCH—Compiler-assisted techniques for checkpointing, *20th Inter. Symp. Fault Tolerant Comput.*, pp. 74–81, 1990.
24. T. A. Welch A technique for high-performance data compression, *IEEE Comput.*, **17**: 8–19, 1984.
25. N. H. Vaidya On checkpoint latency, *Pac. Rim Inter. Symp. Fault-Tolerant Syst.*, Newport Beach, RI, 1995.
26. Y. Huang C. Kintala Y.-M. Wang Software tools and libraries for fault tolerance, *IEEE Tech. Comm. Oper. Syst. Appl. Environ.*,7 (4): 5–9, 1995.
27. T. Tannenbaum M. Litzkow The Condor distributed processing system, *Dr. Dobb's J.*, **227**: 40–48, 1995.
28. E. N. Elnozahy W. Zwaenepoel Manetho: Transparent rollback-recovery with low overhead, limited roll-back and fast output commit, *IEEE Trans. Comput. Spec. Issue Fault-Tolerant Comput.*,41 (5): 1992.
29. J. Casas *et al.* MPVM: A migration transparent version of PVM, *Comput. Syst.*, **8** (2): 171–216, 1995.
30. G. Stellner CoCheck: Checkpointing and process migration for MPI, In *10th Inter. Parallel Process. Symp.*, 1996.
31. M. Costa *et al.* Lightweight logging for lazy release consistent distributed shared memory, *2nd Symp. Oper. Syst. Des. Implement.*, 1996.
32. Y. Chen J. S. Plank K. Li CLIP: A checkpointing tool for message-passing parallel programs, *SC97: High Perform. Network. Comput.*, San Jose, CA, 1997.
33. A. Beguelin E. Seligman P. Stephan Application level fault tolerance in heterogeneous networks of workstations. *J. Parallel Distrib. Comput.* (to be published).
34. L. M. Silva *et al.* Portable checkpointing and recovery, *Proc. HPDC-4, High-Perform. Distrib. Comput.*, Washington, DC, pp. 188–195, 1995.
35. A. Baratloo P. Dasgupta Z. M. Kedem CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms, *4th IEEE Inter. Symp. High Perform. Distrib. Comput.*, 1995.
36. D. Cummings L. Alkalaj Checkpoint/rollback in a distributed system using coarse-grained dataflow, *24th Inter. Symp. Fault-Tolerant Comput.*, Austin, TX, pp. 424–433, 1994.

JAMES S. PLANK
University of Tennessee