Diskless Checkpointing

James S. Plank †

Kai Li §

Michael Puening ‡

† Department of Computer Science
University of Tennessee
Knoxville, TN 37996
plank@cs.utk.edu

§ Department of Computer Science
Princeton University
Princeton, NJ 08544
li@cs.princeton.edu

‡ mpuening@cardinalsolutions.com

December 12, 1997

Technical Report UT-CS-97-380
University of Tennessee

Available via ftp to cs.utk.edu in pub/plank/papers/CS-97-380.ps.Z
Or on the web at http://www.cs.utk.edu/~plank/plank/papers/CS-97-380.html

Submitted for publication. See the web page for futher information.

Diskless Checkpointing

James S. Plank* Kai Li Michael A. Puening

December 18, 1997

This paper has been submitted for publication. Please see http://www.cs.utk.edu/~plank/plank/papers/CS-97-380.html *for up to date information concerning the publication status.*

The precursor to this work (where diskless checkpointing was first described) was presented at FTCS-24 [27].

Abstract

Diskless Checkpointing is a technique for checkpointing the state of a long-running computation on a distributed system without relying on stable storage. As such, it eliminates the performance bottleneck of traditional checkpointing on distributed systems.

In this paper, we motivate diskless checkpointing and present the basic diskless checkpointing scheme along with several variants for improved performance. The performance of the basic scheme and its variants is evaluated on a high-performance network of workstations and compared to traditional disk-based checkpointing. We conclude that diskless checkpointing is a desirable alternative to disk-based checkpointing that can improve the performance of distributed applications in the face of failures.

1 Introduction

Checkpointing is an important topic in fault-tolerant computing as the basis for *rollback recovery.* Suppose a user is executing a long-running computation and for some reason (hardware or software), the machine running the computation fails. In the absence of checkpointing, when the machine becomes functional, the user must start the program over, thus wasting all previous computation. Had the user stored periodic checkpoints of the program's state to stable storage, then he or she could instead restart the program from the most recent checkpoint. This is called *rolling back* to a stored checkpoint. For long-running computations, checkpointing allows users to limit the amount of lost computation in the event of a failure (or failures).

There have been many programming environments intended for users with long-running computations that rely on checkpointing for fault-tolerance. For example, Condor [34], **libckpt** [25] and others [16, 30, 37] provide

*`plank@cs.utk.edu`. This material is based upon work supported by the National Science Foundation under grants CCR-9409496, MIP-9420653 and CDA-9529459, by the ORAU Junior Faculty Enhancement Award, and by DARPA under grant N00014-95-1-1144 and contract DABT63-94-C-0049.

transparent checkpointing for uniprocessor programs, and checkpointers such as MIST [4], CoCheck [33] and others [2, 10, 18, 28, 32] provide checkpointing in parallel computing environments.

All of the above systems store their checkpoints on stable storage (i.e. disk), since stable storage typically survives processor failures. However, since checkpoints can be large (up to hundreds of megabytes per processor), the act of storing them to disk becomes the main component that contributes to the *overhead*, or performance degradation, due to checkpointing. This is more marked in parallel and distributed systems where the number of processors often vastly outnumbers the number of disks.

Several techniques have been devised and implemented to minimize this source of overhead, including incremental checkpointing [11, 38], checkpoint buffering with copy-on-write [9, 21], compression [20, 28] and memory exclusion [25]. However with all of these techniques, the performance of the stable storage medium is still the underlying cause of overhead.

In this paper, we present *diskless checkpointing*. The goal of diskless checkpointing is to remove stable storage from checkpointing in parallel and distributed systems, and replace it with memory and processor redundancy. By eliminating stable storage, diskless checkpointing removes the main source of overhead in checkpointing. However, this does not come for free. The failure coverage of diskless checkpointing is less than checkpointing to stable storage, since none of the components in a diskless checkpointing system can survive a wholesale failure. Moreover, there is memory, processor and network overhead introduced by diskless checkpointing that is absent in standard disk-based schemes.

The purpose of this paper is twofold. We first present basic schemes for diskless checkpointing and then performance optimizations to the basic schemes. Second, we assess the performance of diskless checkpointing on a network of Sparc-5 workstations as compared to standard disk-based checkpointing. As anticipated, diskless checkpointing induces less overhead on applications than disk-based checkpointing, enabling the user to checkpoint more frequently without a performance penalty. This lowers the application's expected running time in the presence of failures.

Diskless checkpointing tolerates single processor failures, and in some cases multiple processor failures. However, it does not tolerate wholesale failures (such as a power outage that knocks out all machines). Thus, an optimized fault-tolerant scheme would be a *two-level* scheme, as advocated by Vaidya [35], where diskless checkpoints are taken frequently, and standard, disk-based checkpoints are taken at a much larger interval. In this way, the more frequent case of one or two processors failing is handled swiftly, with low overhead, while the rarer case of a wholesale failure is handled as well, albeit with higher overhead and a longer rollback penalty.

2 Overview of Diskless Checkpointing

Diskless checkpointing is based on *coordinated checkpointing*. With coordinated checkpointing, a collection of processors with disjoint memories coordinates to take a checkpoint of the global system state. This is called a "coordinated checkpoint". A coordinated checkpoint consists of checkpoints of each processor in the system plus a log of messages in transit at the time of checkpointing. Coordinated checkpointing is a well-studied topic in fault-tolerance. For a thorough discussion of coordinated checkpointing, the reader is directed to the survey paper by Elnozahy, Johnson and Wang [8].

With diskless checkpointing we assume that there is no message log to be stored (for example, the "Sync-and-stop" algorithm for coordinated checkpointing ensures that there is no message log [28]), or that the message log is contained within the checkpoints of individual processors. This reduces the problem of taking a coordinated checkpoint to saving the individual checkpoints of each processor in the system.

Diskless checkpointing is composed of two parts – (1) checkpointing the state of each application processor *in memory*, and (2) encoding these in-memory checkpoints and storing the encodings in *checkpointing processors*. When a failure occurs, the system is recovered in the following manner. First, the non-failed application processors roll themselves back to their stored checkpoints in memory. Next, *replacement processors* are chosen to take the place of the failed processors. Finally, the replacement processors use the checkpointed states of the non-failed application processors plus the encodings in the checkpointing processors to *calculate* the checkpoints of the failed processors. Once these checkpoints are calculated, the replacement processors roll back, and the application continues from the checkpoint. Note that either spare processors or some of the checkpointing processors may be used as replacement processors. If checkpointing processors are used, then the system will continue with fewer (or no) checkpointing processors, thus reducing the fault-tolerance. However, when more processors become available, they may be employed as additional checkpointing processors.

2.1 Exact Problem Specification

The user is executing a long-running application on a parallel or distributed computing environment composed of processors with disjoint memories that communicate by message-passing. The application executes on exactly n processors. With diskless checkpointing, an extra m processors are added to the system, and the $n+m$ processors cooperate to take diskless checkpoints. As long as the number of processors in the system is at least n, and as long as failures occur within certain constraints, the application may proceed efficiently.

As stated above, diskless checkpointing may be broken into two parts: application processors checkpointing themselves, and checkpoint processors encoding the application processors' checkpoints. Each is explained below, followed by issues involved in gluing the two parts together.

3 Application Processors Checkpointing Themselves

Here the goal is for an application processor to checkpoint its state in such a way that if a rollback is called for, due to the failure of another processor, the processor can roll back to its most recent checkpoint. In standard disk-based systems, a processor checkpoints itself by saving the contents of its address space to disk. Typically this involves saving all values in the stack, heap, global variables and registers as in Figure 1(a). If the processor must roll back, it overwrites the current contents of its address space with the stored checkpoint. As a last step, it restores the registers, which restarts the computation from the checkpoint, thereby completing the rollback. For more detail on general process checkpointing and recovery, see the papers on Condor [34] and **libckpt** [25].

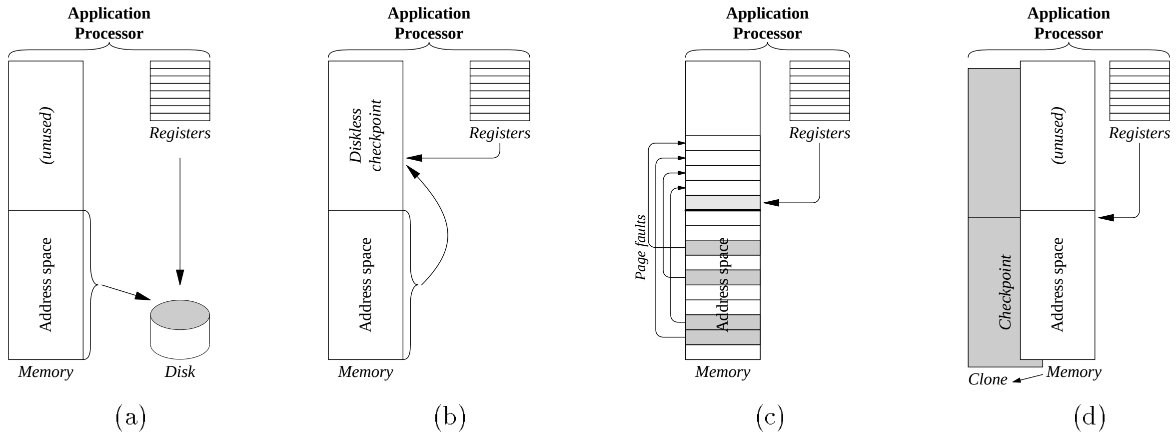

Figure 1: (a) Checkpointing to disk, (b) simple diskless checkpointing, (c) incremental diskless checkpointing, (d) forked diskless checkpointing

With diskless checkpointing, the processor saves its state in memory, rather than on disk. In its simplest form, diskless checkpointing requires an in-memory copy of the address space and registers, as in Figure 1(b). If a rollback is required, the contents of the address space and registers are restored from the in-memory checkpoint. Note that this checkpoint will *not* tolerate the failure of the application processor itself; it simply enables the processor to roll back to the most recent checkpoint if *another* processor fails.

One drawback of simple diskless checkpointing is memory usage. A complete copy of the application must be retained in the memory of each application processor. A solution to this problem is to use *incremental* checkpointing [11, 38], as in Figure 1(c). To take a checkpoint, an application processor sets the virtual memory protection bits of all pages in its address space to be *read-only* [1]. When the application attempts to write a page, an access violation (page fault) occurs. The checkpointing system then makes a copy of the faulting page, and resets the page's protection to *read-write*. Thus, a processor's checkpoint consists of the *read-only* pages in its address space plus the stored copies of all the *read-write* pages. To roll back to a checkpoint, the processor simply copies (or maps) the checkpointed copies of all its *read-write* pages back to the application's address space. As long as the application does not overwrite all of its pages between checkpoints, incremental checkpointing improves both the performance and memory utilization of checkpointing.

The last useful checkpointing method is *forked* (or *copy-on-write*) checkpointing [9, 21, 25]. To checkpoint, the application clones itself (with, for example, the `fork()` system call in Unix) as depicted in Figure 1(d). This clone is the diskless checkpoint. To roll back, the application overwrites its state with the clone's, or if possible, the clone merely assumes the role of the application. Forked checkpointing is very similar to incremental checkpointing because most operating systems implement process cloning with *copy-on-write*. This means that the process and its clone will share pages until one of the processes alters the page. Thus, it works in the same manner as incremental checkpointing, except the identification of modified pages and the page copying are all performed in the operating system. This results in less CPU activity switching back and forth from system to user mode. Moreover, forked checkpointing does not require that the user have access to virtual memory protection facilities, which are not available in all operating systems.

4 Encoding the checkpoints

The goal of this part is for extra checkpoint processors to store enough information that the checkpoints of failed processors may be reconstructed. Specifically, there are m checkpoint processors. These processors encode the checkpoints of the application processors in such a way that when application processors fail, their checkpoints may be recalculated from the checkpoints of the non-failed processors plus the encodings in the checkpoint processors.

4.1 Parity (Raid Level 5)

The simplest checkpoint encoding is parity (Figure 2(a)). Here there is one checkpoint processor (i.e. $m = 1$) that encodes the bitwise parity of each of the application's checkpoints. In other words, let byte b_i^j represent the j-th byte of application processor i. Then the j-th byte of the checkpoint processor will be:

$$b_{ckp}^j = b_1^j \oplus b_2^j \oplus \ldots \oplus b_n^j$$

If any application processor fails, the state of the system may be recovered as follows. First, a replacement processor is selected to take the place of the failed application processor. This could be the checkpoint processor, a spare processor that had previously been unused, or the failed processor itself if the failure was transient. The replacement processor calculates the checkpoint of the failed processor by taking the parity of the checkpoints of the non-failed processors and the encoding in the checkpoint processor. In other words, suppose processor i is the failed processor. Then its checkpoint may be reconstructed as:

$$b_i^j = b_1^j \oplus \ldots \oplus b_{i-1}^j \oplus b_{i+1}^j \oplus \ldots \oplus b_n^j \oplus b_{ckp}^j$$

Note that this is the same recovery scheme as Raid Level 5 in disk array technology [5]. When the replacement processor has calculated the checkpoint of the failed processor, then all application processors roll back to the previous checkpoint, and the computation proceeds from that point.

Besides parity, there are several other schemes than can be used to encode the checkpoints. They vary in the number of checkpoint processors, the efficiency of encoding, and the amount of failure coverage. They are detailed below.

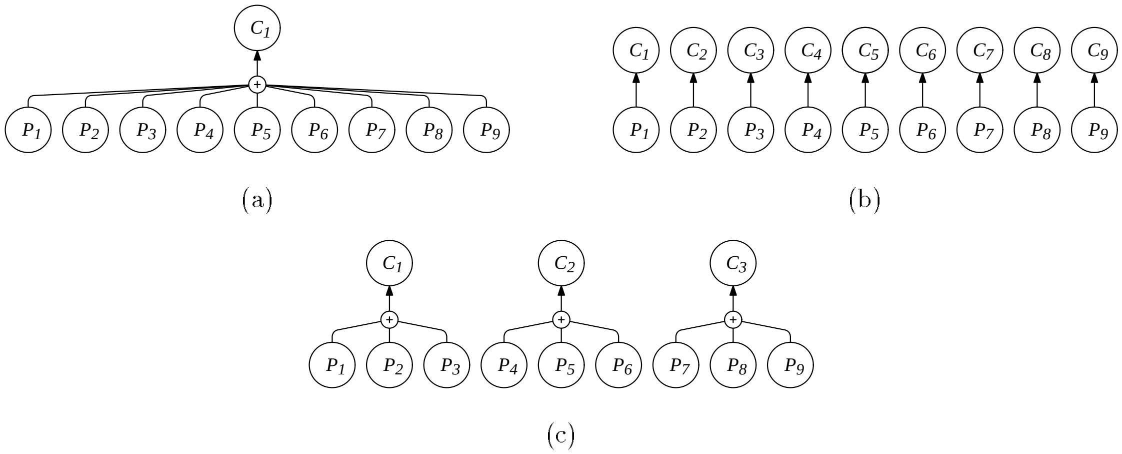

Figure 2: Encoding the checkpoints: (a) Raid Level 5, (b) Mirroring, (c) One-dimensional parity

4.2 Mirroring

Checkpoint mirroring (Figure 2(b)) is another simple encoding scheme. With mirroring, there are $m = n$ checkpoint processors, and the i-th checkpoint processor simply stores the checkpoint of the i-th application processor. Thus, up to n processor failures may be tolerated, although the failure of both an application processor and its checkpoint processor cannot be tolerated. Checkpoint mirroring should have a very low checkpointing overhead because no encoding calculations (such as parity) need to be made.

4.3 1-dimensional parity

With one-dimensional parity (Figure 2(c)) there are $1 \leq m \leq n$ checkpoint processors. The application processors are partitioned into m groups $g_1, \ldots, g_m$ of roughly equal size. Checkpoint processor i then calculates the parity of the checkpoints in group i. This increases the failure coverage, because now one processor failure per group may be tolerated. Moreover, the calculation of the checkpoint encoding should be more efficient because there is no longer a single bottleneck (the checkpoint processor). Note that 1-dimensional parity reduces to Raid Level 5 when $m = 1$, and to mirroring when $m = n$.

4.4 2-dimensional parity

Two-dimensional parity (Figure 3(d)) is an extension of one-dimensional parity. With two-dimensional parity, the application processors are arranged logically in a two-dimensional grid, and there is a checkpoint processor for each row and column of the grid. Each checkpoint processor calculates the parity of the application processors in its row or column. Two-dimensional parity requires $m \geq 2\sqrt{n}$ checkpoint processors, and can tolerate the failure of any one processor in each row and column. This means that *any* two-processor failures may be tolerated.

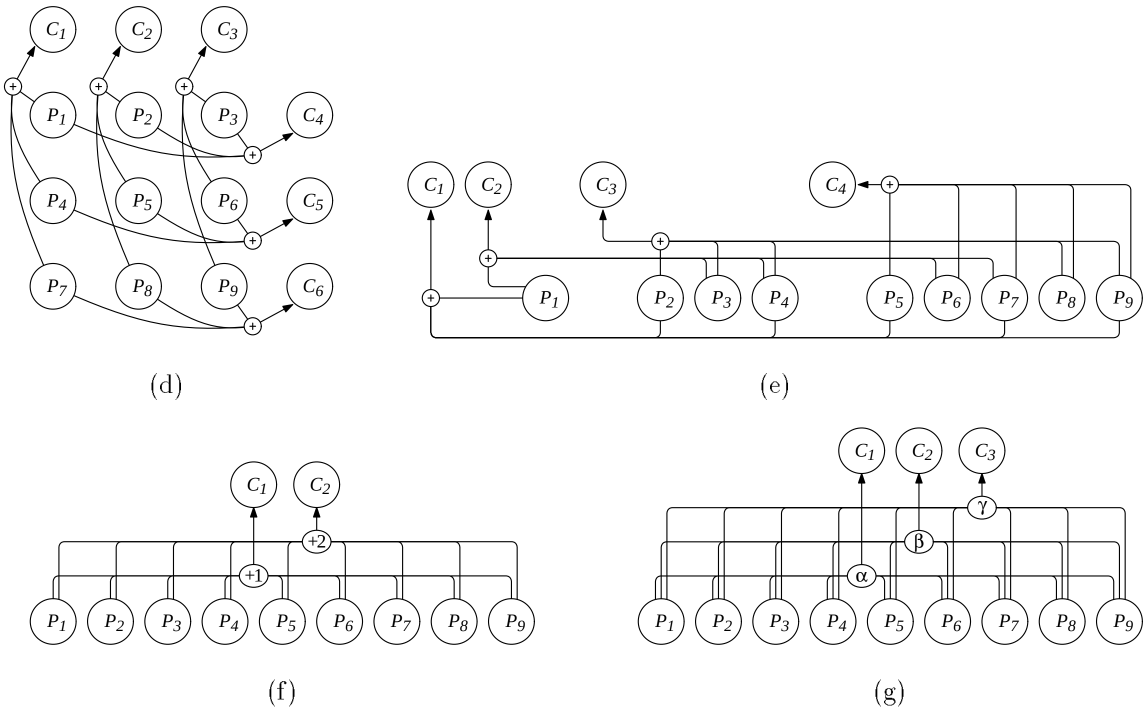

Figure 3: Encoding the checkpoints: (d) Two-dimensional parity, (e) Hamming coding, (f) EVENODD coding, (g) Reed-Solomon coding

4.5 Other parity-based codes

The well-known Hamming codes (Figure 3(e)) may be used to tolerate any two-processor failures with the addition of roughly $\log n$ processors [13]. Each checkpoint processor calculates the parity of a subset of the application processors. EVENODD coding (Figure 3(f)) is a technique where $m = 2$ checkpoint processors are employed and all two-processor failures may be tolerated [3]. The encoding is based on parity calculations, but is a little more complex than the above schemes.

4.6 Reed-Solomon coding

The most general purpose encoding technique is Reed-Solomon coding [24] (Figure 3(g)). Here m checkpointing processors use Galois Field arithmetic to encode the checkpoints in such a way that *any* m failures may be tolerated. Since the encoding is more complex than parity, the CPU overhead of Reed-Solomon coding is greater than the other methods, but it achieves maximal failure coverage per checkpoint processor.

5 Gluing the two parts together

Sections 3 and 4 have discussed how application processors store checkpoints internally, and how the checkpoint processors encode information. The final component of diskless checkpointing is coordinating the application and checkpointing processors in an efficient and correct way. This section discusses the relevant details in the coordination of the two sets of processors. We focus primarily on Raid Level 5 encodings, and then discuss the differences that the other encodings entail.

5.1 Tolerating failures when checkpointing

As with all checkpointing systems, diskless checkpointing systems must take care to remain fault-tolerant even if there is a failure while checkpointing or recovery is underway. This is done by making sure that each coordinated checkpoint remains valid until the next coordinated checkpoint has been completed. The checkpointing processors control this process. When all the checkpointing processors have completed calculating their encodings for the current checkpoint, then they may discard their previous encodings, and then notify the application processors that they may discard their previous checkpoints.

Upon recovery, if the checkpointing processors all have valid encodings for the most recent checkpoint, then these are used for recovery, along with the most recent checkpoints in the non-failed application processors. If any checkpointing processor does not have a valid encoding for the most recent checkpoint, then the previous encoding must be used along with the previous checkpoints in the non-failed application processors.

This protocol ensures that there is always a valid coordinated checkpoint of the system in memory. If all checkpoint processors have their encodings for coordinated checkpoint i, then all application processors will have their checkpoints for coordinated checkpoint i. If any checkpoint processor has an incomplete encoding for checkpoint i, then all checkpoint processors will still contain their encodings for coordinated checkpoint $i-1$. Moreover, all application processors will have their checkpoints for coordinated checkpoint $i-1$. Thus, the whole system may recover to coordinated checkpoint $i-1$.

If a failure is detected during recovery, then the remaining processors simply initiate the recovery procedure anew.

5.2 Space demands

A ramification of the preceding protocol is that at the moment when the checkpoint processors finish storing their encodings, all processors contain two checkpoints in memory: the current checkpoint and the previous checkpoint. Thus, the memory usage of diskless checkpointing is a serious issue.

Suppose the size of an application processor's address space is M bytes. Then simple diskless checkpointing consumes an extra M bytes of memory to hold a checkpoint. To ensure that only M bytes of extra memory are consumed at all times, the application must be frozen during checkpointing. Then the application's address

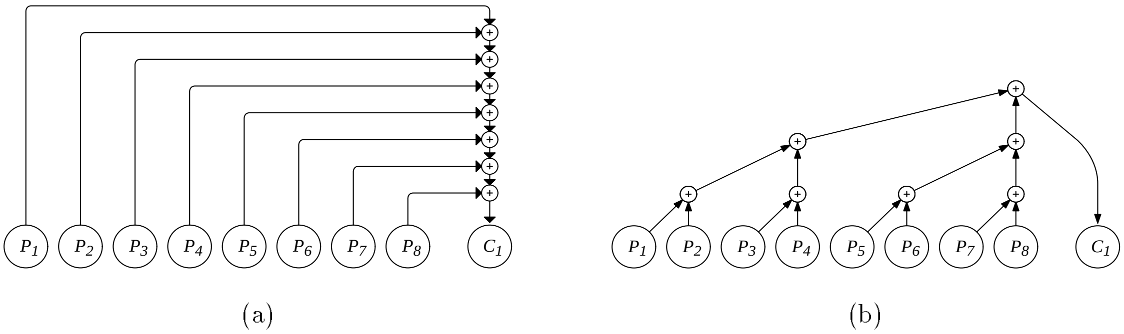

Figure 4: Calculating the encoding: (a) DIRECT, (b) FAN-IN

space may be used (without being copied) to calculate the checkpoint encodings. When the encodings have been calculated, the application's address space may be copied over its previous checkpoint, which is now expendable. Then the application is unfrozen.

With incremental checkpointing, checkpointed copies of pages are made when page faults are caught. At checkpoint time, the processors calculate the encodings, then discard the checkpointed copies of pages and set the protection of all application pages to *read-only*. Thus, if the incremental checkpoint size is I, then only I extra bytes of memory are necessary. In the worst case, all pages are modified between checkpoints, and I equals M.

With forked checkpointing, each checkpoint is a separate process. When the checkpoint processors complete their encodings, there are three processes contained by each application processor: the application itself, its most recent checkpoint, and the previous checkpoint. Since process cloning uses the copy-on-write optimization, each checkpoint process only consumes an extra I bytes of memory. Therefore, forked checkpointing requires an extra $2I$ bytes of memory during checkpointing, and I bytes at all other times. In the worst case, this is $2M$ during checkpointing, and M at other times.

Finally, disk-based checkpointing using the fork optimization requires I' bytes of memory, where I' consists of all pages that are modified while checkpointing is taking place. I' should be less than I, though if the latency of checkpointing is large compared to the checkpointing interval, I' may be close to I.

5.3 Sending and calculating the encoding

With Raid Level 5 encoding, there is one checkpoint processor C_1, and n application processors $P_1, \ldots, P_n$. C_1 stores the bitwise parity of the checkpoints of each application processor. The simplest way to calculate the parity is to employ the DIRECT method: each application processor simply sends its checkpoint to C_1. Initially, C_1 clears a portion of its memory, which we call e_1, to store the checkpoint encoding. Upon receiving ckp_i from P_i, it sets e_1 to $(e_1 \oplus ckp_i)$. This is shown in Figure 4(a). In Figure 4, the $\oplus$ signs are shown directly above the processors that perform the bitwise exclusive or. Arrows from one processor to another represent one processor sending its checkpoint to another.

There are two problems with the DIRECT method. First, C_1 can become a message-receiving bottleneck, since it is the destination of all checkpoint messages. Second, C_1 does all of the parity calculations. Both problems may be alleviated with the FAN-IN algorithm. Here, the application processors perform the parity calculation in $\log n$ steps and send the final result to C_1, which stores the result in its memory. This is shown in Figure 4(b).

For other encodings besides Raid Level 5, these two methods may be extended. In the DIRECT method, each processor sends its checkpoint in a multicast message to the proper checkpointing processors. If necessary (e.g., for Reed-Solomon coding), the checkpointing processors modify the checkpoints, and then exclusive-or them into their checkpoints. In the FAN-IN method, there is one FAN-IN performed for each checkpointing processor. This may entail the cooperation of all application processors (e.g., in Reed-Solomon coding), or a subset of the application processors (e.g., in one-dimensional parity). If a checkpoint must be modified for the encoding, it is done at application processor P_i before the fan-in starts.

For most networks, the FAN-IN algorithm will be preferable to the DIRECT because it eliminates bottlenecks and distributes the parity calculations. However, if the network supports multicast, the encodings involving multiple checkpointing processors may profit from the DIRECT method.

5.4 Breaking the checkpoint into chunks

The preceding description implies that whole checkpoints are sent from processor to processor. Since checkpoints may be large, it often makes more efficient use of memory to break the checkpoint into *chunks* of a fixed size. For example, in the FAN-IN algorithm, only two extra chunks of memory are needed to receive an incoming chunk from another processor, make the parity calculation, and then send off the result. The chunks should be small enough that they do not consume too much memory, but large enough that the overhead in sending chunks is not dominated by message-sending start-up.

5.5 Sending *diffs*

If the application processors use incremental checkpointing, then they can avoid overhead by sending only pages that have been modified since the previous checkpoint. However, this can cause problems in creating the checkpoint encoding. Specifically, if the encoding is to be created anew at each checkpoint, it needs to have all checkpointing data from all processors. The solution to this is to use *diffs*.

Assume that the DIRECT encoding method is being employed. The checkpoint processor first copies its previous checkpoint to its current checkpoint. Then each application processor does the following. For each modified page $page_k$ in its address space, it calculates $diff_k$, which is the bitwise exclusive-or of the current copy of the page and the copy of the page in the previous checkpoint (which of course is available to the application processor). It then sends $diff_k$ to the checkpoint processor, which `XOR`'s it into its checkpoint. This has the effect of subtracting out the old copy of the page and adding in the new copy. In this way, unmodified pages need not be sent to the checkpointing processor.

One may use *diffs* with the FAN-IN algorithm as well, stipulating that if a processor does not modify a page during the checkpoint interval, then it does not need to send that page or `XOR` it with other pages when performing the fan-in.

5.6 Compressing Diffs

By sending *diffs* rather than actual bytes of the checkpoint, an interesting opportunity for compression arises. Suppose that an application modifies just a few bytes on a page. Then the *diff* of that page and its previously checkpointed copy will be composed of mostly zeros, which can be easily compressed using either run-length encoding or an algorithm that sends tagged bytes rather than whole pages. Such compression trades off use of more CPU for a reduced load on the network.

Compression combines naturally with incremental checkpointing, where modified pages are compressed before being sent. It may also be used with simple and forked checkpointing by converting the entire checkpoint into a *diff* and compressing it before sending it along. This has the effect of emulating incremental checkpointing, because regions of memory that have not been modified get compressed to nothing.

6 Implementation and Experiment

In order to assess the performance of diskless checkpointing as compared to standard disk-based checkpointing on networks of workstations, we implemented a small transparent checkpointing system on a network of 24 Sun Sparc5 workstations at the University of Tennessee. Each workstation has 96 Mbytes of physical memory and runs SunOS version 4.1.3. The workstations are connected to each other by a fast, switched Ethernet which can be isolated for performance testing. The measured peak bandwidth between any two processors is roughly 5 megabytes per second. The workstations have very little accessible local disk storage: 38 megabytes per machine. However, the machines are connected via regular Ethernet to the department's file servers using Sun NFS. These disks have a bandwidth of 1.7 megabytes per second, but the performance of NFS on the Ethernet is far worse. With NFS, remote file writes achieve a bandwidth of 0.13 megabytes per second. The page size of each machine is 4096 bytes, and access to the page tables is controlled by the `mprotect()` system call.

Our checkpointer runs on top of PVM [12] and works like many PVM checkpointers [4, 33]. Applications do not need to be recompiled, but their object modules must be relinked with our checkpointing/modified PVM library. When the applications are started, the checkpointing code gets control and reads startup information from a control file. This information includes the checkpoint interval, which checkpointing optimizations to use, plus where checkpoints should be stored (to disk or to checkpointing processors).

The application then starts, and one of the application processors is interrupted when the checkpointing interval has expired. This processor coordinates with the other application processors using the "Sync-and-stop" synchronization algorithm, and once consistency has been determined, the processors checkpoint.

Abbreviation	Description
BASE	No checkpointing
DISK-FORK	Checkpointing to disk using `fork()`
SIMP	Simple diskless checkpointing
INC	Incremental diskless checkpointing
FORK	Forked diskless checkpointing
INC-FORK	Incremental, forked diskless checkpointing
C-SIMP	Simple diskless checkpointing with compression
C-INC	Incremental diskless checkpointing with compression
C-FORK	Forked diskless checkpointing with compression
C-INC-FORK	Incremental, forked diskless checkpointing with compression

Table 1: Checkpointing variants implemented in our experiments

PVM includes some basic forms of failure detection. Specifically, if a processor in the current PVM session fails, the rest of the processors eventually notice the failure and remove the failed processor from the PVM session. PVM allows the user to be notified of such events. Our checkpointer uses this facility to recognize processor failures. When such a failure occurs, then if there is a spare processor in the PVM session, it is selected to replace the failed processor. If there is no spare processor, and diskless checkpointing is being employed, then a checkpoint processor is chosen to be the replacement processor. Recovery proceeds automatically, either from the disk-based or diskless checkpoint.

It is important to note that our checkpointer does not require the programmer to modify his or her code to enable checkpointing. A simple relinking is all that is necessary.

The gamut of checkpointing variants is enumerated in Table 1. This includes standard disk-based checkpointing using the `fork()` optimization. We do not test incremental, disk-based checkpointing because it does not improve the performance of checkpointing in any of our tests.[1].

For diskless checkpointing, we implement Raid Level 5 encoding using the FAN-IN algorithm. Checkpoint encodings are created in chunks of 4096 bytes (conveniently, also the page size). The choice of algorithm has some ramifications on how certain optimizations work. For example, when performing incremental checkpointing, the encoding is created chunk-by-chunk, but if a processor has not modified the corresponding page, then an empty message is sent as part of the fan-in instead of the page.

When using *diff*-based compression, pages are compressed using a bitmap-based compression algorithm [29]. Compression is performed by the sending processor before sending, and then uncompressed by the receiving

[1]This is not to say that incremental, disk-based checkpointing is not often a useful optimizations. It simply does not help in any of our tests.

Application	Running Time		Checkpoint Size per node
	(sec)	(h:mm:ss)	(Mbytes)
NBODY	5722	1:35:22	3.7
MAT	6602	1:50:02	15.5
PSTSWM	5610	1:33:30	24.4
CELL	6351	1:45:51	41.4
PCG	5873	1:37:53	66.6

Table 2: Basic parameters of the testing applications

processor, which merges the page with its own, and compresses the result before sending it along. When the final compressed chunk reaches the checkpointing processor, it uncompresses the chunk and merges it with the previous checkpoint encoding, which is then stored as the next encoding.

7 Applications

We used five applications to test the performance of checkpointing. These applications are all CPU-intensive, parallel programs of the sort that often require hours, or sometimes days of execution. We executed instances of these programs that took between 1.5 and 2 hours to run on sixteen processors in the absence of checkpointing. In all cases, it is clear how the programs scale in size, and how this scaling will affect the performance of checkpointing. The basic parameters of each application are presented in Table 2. We briefly describe each application, ordered by checkpoint size, below.

7.1 NBODY

NBODY computes N-body interactions among particles in a system. The program is written in C, and uses the parallel multipole tree algorithm [19]. The instance used in our tests was 15,000 particles and ten iterations.

The basic structure of the program is as follows. Each particle is represented by a data structure with several fields. The particles are partitioned among "slave" processors (sixteen in our tests) in such a way that processors that are "close to each other" (by some metric) reside in the same slave, to limit interslave communication. For this reason, slave processors can differ in the number of particles they hold and therefore in their sizes. For example, in our tests, the slave processors averaged 3.7 megabytes in size, but the largest was six megabytes.

At each iteration, the "location" field (among others) of each particle is updated to reflect the n-body interaction. Since the size of a particle's data structure is less than the machine's page size, this means that almost all pages of the slave processors are modified during each iteration, leading to poor incremental checkpointing behavior when the checkpointing interval spans multiple iterations. However, since much of each particle's data is left unmodified from iteration to iteration, only a few bytes per page are changed, resulting in good *diff*-based compression.

There are two parameters that affect the running time and memory usage of **NBODY**. These are the number of particles, which affects both time and space, and the number of iterations, which only affects the running time.

NBODY is the only application where the checkpoints are small enough to allow the same number of checkpoints (six) in both diskless and disk-based checkpointing.

7.2 MAT

MAT is a C program that computes the floating point matrix product of two square matrices using Cannon's algorithm [17]. The matrix size in our tests was 4,608×4,608, leading to 15.1 megabyte checkpoints per processor.

On a uniprocessor, matrix multiplication typically shows excellent incremental checkpointing behavior, since the two input matrices are read-only, and the product matrix is calculated sequentially, filling up whole pages at a time in such a way that once a product element is calculated, it is never subsequently modified [25]. However, most high-performance parallel algorithms, such as Cannon's algorithm, differ in this respect.

In Cannon's algorithm, all three matrices are partitioned in square blocks among the n processors (and it is assumed n is a perfect square). The algorithm proceeds in $\sqrt{n}$ steps. In each step, each processor adds the product of its two input submatrices to its product submatrix. Then the processors send their input submatrices to neighboring processors, receiving new ones in their place, and repeat until the product submatrices are calculated. The ramification of this data movement is that during the course of an iteration, *all* matrices are modified. Therefore, if checkpoints span iterations (as is the case in disk-based checkpointing), incremental checkpointing will have no beneficial effect. If multiple checkpoints are taken in the same iteration (as is the case in diskless checkpointing), then incremental checkpointing will be successful as in the uniprocessor case.

When pages are updated in **MAT**, they are updated in their entirety, leading to very poor *diff*-based compression.

MAT's time and space demands are determined by the size of the matrix. For an $N \times N$ matrix, the memory usage is proportional to N^2, and the running time is proportional to N^3. The communication patterns of **MAT** depend on the number of processors, and are the same for all matrix sizes.

MAT and **NBODY** are the only applications where it is possible to take more than one disk-based checkpoint during the program's execution. Three disk-based checkpoints (as opposed to seven diskless checkpoints) are taken in **MAT**.

7.3 PSTSWM

PSTSWM is a FORTRAN program that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method [14]. The instance used here simulates the state of a 3-D system for a duration of 102 (simulated) hours.

Like **NBODY**, **PSTSWM** modifies the majority of its pages during each iteration, but it only modifies a few bytes per page. Therefore, incremental checkpointing should show limited improvement, but *diff*-based compression should work well. **PSTSWM**'s checkpoints are large — approximately 25 megabytes per processor. However, since each machine has 96 megabytes of physical memory, two checkpoints may be stored in their entirety without stressing the limits of physical memory.

PSTSWM can scale in size by simulating a denser particle grid. Once the size is set, each iteration performs roughly the same actions. Therefore, simulating longer time frames increases the running time in a linear fashion without altering the general behavior (e.g. memory access pattern) significantly.

7.4 CELL

CELL is a parallel cellular automaton simulation program. Written in C, this program distributes two grids of cellular automata evenly across all the application processors. One grid is denoted *current*, and one is denoted *next*. The values of the *current* grid are used to calculate the values in the *next* grid, and then the two grids' identities are swapped. The instance used in our tests simulates a 18,512 by 18,512 cellular automaton grid for 475 generations.

During each iteration, **CELL** updates every automaton in the *next* grid. Therefore, if checkpoints span two or more iterations, all memory locations will be updated, rendering incremental checkpointing useless. Compressibility depends on the data itself. "Sparse" grids (where many automata take on zero values) may see little change in the automata's values over time, which can lead to good compression. Denser grids lead to less compression. In our tests, we used very sparse grids.

The program size is directly proportional to the grid size. The running time is proportional to the grid size times the number of iterations. Each pair of iterations performs the same operations, and thus has the same memory access and communication patterns.

7.5 PCG

PCG is a FORTRAN program that solves $Ax = b$ for a large, sparse matrix A using the "Preconditioned Conjugate Gradient" iterative method The matrix A is converted to a small, dense format, and then approximations to x are calculated and refined iteratively until they reach a user-specified tolerance from the correct values. In our tests, A is a 1,638,400 by 1,638,400 element sparse matrix, and the program takes 3,750 iterations.

The exact mechanics and memory usage of **PCG** are detailed in [26]. The salient points are as follows. The main data structures in the program may be viewed as many vectors of length N (in our instances, $N = 1,638,400$). These vectors are distributed among all the application processors. Roughly three quarters of these vectors are never modified once the program starts calculating. The rest are updated in their entirety at each iteration. Therefore, incremental checkpoints should be one quarter the size of non-incremental checkpoints. The data that gets updated at every iteration is stored densely on contiguous pages, offering little opportunity for *diff*-based compression.

The program size is directly proportional to N, and like **CELL** and **PSTSWM**, the running time is proportional to the size times the number of iterations.

Each application processor holds 66.6 megabytes worth of data in **PCG**. Therefore, one simple diskless checkpoint will not fit into memory. However, when incremental and copy-on-write checkpointing are employed, the application and one or two checkpoints consume just a few megabytes more memory than is available. The size of the checkpoints combined with the speed of Sun NFS results in the inability to take disk-based checkpoints of **PCG**. This is because the time to store one checkpoint is longer than the running time of the application.

8 Results

It should be reiterated that the instances for these tests were chosen to run for a period of time that was long enough to measure the impact of checkpointing and recovery. In all applications, there are natural input parameters which result in longer execution times and larger checkpoints. Our goal in these tests is to assess the performance of checkpointing so that users of longer-running applications may be able to project the expected running time of their applications in the presence of failures while employing the various checkpointing variants.

The raw data for the experiments is in the Appendix of this paper. All graphs in this section are derived directly from the raw data. In most cases, the tests were executed in triplicate. The number of times each test was executed plus the standard deviations in execution times is displayed in the tables in the Appendix. The tables and graphs display average data.

We concentrate on two performance measures: latency and overhead. Latency is the time between when a checkpoint is initiated, and when it may be used for recovery. Overhead has been defined previously. Overhead is a direct measure of the performance penalty induced on an application due to checkpointing. The impact of latency is more subtle, and will be discussed in detail in Section 9.

8.1 Checkpointing to disk

Figure 5 plots checkpoint latency and overhead of checkpointing to disk (the DISK-FORK tests). These are plotted as a function of the applications' per-processor checkpoint sizes. As displayed in leftmost graph, the latency in the DISK-FORK tests is directly proportional to the checkpoint size, achieving a bandwidth of 0.129 Mbytes/sec. Here bandwidth is calculated as per-processor checkpoint size times the number of processors, divided by the checkpoint latency. Using that information, the checkpoint latency of the **PCG** test is projected to be roughly 8,663 seconds.

The rightmost graph displays overhead as a function of checkpoint size. While the graph appears roughly linear, it should be noted that the overhead of checkpointing is not a simple function of checkpoint size. The bulk

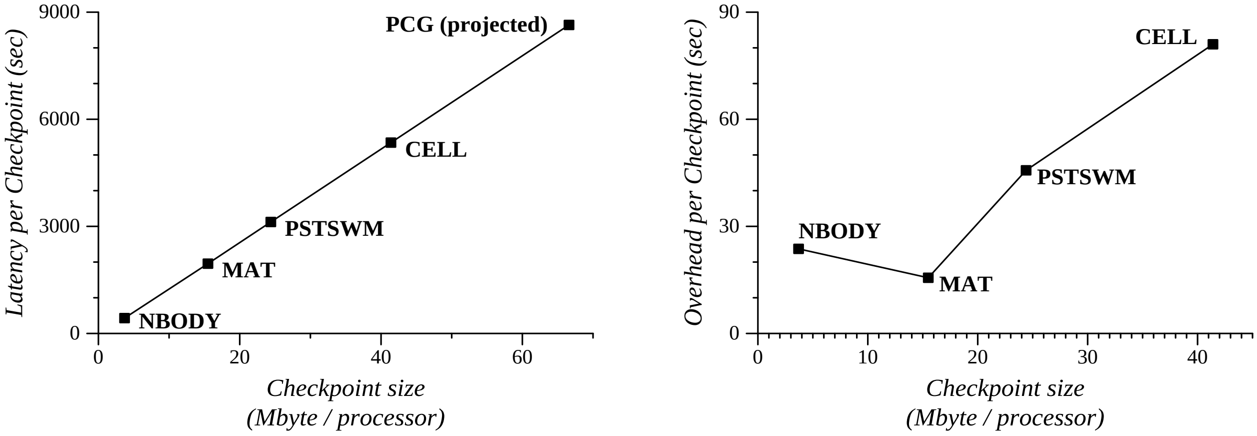

Figure 5: Checkpoint latency and overhead of checkpointing to disk (DISK-FORK)

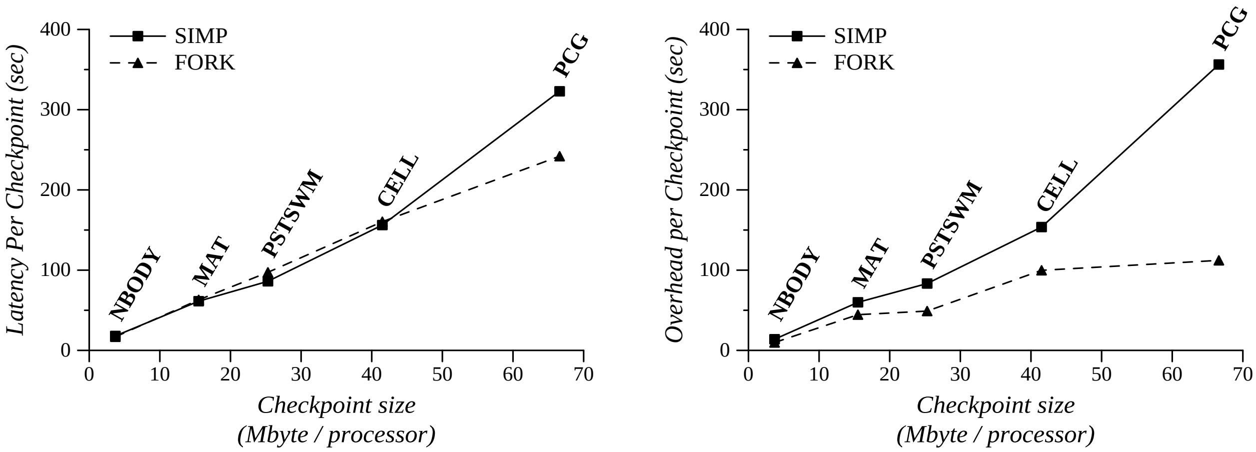

Figure 6: Checkpoint latency and overhead of SIMP and FORK

of work performed in checkpointing involves DMA from each processor's memory to its network interface card. The CPU is only affected significantly when one of the following occurs:

- A DMA transaction needs to be initiated or repeated.
- A copy-on-write page fault occurs in the application.
- There is contention for the memory bus.

There are also effects on the cache as a result of checkpointing. Therefore, although checkpoint size is a rough measuring stick for computing the overhead of DISK-FORK checkpointing, it is not the whole story. As has been shown in other research, the copy-on-write optimization does an excellent job of reducing overhead [9, 21, 25]. In this test, the overhead is between 0.7 and 5.5 percent of the checkpoint latency.

8.2 Diskless checkpointing: SIMP and FORK

Figure 6 plots checkpoint latency and overhead of the SIMP and FORK tests, again plotted as a function of checkpoint size. As in the DISK-FORK case, both the SIMP and FORK latencies are directly proportional to the checkpoint size, with the exception of the SIMP test in the **PCG** application. Here, the combined size of the application and its checkpoint exceeds the size of physical memory, resulting in pages being swapped to the backing store. This degrades the performance of checkpointing. In the FORK test, the checkpoint only requires an additional 16.6 Mbytes of memory, since the unmodified pages of memory are shared between the application and its checkpoint. Therefore, the checkpoint latency follows the same linear pattern as in the other applications. With the exception of the SIMP test in the **PCG** application, the bandwidth of checkpointing in SIMP and FORK is roughly 4.4 Mbytes/sec. This is a factor of 34 faster than the DISK-FORK bandwidth.

The overhead of the SIMP tests is identical to the latency, since the application is halted during checkpointing. In the FORK tests, the overhead is reduced by 29.4 (in **MAT**) to 53.7 (in **PCG**) percent. Although this is an improvement, it is not the same degree of improvement as in the DISK-FORK tests. The reason for this is that the CPU is more involved in diskless checkpointing than in disk-based checkpointing. In diskless checkpointing, the parity of each processor's checkpoint must be calculated, and this takes the CPU (plus some memory) away from the application. The only time when disk-based checkpointing makes more use of the CPU than diskless checkpointing is when the longer latency of checkpointing causes more copy-on-write page faults to occur.

8.3 The rest of the tests

All of the diskless checkpointing results are displayed in Figure 7. The top row of graphs shows the checkpoint latency for each test in each application. The middle row shows checkpoint overhead, and the bottom row shows the average checkpoint size. This is a bit of a misnomer, because in all cases, the in-memory and parity processor checkpoints are the same size. However, with incremental checkpointing and compression, fewer bytes are sent per processor. The "checkpoint size" graphs (and the "checkpoint size" columns in the Appendix) display the average number of bytes that each processor sends during checkpointing.

Some salient features from Figure 7 are as follows. First, incremental checkpointing significantly reduces the average checkpoint sizes in the **MAT** and **PCG** applications. In the other three applications, the checkpoint size of SIMP and INC are roughly the same. In the **MAT** and **PCG** applications, significant reductions in checkpoint latency and overhead result from incremental checkpointing. In both cases, the mixture of incremental and forked checkpointing result in the lowest overhead of the all diskless checkpointing tests.

When incremental checkpointing fails to decrease the size of checkpoints, as in the **NBODY** and **CELL** applications, the overhead of checkpointing is greater than with simple checkpointing. In both of these applications, the INC-FORK tests yielded the highest checkpoint latencies.

The results of *diff*-based compression are interesting. In three applications (**NBODY**, **PSTSWM** and **CELL**),

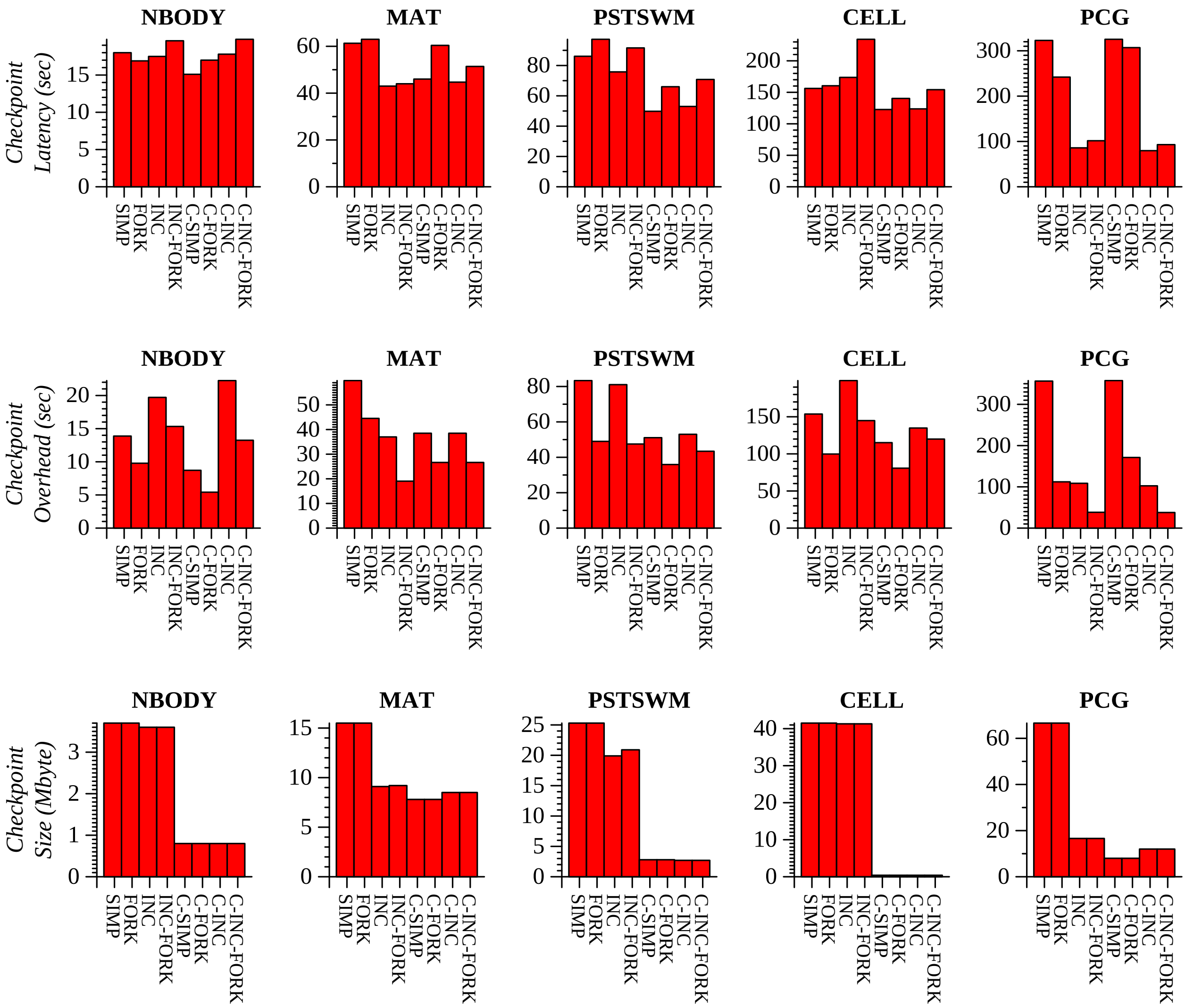

Figure 7: Diskless checkpoint latency, overhead, and size per application

incremental checkpointing fails because most of the programs' pages are updated at every iteration. However, *diff*-based compression succeeds in reducing checkpoint size because the pages are either sparsely modified (**NBODY** and **PSTSWM**) or updated with the same values (**CELL**). In these three applications, the C-FORK tests yielded the lowest checkpoint overhead. Note that since compression adds extra demands on the CPU, the reduction in overhead is not as drastic as with incremental checkpointing. It is also interesting to note that the lowest overhead is achieved with C-FORK rather than C-INC or C-INC-FORK. This is because in these tests, almost all pages are modified between checkpoints, and therefore incremental checkpointing merely adds the overhead of processing page faults.

In the other two tests (**MAT** and **PCG**), *diff*-based compression brings the checkpoint sizes of the FORK and SIMP tests to roughly the same size as incremental checkpointing. However, it does not improve upon incremental

Application	Recovery Time (sec)
NBODY	15.7
MAT	46.0
PSTSWM	66.3
CELL	138.3
PCG	375.3

Table 3: Recovery times for the SIMP tests

checkpointing in terms of size or overhead. This is because the modified pages showed little compressibility.

8.4 Recovery time

Table 3 shows the time that it takes the system to recover from a single failure and continue execution from the most recent checkpoint during the SIMP tests. Here, a processor failure is simulated by terminating one of the application processors. PVM has been written so that the other processors recognize this failure, and our modifications take advantage of this to automate the process of recovery. In our tests, the checkpointing processor takes the place of the failed application processor.

The recovery times are roughly equal to the checkpoint latencies of the SIMP applications. It should be noted that in all but the DISK-FORK tests, the recovery times are equal, since the entire diskless checkpoint of the failed processor must be calculated. In the DISK-FORK tests, the recovery times are equal to the checkpoint latencies. Thus, like the latencies, they are extremely large.

9 Discussion

9.1 Diskless vs. disk-based checkpointing

There are two basic results that we may draw from our tests concerning diskless vs. disk-based checkpointing:

- **The checkpoint latency and recovery time of diskless checkpointing is vastly lower than disk-based checkpointing.** As stated in section 8.2, the latency (and recovery time) of disk-based checkpointing is a factor of 34 slower than diskless checkpointing. This is a result of the poor performance of Sun NFS combined with the fact that all processors use the same disk.

- **The overhead of diskless checkpointing is comparable to disk-based checkpointing.** Figure 8 plots the overhead of disk-based checkpointing and the overhead of the best diskless variant for each application.

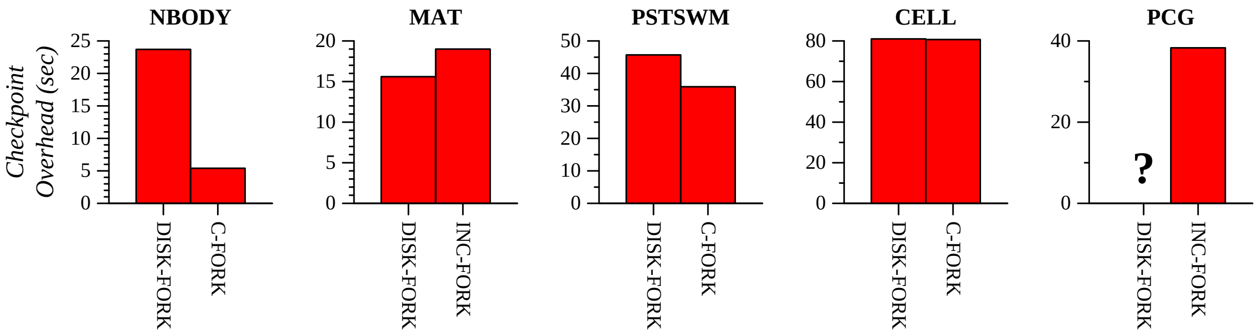

Figure 8: Checkpoint overhead of disk-based checkpointing as compared to the best diskless variant.

In some cases (**NBODY** and **PSTSWM**), diskless checkpointing outperforms disk-based, and in others (**MAT**) disk-based outperforms diskless. The question mark is plotted in PCG because we were unable to complete a disk-based checkpoint during the lifetime of the application.

There are two reasons why diskless checkpointing may be viewed as preferable to disk-based checkpointing. First, it lowers the expected running time of the application in the presence of failures. Second, it has less effect on the computing environment, which is of special concern if the environment is shared. We consider each of these in turn.

9.1.1 Expected running time

Supposing that failure rate is governed by a Poisson process, Vaidya has derived equations for assessing the performance of an application in the presence of checkpointing and rollback recovery [36]. These equations take as input the average overhead, latency, and recovery time per checkpoint, plus the rate of failures, and are defined as follows.

$$
\begin{aligned}
e^{\lambda(T_{opt}+O)}(1-\lambda T_{opt}) &= 1 \text{ for } T_{opt} \neq 0 && (1)\\
\Gamma &= \lambda^{-1} e^{\lambda(L-O+R)}(e^{\lambda(T_{opt}+O)}-1) && (2)\\
r &= \frac{\Gamma}{T_{opt}} - 1 && (3)\\
T_{ckp} &= T_{base}(r+1) && (4)\\
T_{nockp} &= \lambda^{-1}(e^{\lambda T_{base}}-1) && (5)
\end{aligned}
$$

where:

$\lambda =$ The rate of failures ($1/MTBF$).
$T_{opt} =$ The optimal checkpoint interval.
$O =$ The average overhead per checkpoint.
$L =$ The average latency per checkpoint.
$R =$ The average recovery time from a checkpoint.
$T_{base} =$ The running time of the application in the absence of checkpointing, recovery, and failures

(i.e. the BASE test).

$r =$ The "overhead ratio," which is a measure of the performance penalty due to checkpointing, recovery and failures[36].

$\Gamma =$ The expected running time of the optimal checkpoint interval in the presence of failures, checkpointing and recovery.

$T_{ckp} =$ The optimal expected running time of the application in the presence of failures, checkpointing and recovery.

$T_{nockp} =$ The expected running time of the application in the presence of failures, but no checkpointing and recovery (i.e. the application is restarted from scratch following a failure).

In all these equations, the repair time is assumed to be zero. This approximates the case when a spare processor is ready to continue computation immediately following a failure. If repair time is significant, then Eq's 2 and 5 become:

$$\Gamma = \lambda^{-1} e^{\lambda(L-O+R+T_{repair})}(e^{\lambda(T_{opt}+O)} - 1) \quad (6)$$

$$T_{nockp} = \lambda^{-1} e^{\lambda T_{repair}}(e^{\lambda T_{base}} - 1) \quad (7)$$

These equations may be used to compare checkpointing algorithms as follows. First, for each algorithm T_{opt} may be calculated from λ and O using Eq. 1. Next, Γ and r may be determined by Eqs. 2 and 3. If so desired, the expected running time of an application (T_{ckp}) for each algorithm may then be determined by Eq. 4. The checkpointing algorithm with the lowest value of r will be the one with the smallest expected running time. Thus, r suffices as a metric by which to compare checkpointing algorithms.

If T_{ckp} is greater than T_{nockp}, then the application cannot benefit from checkpointing. This occurs when the application's running time (T_{base}) is not significantly greater than T_{opt}. However, as T_{base} grows, T_{nockp} increases more rapidly than T_{ckp} to the point that checkpointing improves the program's expected running time in the presence of failures.

In Table 4, we use the data from Section 8 to derive values for T_{opt}, Γ, r, T_{ckp} and T_{nockp} for each of the tests presented in Figure 8. We calculated λ in the following manner. In their study of host reliability on the Internet, Long *et at* [22] determined an average MTBF of 29.29 days. Assuming independent processor failures, this means that the MTBF of a collection of 16 processors is $29.29/16 = 1.837$ days, and the MTBF of a collection of 17 processors is $29.29/17 = 1.729$ days. This gives λ a value of $6.301 * 10^{-6}$ failures per second for 16 processors, and $6.694 * 10^{-6}$ failures per second for 17 processors. We use the former value as the failure rate for disk-based checkpointing and for no checkpointing, and the latter value for diskless checkpointing.

Table 4 shows that in all applications, diskless checkpointing performs better than disk-based checkpointing. This can be seen in the lower expected running times (T_{ckp}), and the lower overhead ratios (r). Therefore, even though the two have similar checkpoint overheads, the extremely large latency and recovery time of disk-based checkpointing makes it unattractive in comparison to diskless checkpointing.

Another significant result of Table 4 is that in two applications, **NBODY** and **MAT**, the expected running time in the presence of failures is minimized by diskless checkpointing. In the other three applications, no checkpointing

Application	Test	T_{base} (sec)	T_{opt} (sec)	Γ (sec)	r	T_{ckp} (sec)	T_{nockp} (sec)
NBODY	DISK-FORK	5722	2727	2789	0.0229	5853	5826
NBODY	C-FORK	5722	1267	1278	0.0087	5772	5826
MAT	DISK-FORK	6602	2215	2302	0.0393	6862	6741
MAT	INC-FORK	6602	2370	2409	0.0166	6711	6741
PSTSWM	DISK-FORK	5610	3778	4024	0.0652	5976	5710
PSTSWM	C-FORK	5610	3251	3325	0.0229	5739	5710
CELL	DISK-FORK	6351	5017	5539	0.1040	7012	6480
CELL	C-FORK	6351	4856	5025	0.0350	6573	6480
PCG	INC-FORK	5874	3357	3444	0.0260	5991	5984

Table 4: Calculated values of T_{int}, Γ, r, T_{ckp} and T_{nockp}.

gives the smallest expected running time. That any checkpointing improves performance is somewhat surprising, given the relatively small execution times of the experiments with respect to the MTBF. There are no cases where disk-based checkpointing gives a smaller expected running time.

As the execution time of an application grows, checkpointing becomes much more attractive. For example, suppose the user desires to simulate 5000 hours in PSTSWM instead of 102. Then the program will take roughly 275,000 seconds, or 3.18 days. Such an execution would not alter the size of the checkpoints, and therefore we may use the same overhead, latency and recovery times as presented in Section 8. This leads to expected execution times of 3.256 days for diskless checkpointing, 3.390 days for disk-based checkpointing and 8.553 days for no checkpointing.

9.1.2 The effect on shared resources

Large checkpoint latencies can be detrimental in other ways. For example, in disk-based checkpointing, the entire latency period is spent writing checkpoint data to stable storage. If other programs or users share the stable storage, large checkpoint latencies are undesirable, because the performance of stable storage as seen by others is degraded for a long period of time.

In [23], the effect of DISK-FORK checkpointing on the performance of stable storage was assessed. While a DISK-FORK checkpoint was being stored to the central disk, a processor not involved in the application timed the bandwidth of disk writes. In that test, the performance of stable storage was degraded by 87 percent. This is significant, for it means that extremely long checkpoint latencies, such as those measured in our tests, have the potential to degrade the performance of the system in a severe manner for a long time. Diskless checkpointing, on the other hand, exhibits much smaller checkpoint latencies, and because the calculation of the checkpoint encoding involves both the network and the CPU, the impact on shared resources (in this case, the network) is far less [23].

9.2 Recommendations

Given the results of these experiments, we can make the following recommendations. Of the checkpointing variants tested in this paper, three stand out as the most useful: DISK-FORK, C-FORK and INC-FORK. On a system with similar performance to ours, each is the most useful in certain cases:

- If checkpoints are small or the likelihood of wholesale system failures is high, then DISK-FORK checkpointing should be employed.
- If the program modifies a few bytes per page between checkpoints, or if the machine does not provide access to virtual memory facilities, then C-FORK diskless checkpointing should be employed.
- If the program does not modify a significant number of pages between checkpoints, then INC-FORK diskless checkpointing should be employed.

Although we did not test such applications, there may be times when FORK and SIMP are the most useful checkpointing methods. This is when all pages are modified in a dense manner between checkpoints. Then FORK will have the lowest overhead when there is enough memory to store two checkpoints, and SIMP will have the lower overhead otherwise.

None of our applications would have benefited from incremental checkpointing to disk. However, if multiple checkpoints are taken and the program modifies only a fraction of its pages between checkpoints, incremental forked checkpoints will outperform DISK-FORK.

Finally, in interpreting the results, it is important to note that the speed of stable storage in these experiments is quite slow. A faster network, a faster file system, or a file system with multiple disks will improve the performance of disk-based checkpointing relative to diskless checkpointing. On the other hand, a system with more processors will degrade the performance of disk-based checkpointing relative to diskless checkpointing. It should be possible using the equations in Section 9.1.1 to extrapolate the results of our experiments to systems with different performance parameters.

10 Related Work

There has been much research performed on checkpointing and rollback recovery. The important algorithms and performance optimizations for disk-based checkpointing in parallel and distributed systems are presented in [8]. Research more directly related to diskless checkpointing is cited below.

The first paper on diskless checkpointing was presented by Plank and Li [27]. This paper may be viewed as a completion of that original paper.

Silva *et al* [32] implemented checkpoint mirroring on a transputer network, and performed experiments to determine that it outperformed disk-based checkpointing. Chiueh and Deng [6] implemented checkpoint mirroring and Raid Level 5 checkpointing on a massively parallel (4096 processors) SIMD machine. They found that mirroring improved performance by a factor of 10. Both implementations involved modifying the application to perform checkpointing, rather than simply relinking with a checkpointing library.

Scales and Lam [31] implemented a distributed programming system built on special primitives with shared-memory semantics. They use redundancy built into the system, plus checkpoint mirroring when necessary to tolerate single processor failures with low overhead. In a similar manner, Costa *et al* [7] took advantage of the natural redundancy in a distributed shared memory system to make it resilient to single processor failures. Both of these systems export a shared-memory interface to the programmer and embed fault-tolerance into the implementation with no reliance on stable storage.

Plank *et al* [26] embedded diskless checkpointing (with Raid Level 5 encoding) into several matrix operations in the ScaLAPACK distributed linear algebra package, thus making them resilient to single processor failures with low overhead. Kim *et al* [15] extended this work to employ one-dimensional parity encoding, which both lowers the overhead and increases the failure coverage.

In [23], diskless checkpointing ideas are extended to a disk-based checkpointing system where there is disparity between the performance of local and remote disk storage. In such environments, diskless checkpointing may be extended so that in-memory checkpoints are stored on local disks (which are fast, but do not survive processor failures), and checkpoint encodings are stored on remote disks (which are slow, but are available following a failure). The performance of mirroring, Raid Level 5, and Reed-Solomon codings are all assessed and compare favorably to standard checkpointing to remote disk. The impact of checkpointing on the remote disk and the network is also assessed.

Finally in [35], Vaidya makes the case for *two-level* recovery schemes, where a fast checkpointing method tolerating single processor failures is combined with a slower method that tolerates wholesale system failures. In his examples, checkpoint mirroring is employed for the fast method, and DISK-FORK checkpointing is employed for the slow method. His analysis applies to the methods presented in this paper as well.

11 Conclusion

Diskless checkpointing is a technique where processor redundancy, memory redundancy and failure coverage are traded off so that a checkpointing system can operate in the absence of stable storage. In the process, the performance of checkpointing, as well as its impact on shared resources is improved.

In this paper, we have described basic diskless checkpointing plus several performance optimizations. These have all been implemented and tested on five long-running application programs on a network of workstations and compared to standard disk-based checkpointing. In this implementation, the diskless checkpointing algorithms show a 34-fold improvement in checkpointing latency combined with comparable checkpoint overhead. The result is a lower expected running time in the presence of single processor failures.

Several checkpointing systems [6, 23, 26, 32] have included variants of diskless checkpointing to improve the performance of checkpointing. Designers of checkpointing systems should consider the variants of diskless checkpointing presented in the paper to optimize performance and minimize the impact of checkpointing on shared resources.

References

[1] A. Appel and K. Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991.

[2] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, September 1997.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual International Symposium on Computer Architecture*, pages 245—254, Chicago, IL, April 1994.

[4] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[6] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *26th International Symposium on Fault-Tolerant Computing*, pages 370–379, Sendai, June 1996.

[7] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[8] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[10] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5), May 1992.

[11] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[12] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jaing, and V. Sunderam. *PVM — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Boston, 1994.

[13] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. The MIT Press, Cambridge, Massachusetts, 1992.

[14] J. J. Hack, R. Jakob, and D. L. Williamson. Solutions to the shallow water test set using the spectral transform method. Technical Report TN-388-STR, National Center for Atmospheric Research, Boulder, CO, 1993.

[15] Y. Kim, J. S. Plank, and J. J. Dongarra. Fault tolerant matrix operations for networks of workstations using multiple checkpointing. In *High Performance Computing on the Information Superhighway, HPC Asia '97*, pages 460–465, Seoul, Korea, April 1997.

[16] B. A. Kingsbury and J. T. Kline. Job and process recovery in a UNIX-based operating system. In *Usenix Winter 1989 Technical Conference*, pages 355–364, San Diego, CA, January 1989.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1994.

[18] C. R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, September 1992.

[19] J. F. Leathrum, Jr. *Parallelization of the fast multipole algorithm: Algorithm and architecture design*. PhD thesis, Duke University, 1992.

[20] C-C. J. Li and W. K. Fuchs. CATCH – Compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.

[21] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[22] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *14th Symposium on Reliable Distributed Systems*, pages 2–9, Bad Neuenahr, September 1995. IEEE.

[23] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, pages 76–85, October 1996.

[24] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under unix. In *Usenix Winter 1995 Technical Conference*, pages 213–223, January 1995.

[26] J. S. Plank, Y. Kim, and J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43:125–138, September 1997.

[27] J. S. Plank and K. Li. Faster checkpointing with $N+1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[28] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[29] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[30] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In *25th International Symposium on Fault-Tolerant Computing*, pages 67–71, Pasadena, CA, June 1995.

[31] D. J. Scales and M. S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Usenix 1996 Technical Conference on UNIX and Advanced Computing Systems*, San Diego, January 1996.

[32] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High Performance Computing Conference*, pages 694–701, Knoxville, TN, May 1994.

[33] G. Stellner. Consistent checkpoints of PVM applications. In *First European PVM User Group Meeting*, Rome, 1994.

[34] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, February 1995.

[35] N. H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Ottawa, May 1995.

[36] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, August 1997.

[37] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In *25th International Symposium on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

[38] P. R. Wilson and T. G Moher. Demonic memory for process histories. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989.

Raw Data

NBODY

TEST	# of Runs	Running Time (sec)	(std dev)	# of Checkpoints	Avg Checkpoint Size (Mbytes)	Avg Checkpoint Latency (sec)
BASE	3	5722.0	15.6			
SIMP	3	5805.3	4.5	6	3.7	18.0
FORK	3	5780.7	4.6	6	3.7	16.9
INC	3	5840.3	3.7	6	3.6	17.5
INC-FORK	2	5814.0	13.0	6	3.6	19.6
C-SIMP	3	5774.3	11.9	6	0.8	15.1
C-FORK	2	5754.5	1.5	6	0.8	17.0
C-INC	2	5855.5	25.5	6	0.8	17.8
C-INC-FORK	2	5801.5	5.5	6	0.8	19.8
DISK-FORK	3	5864.3	44.9	6	3.7	430.3

MAT

TEST	# of Runs	Running Time (sec)	(std dev)	# of Checkpoints	Avg Checkpoint Size (Mbytes)	Avg Checkpoint Latency (sec)
BASE	3	6602.0	39.1			
SIMP	3	7021.0	17.5	7	15.5	61.3
FORK	3	6913.7	54.5	7	15.5	63.0
INC	3	6861.0	25.5	7	9.1	43.0
INC-FORK	3	6735.3	3.4	7	9.2	44.0
C-SIMP	2	6871.5	9.5	7	7.8	46.0
C-FORK	2	6788.5	0.5	7	7.8	60.4
C-INC	2	6871.5	26.5	7	8.5	44.7
C-INC-FORK	2	6788.5	9.5	7	8.5	51.4
DISK-FORK	3	6648.7	37.3	3	15.5	1955.0

PSTSWM

TEST	# of Runs	Running Time (sec)	(std dev)	# of Checkpoints	Avg Checkpoint Size (Mbytes)	Avg Checkpoint Latency (sec)
BASE	3	5610.0	22.8			
SIMP	3	6110.0	5.7	6	25.3	86.1
FORK	3	5904.0	18.5	6	25.3	97.3
INC	3	6096.3	28.4	6	19.9	75.8
INC-FORK	3	5895.0	46.0	6	20.9	91.6
C-SIMP	2	5916.5	19.5	6	2.8	49.8
C-FORK	2	5825.5	17.5	6	2.8	66.0
C-INC	2	5928.0	2.0	6	2.7	53.0
C-INC-FORK	2	5870.5	9.5	6	2.7	70.8
DISK-FORK	3	5655.7	15.5	1	24.4	3122.7

CELL

TEST	# of Runs	Running Time (sec)	(std dev)	# of Checkpoints	Avg Checkpoint Size (Mbytes)	Avg Checkpoint Latency (sec)
BASE	3	6351.3	16.9			
SIMP	3	7119.7	8.5	5	41.5	156.2
FORK	3	6850.3	33.3	5	41.5	160.4
INC	2	7345.0	55.0	5	41.3	173.7
INC-FORK	2	7075.5	42.5	5	41.3	234.2
C-SIMP	3	6927.0	47.1	5	0.4	122.7
C-FORK	3	6755.0	5.0	5	0.4	140.3
C-INC	2	7025.5	6.5	5	0.4	123.7
C-INC-FORK	1	6951.0	0.0	5	0.4	154.2
DISK-FORK	3	6432.3	6.9	1	41.4	5346.0

PCG

TEST	# of Runs	Running Time (sec)	(std dev)	# of Checkpoints	Avg Checkpoint Size (Mbytes)	Avg Checkpoint Latency (sec)
BASE	3	5873.7	19.3			
SIMP	3	8011.7	34.2	6	66.6	322.9
FORK	3	6546.7	36.4	6	66.6	242.0
INC	3	6525.3	17.9	6	16.6	85.9
INC-FORK	3	6103.7	6.9	6	16.6	101.6
C-SIMP	2	8019.0	2.0	6	8.0	325.4
C-FORK	2	6901.0	24.0	6	8.0	307.0
C-INC	2	6488.5	52.5	6	12.0	79.7
C-INC-FORK	2	6100.5	8.5	6	12.0	93.0