Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques

James S. Plank

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

Appearing in:
SRDS-15: 15th Symposium on Reliable Distributed Systems
October, 1996

Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques

James S. Plank*
Department of Computer Science
University of Tennessee

Abstract

Coordinated checkpointing systems are popular and general-purpose tools for implementing process migration, coarse-grained job swapping, and fault-tolerance on networks of workstations. Though simple in concept, there are several design decisions concerning the placement of checkpoint files that can impact the performance and functionality of coordinated checkpointers. Although several such checkpointers have been implemented for popular programming platforms like PVM and MPI, none have taken this issue into consideration.

This paper addresses the issue of checkpoint placement and its impact on the performance and functionality of coordinated checkpointing systems. Several strategies, both old and new, are described and implemented on a network of SPARC-5 workstations running PVM. These strategies range from very simple to more complex, borrowing heavily from ideas in RAID (Redundant Arrays of Inexpensive Disks) fault-tolerance. The results of this paper will serve as a guide so that future implementations of coordinated checkpointing can allow their users to achieve the combination of performance and functionality that is right for their applications.

1. Introduction

Coordinated checkpointing is a well-known technique for providing fault-tolerance in parallel programs [4, 8, 14]. With coordinated checkpointing, all processors coordinate to define a global consistent state and then save that state to disk. Upon a failure, all processors roll back to that saved state and restart the computation. Coordinated checkpointing is attractive over other methods (e.g. optimistic or independent checkpointing [12, 13, 32, 37, 39]) because it is simple, it never requires multiple rollbacks, and it can handle nondeterminism in the program. Moreover, the same checkpointing mechanism can be used to implement three very useful functionalities: process migration, coarse-grained job-swapping, and tolerance of processor failures.

Recently there have been three separate implementations of coordinated checkpointing on PVM and MPI, the *de facto* standards for parallel programming on networks of workstations [11, 20]. These are Fail-Safe PVM [16], MIST [3], and CoCheck [30, 31]. All three have similar goals — to build public-domain tools that use transparent coordinated checkpointing for process migration, coarse-grained job swapping and wholesale fault-tolerance.

This paper focuses on a deficiency in all three tools: the inability to provide a useful degree of fault-tolerance with good performance for the variety of workstation networks that exist. New approaches are suggested that solve this deficiency with very little extra complexity. These approaches apply RAID techniques [5] to standard coordinated checkpointing. We evaluate the performance of the old and new approaches as observed on a network of SPARC-5 workstations running PVM. The goal of this work is to present a convincing argument for these approaches to be incorporated as standard features in future releases of coordinated checkpointing tools.

2. Coordinated Checkpointing

Coordinated checkpointing is a topic in fault-tolerance with a very rich history [7]. The goal of coordinated checkpointing is for a collection of processors with distinct memories and clocks to define and store a *consistent checkpoint*. This consistent checkpoint is

*`plank@cs.utk.edu`. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9409496, and by the ORAU Junior Faculty Enhancement Award.

composed of local checkpoints for each processor, and a log of messages. The system recovers from the consistent checkpoint by having each processor recover from its local checkpoint and then replaying messages from the log. The major obstacles in coordinated checkpointing are defining the points at which each processor should checkpoint, and determining the composition of the message log.

There have been many algorithms presented for coordinated checkpointing (e.g. [1, 4, 14, 15, 36]). Implementations have shown that the major source of overhead in checkpointing systems is saving the processor states. The choice of consistency algorithm is largely immaterial [8, 9, 27]. For this reason, an algorithm called *"sync-and-stop"* is usually implemented. In this algorithm, the processors are frozen until all messages that are currently in transit have reached their destinations. The processors then checkpoint and resume the computation. No message logging is necessary. This algorithm is attractive because of its simplicity. It is easy to implement on most message-passing systems, and its performance is not noticeably worse than more complex algorithms [27].

Since 1993, there have been three coordinated checkpointers either released as public domain, or targeted for release in the near future. These are Fail-Safe PVM [16], MIST [3], and CoCheck [30, 31]. All three have been implemented on top of PVM, a very popular public-domain system for parallel programming on networks of workstations [11]. In addition, CoCheck has been implemented on top of MPI [20], which can be viewed as the next generation of PVM.

These checkpointers very similar. They are designed to work on networks of standard Unix workstations with no modification to the workstations' operating systems. Moreover, they are transparent. In other words, the application programs also do not need to be modified. Instead, they are linked with a checkpointing library, and checkpointing is controlled through initialization files, command line arguments, or external processes. All use the sync-and-stop algorithm for checkpoint consistency, and therefore require no message logging.

One functionality implemented by all three tools is process migration. For process migration all processors do not explicitly checkpoint. Instead, a processor that is either too heavily loaded or needed by its owner checkpoints itself to a spare processor which then takes over its computation. The message-passing system must be modified so that messages destined for the old processor get rerouted. For PVM and MPI, this modification is simplified by coordinated checkpointing and the sync-and-stop algorithm. If all processors coordinate so that there are no outstanding messages during the migration, then the rerouting is simplified greatly [31]. The performance of process migration depends solely on the bandwidth of the network and thus each of the three implementations has optimal performance. We do not discuss process migration further in this paper.

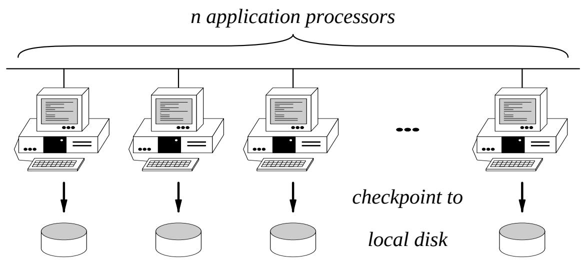

Figure 1. CGJS checkpointing

The checkpointers also implement two other checkpointing strategies – *CGJS (coarse-grained job swapping) checkpointing*, and *CFS (central file server) checkpointing*. With CGJS checkpointing, each processor checkpoints to a local disk (see Figure 1). This kind of checkpointing is ideal for coarse-grained job swapping: after the processors have checkpointed, the application is terminated. To restart the application, the same processors restart from their local checkpoints.

Alternatively, new processors can be used if they are able to access the old processors' checkpoints. In this way, CGJS checkpointing may also be used for fault-tolerance.

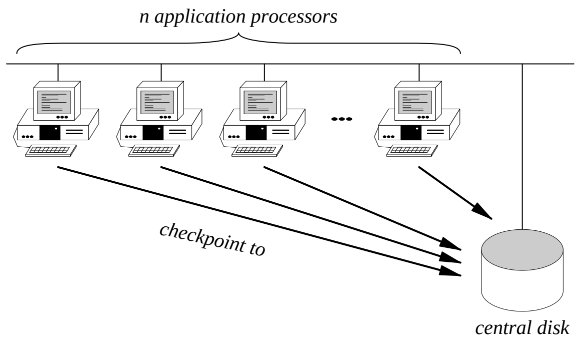

Figure 2. CFS checkpointing

With CFS-checkpointing, the processors all checkpoint to a central file system that is assumed to be stable (see Figure 2). If any number of processors fail, then the computation is halted completely. It then restarts using the same number of machines and checkpoints from the central file system. It is assumed that all machines are of the same architecture, and that checkpoint files from one machine may be restarted by another.

3. Performance Considerations

While the performance and functionality of these three checkpointers is good for many applications and workstation environments, there are many circumstances where they are lacking. To fully understand why, we first describe the various criteria by which the performance of coordinated checkpointing can be judged.

Checkpoint Overhead: Checkpoint overhead is defined to be the time added to the running time of the target program as a result of checkpointing. This is the single most important part of a checkpointer's performance — if the checkpoint overhead is too high, users would rather risk having a failure than enduring the performance degradation of checkpointing.

Checkpoint Latency: Checkpoint latency is the time that it takes for the checkpointer to complete a checkpoint, from start to finish. Checkpoint latency affects the interval of checkpointing, since a new checkpoint cannot be started until the previous checkpoint has completed. It also affects the progress of the application following a failure, since the program can only roll back to completed checkpoints. In unoptimized checkpointers, the checkpoint overhead equals the sum of the checkpoint latencies, and thus the two can be viewed as equivalent.

An important optimization to checkpointing systems is the "copy-on-write" optimization [8, 18, 25], where an in-memory copy of the checkpoint is made using copy-on-write, and that copy is written to disk while the application continues to execute. This optimization is very effective at decreasing the overhead of checkpointing while slightly increasing the latency.

Vaidya has demonstrated that small fluctuations in checkpoint overhead have far more impact on the performance of a checkpointing system than large fluctuations in checkpoint latency [35]. In other words, as long as the latency is smaller than the desired checkpointing interval, its value is of little importance.

Recovery Time: This is the time that it takes for the system to recover from a failure. Like checkpoint latency, recovery time is only important if the chances of failure are high.

Disk Space Consumed: This is the total amount of disk space that is used to take a checkpoint. Since the total size of coordinated checkpoints can be very large on a parallel system, a user must check to see if there is enough disk space available for checkpointing. Moreover, since the previous checkpoint cannot be deleted until the current checkpoint is completed, there must be at least two checkpoints' worth of disk space available to a user that plans to use checkpointing [14].

Fault Coverage: This is the number of processor failures that a checkpointing system can tolerate. If the parallel processing system can operate in the face of single processor failures and failures occur independently, then it is often beneficial to checkpoint in such a way that only single processor failures are tolerated. This is because checkpointing for single processor failures can incur far less overhead than checkpointing for wholesale system failures. Vaidya has shown that such cases warrant a two-level checkpointing scheme, where checkpoints for single-processor failures are taken frequently with low overhead, and checkpoints tolerating wholesale failures are taken infrequently with a higher overhead [34].

Impact on Shared Resources: Most workstation networks are shared. Either the workstations are privately owned and lent to a public pool when not used by the owner, or the entire network is shared by all users. We assume that the workstations used by the application are allocated for exclusive use. However, the network and the central file servers may be shared by all users. Thus, the impact of checkpointing on the network and the central file servers can be a significant concern.

3.1. Local vs. Central Storage

A critical feature in the performance of coordinated checkpointing is the difference between local and central disk storage. The following are assumptions that we make concerning these two types of storage:

- Each processor has access to local disk storage. This storage is relatively fast and is only accessible by that processor. In the case of a processor crash, the contents of this disk are unavailable until the processor resumes operation or someone physically removes the disk and attaches it to a functional processor.
- Each processor has access to a central file server. The disks belonging to this server have more capacity than the local disks and are often faster. However, the processors perform file operations to this central server via a network file system like NFS, and often exhibit performance that is far worse than their local disks. We assume that the file server is always available. (For example, many vendors sell special "server" machines with built-in fault-tolerance for high availability).

The ramification of these assumptions on Fail-Safe PVM, MIST and CoCheck is that CGJS checkpointing is fast, while CFS checkpointing can be extremely slow, since *all* processors write to the central file system at one time. The checkpoint overheads need not be large, since all three systems use the "copy-on-write" optimization. However, the latencies and recovery times of CFS checkpointing are very large.

CGJS checkpointing has no impact on shared resources, since all checkpointing activity is local. However, CFS checkpointing saturates the central file server for the duration of checkpointing, which as mentioned above, can be very long. Thus, CFS checkpointing can have a very negative impact on the central file system.

4. Common Workstation Networks

In assessing whether CGJS and CFS checkpointing are sufficient for long-running parallel applications on networks of workstations, we must ask ourselves whether CFS checkpointing is ever necessary, and if so, whether checkpoint latency, recovery time, and load on the central file server are important performance criteria. In this section we describe four common types of workstation networks, detailing their defining characteristics and the impact they have on the performance of checkpointing.

Privately Owned by the User: In these workstation networks, the user exclusively owns everything: processors, network and central file servers. In such an environment, the user's only concern is for hardware and transient software failures. Both are relatively infrequent [19], and can be tolerated by checkpoints to local disk. If a processor fails and remains unavailable, the user can physically move its local disk to an available processor. In such a network, the user's sole concern is minimizing checkpoint overhead.

Shared Pool of Workstations: Here a organization or department owns a collection of workstations that all users must share. In such environments, a user may be able to allocate processors exclusively, but the network and file servers are shared by all. Often in such environments, interactive users of workstations take priority over CPU-intensive users, and thus a user who is executing a long-running program on many machines may be asked to relinquish a machine. Thus, failures are more common than in a privately owned network. Moreover, the user must be concerned with the impact of his or her program on shared resources. In particular, if the user degrades the performance of the network or file server for a significant period of time, his or her jobs may be terminated by the system manager. If a processor does fail, the local disk may or may not be movable, depending on the machine and the department's policy.

Shared Pool of Privately Owned Workstations: In these networks, workstations are owned by individuals. When the owners are not using the workstations, they are "loaned" to a pool for other users. When the owners return, they reclaim their machines from the pool. Such is the environment supported by Condor [33]. One requirement of such a system is that when a user reclaims his/her machine, there should be no residual load from other jobs. Therefore, once a machine is revoked, its memory and local disk should be unavailable to other users.

In these networks, the users desiring CPU capacity must be concerned with processor revocations that look like permanent processor failures and may be frequent [21]. Moreover, since the network and file servers are shared, users must also be concerned with the impact of their programs on shared resources.

Combination Networks: Some networks are a combination of the above. For example, at the University of Tennessee, our network of 24 SPARC-5 workstations has two roles. During the day, it is used by students for classwork and labs, and thus fits the model of a shared pool of workstations: only idle processors may be employed for general-purpose CPU-intensive computing. From 12:15 AM to 8:45 AM however, it may be allocated exclusively as a parallel programming platform, and at this time it resembles a network that is privately owned by the user. One major difference is that if a processor fails and remains failed in this environment, its local disk becomes unavailable, because the machine room is inaccessible during the late evening.

4.1. Ramifications

There are two important characteristics of the above networks that affect coordinated checkpointing. First, in shared workstation environments, the probability of failures is much higher than in a private network. Moreover, these failures can be viewed as permanent, involving the loss of the failing processors' local disks. Second, in all but the privately owned network, the impact on shared resources is a valid concern. The combination of these two characteristics means that both CFS and CGJS checkpointing are undesirable. With the permanent loss of local disks, CGJS checkpointing has no fault-coverage, and while CFS checkpointing can tolerate all failures, it has poor latency and recovery time, and a very severe impact on the performance of the central file server.

5. RAID Strategies for Storing Checkpoints

In this section, we present three different strategies for storing coordinated checkpoints. All three stem from the fault-tolerant mechanisms of RAID systems.

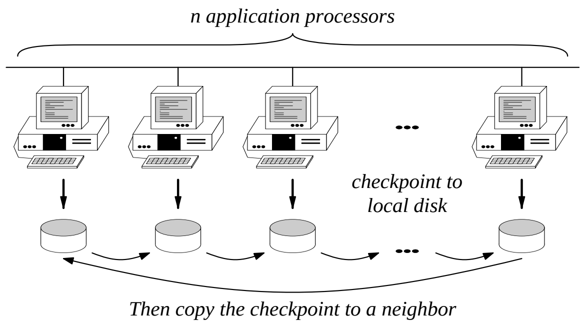

Figure 3. MIR checkpointing

MIR: Checkpoint Mirroring: This is a simple approach to checkpointing for single processor failures. It was suggested by both León [16] and Vaidya [34] and is analogous to disk mirroring in RAID systems [5]. With checkpoint mirroring, a CGJS checkpoint is taken, and then each processor copies its checkpoint to another processor's local disk (see Figure 3). This can be done by background processes on both machines that cooperate to copy the checkpoint file.

If a processor fails, its checkpoint will be available on the local disk of a non-failed processor. This allows a spare processor to take its place without recovering the failed processor's local disk. The logical extension to disk mirroring to recover from $m > 1$ processor failures is to have each processor store its checkpoint on the local disks of m other processors.

Like its RAID counterpart, checkpoint mirroring has good performance, but uses a great deal of space: $m+1$ checkpoints per processor.

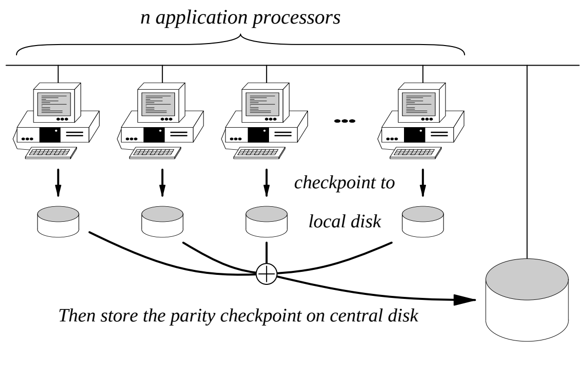

Figure 4. PAR checkpointing

PAR: $n+1$-Parity: This is analogous to RAID Level 5 fault-tolerance [5], and is motivated by diskless checkpointing techniques [26]. Like checkpoint mirroring, it tolerates single processor failures, but with considerably less space overhead. With $n+1$-parity, a CGJS checkpoint is taken, and then the processors cooperate to store a *parity checkpoint* on the central disk (see Figure 4). The parity checkpoint is defined to be the bitwise exclusive-or ($\oplus$) of each processor's checkpoint. In other words, assuming that there are n application processors, the i-th byte of the parity checkpoint, $c_{n+1,i}$, is defined to be

$$c_{n+1,i} = c_{1,i} \oplus c_{2,i} \oplus \ldots \oplus c_{n,i},$$

where $c_{j,i}$ is the i-th byte of processor j's checkpoint (or zero if processor j's checkpoint is smaller than i bytes).

If a processor fails, then its checkpoint can be reconstructed as the bitwise exclusive or of the remaining processors' checkpoints and the parity checkpoint. Thus, with $n+1$-parity, single processor failures may be tolerated with $n+1$ total checkpoints as opposed to $2n$ checkpoints with checkpoint mirroring.

RS: Reed-Solomon Coding: The logical extension of $n+1$-Parity for tolerating $m > 1$ failures is Reed-Solomon coding. This technique is too complex to describe in detail — the interested reader is referred to papers by Schwarz [28] and Plank [23]. To sketch, after taking the CGJS checkpoint, the processors cooperate to store m checkpoints on central storage. The i-th byte of each of these checkpoints is defined to be a linear combination of the i-th byte of each processor's checkpoint:

$$c_{n+j,i} = \sum_{k=1}^{n} j^{k-1} c_{k,i}$$

where arithmetic is performed over the Galois Field $GF(2^8)$. Since arithmetic over $GF(2^8)$ is more expensive to compute than parity, each checkpoint takes longer than with $n+1$ parity. However, if disk space is an issue, RS checkpointing consumes far less space ($n + m$ checkpoints) than solutions based on disk mirroring ($(m + 1)n$ checkpoints).

5.1. CFS-BUF Checkpointing

As an aside, we mention one more checkpointing strategy which improves the overhead of CFS checkpointing. It is a simple use of CGJS checkpoints as buffers for checkpointing to central storage. Specifically, the processors first take a CGJS checkpoint and then each processor spawns a background process that copies its local checkpoint to central storage. CFS-BUF checkpointing has a distinct advantage over CFS checkpointing. First, CFS checkpointing is only reasonable if combined with the copy-on-write optimization. With copy-on-write, a page is only physically copied if the application program writes to it while the checkpoint is being taken. Thus, the long checkpoint latencies of CFS checkpointing increase the probability of pages being copied. If the application processes consume a great deal of memory, then the process and its copy can exceed physical memory, leading to degraded performance. With CFS-BUF checkpointing, the copy is extant only during the CGJS checkpoint. After that point, each processor can copy its local checkpoint to central storage in small increments, thus keeping the memory requirements of checkpointing to a minimum.

6. Implementation and Experiments

We implemented the six algorithms described above (CGJS, CFS, CFS-BUF, MIR, PAR and RS) in a small library to be linked with PVM programs. We then tested it on a challenging, long-running application called PCELL. PCELL is a program that executes a large grid of cellular automata for numerous iterations. PCELL is challenging for a checkpointing system for several reasons. First, for each pair of iterations, it writes almost every word in memory. Thus, incremental checkpointing [10, 38] cannot optimize the performance if checkpoints are taken more than two iterations apart. Second, if checkpoints have a high latency, then the copy-on-write optimization does not work very well since many pages are written to and copied during checkpointing. Third, the checkpoint images are not patterned, and therefore do not compress well. And fourth, the processors synchronize after each iteration, meaning that staggering checkpoints to reduce contention results in increased overhead [22]. PCELL is also convenient because each pair of iterations performs the same actions. Thus, as long as the same grid size is used, one can use the results of checkpointing short runs to project the behavior of checkpointing longer runs.

We executed PCELL on a network of SPARC-5 workstations at the University of Tennessee. This network is composed of 24 workstations connected by a fast (100 megabit) switched Ethernet. Although the network is "100 megabit", the SunOS drivers are unable to get peak performance from this configuration. The actual bandwidth between two random nodes is about 2 Mbyte/sec. Each workstation has 96 Mbytes of physical memory and a local disk with 550 Mbytes of storage that can be accessed at a bandwidth of approximately 1.7 Mbytes/sec. The network is connected to the department's central file servers by a standard (10 megabit) Ethernet running NFS. This disk also has a bandwidth of 1.7 Mbytes/sec, but the performance of NFS on the Ethernet is far worse. With NFS, remote file writes achieve a bandwidth of 0.11 Mbytes/sec.

Instance	# of Iter-ations	Running Time (sec)	Running Time (hh:mm:ss)	Memory Usage per Processor (Mbytes)
SMALL-28	28	912	00:15:12	8.2
LARGE-03	3	794	00:13:14	64.1
LARGE-09	9	2331	00:38:51	64.1
LARGE-35	35	9028	02:30:28	64.1

Table 1. Performance of PCELL without checkpointing

We ran two sets of tests – one with a small (8192×8192) grid, and one with a large (23152×23152) grid, both on 16 processors. Table 1 summarizes the performance of PCELL in the absence of checkpointing. Note that several numbers of iterations were used for the large grid. This was so that the checkpointing tests could complete in a reasonable amount of time. Since PCELL scales linearly with the number of iterations, this is a valid experimental short-cut.

In the SMALL tests, each processor calls `fork()` to spawn a child process that takes the checkpoint. This is the standard way to perform copy-on-write checkpointing in Unix systems [16, 25]. In the LARGE tests, the `fork()` system call fails, citing not enough memory. Therefore, in these tests, the processors are stopped until the CGJS checkpoints are taken. The application processes then resume and all other checkpointing activity occurs in the background. There is no CFS checkpoint taken in the LARGE tests because of the extremely high overhead (over 8000 seconds) that would occur.

All results were collected when the machines, the fast network and the central file server were allocated exclusively for our use. In each test, one checkpoint is taken shortly after the processors begin execution. The results of these tests are displayed in Table 2. The graphs in Figures 5 and 6 are derived from the numbers in this table.

6.1. Checkpoint Overhead

As expected, in both cases the CGJS strategies have the lowest checkpoint overheads. The PAR and MIR strategies have higher overheads, but these are still relatively small. The highest overheads belong to the RS strategies. This is because they need a considerable

Test	Alg.	Running Time (sec)	Checkpoint Overhead (sec)	Checkpoint Latency			Disk Space Consumed		Fault Coverage (# Perm. Failures)
				Total (sec)	CGJS Latency (sec)	Rest (sec)	Local (Mbytes)	Central (Mbytes)	
SMALL-28	CGJS-COW	915.1	3.0	4.5	4.5	0.0	130.5	0.0	0
SMALL-28	MIR-COW	926.1	14.0	19.2	4.5	14.7	261.0	0.0	1
SMALL-28	PAR-COW	932.5	20.4	60.1	4.4	55.7	130.5	8.2	1
SMALL-28	RS-2-COW	949.7	37.6	232.5	4.3	228.2	130.5	16.3	2
SMALL-28	RS-3-COW	966.8	54.7	286.7	4.4	282.2	130.5	24.5	3
SMALL-28	CFS-COW	938.5	26.3	867.2	0.0	867.2	0.0	130.5	16
SMALL-28	CFS-BUF-COW	924.9	12.8	821.8	4.0	817.8	130.5	130.5	16
LARGE-03	CGJS	852.9	59.3	40.5	40.5	0.0	1026.1	0.0	0
LARGE-03	MIR	973.8	180.3	199.4	39.8	159.5	2052.3	0.0	1
LARGE-03	PAR	1098.8	305.2	563.5	38.6	524.9	1026.1	64.1	1
LARGE-09	RS-2	3129.7	798.5	1820.7	39.6	1781.1	1026.1	128.3	2
LARGE-09	RS-3	3373.3	1042.1	2323.8	39.0	2284.8	1026.1	192.4	3
LARGE-35	CFS-BUF	9529.7	501.9	8564.8	47.5	8517.3	1026.2	1026.2	16

Table 2. Performance of PCELL with checkpointing

number of CPU cycles to compute two and three encodings of the checkpoint files respectively.

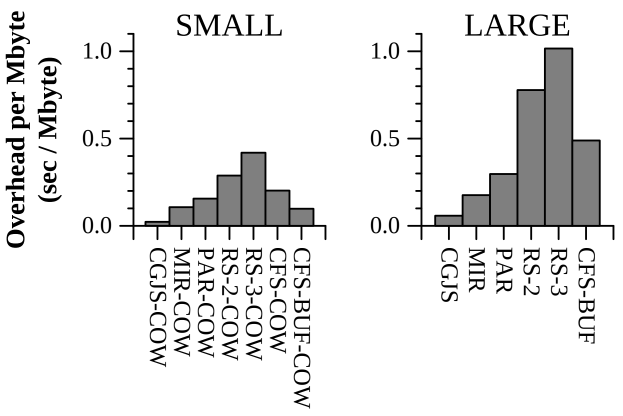

Figure 5. Overhead per megabyte of checkpointing

Figure 5 displays the overheads of all the strategies normalized per Mbyte of the application (i.e. 130.5 Mbytes for the SMALL instances, and 1,026 Mbytes for the LARGE instances). The SMALL instances show smaller values of this metric because of the copy-on-write optimization. In the SMALL instances, the CFS-BUF-COW strategy outperforms the CFS-COW strategy by roughly a factor of two. These savings come from the better use of physical memory and the reduction of copy-on-write page faults. The CFS-BUF strategy has a rather high overhead on the LARGE instances. This is from the operating system overhead that accumulates during more than two hours of NFS writes that saturate the Ethernet.

6.2. Checkpoint Latency and Recovery

Figure 6 displays the latencies of all the strategies normalized per Mbyte of the application. This figure suggests that the strategies fall roughly into to four groups – extremely fast (CGJS), very fast (MIR and PAR), not so fast (RS-2 and RS-3), and extremely slow (CFS and CFS-BUF).

Recovery time is roughly equivalent to the "Rest" portion of checkpoint latency (column 7 in Table 2). Thus, recovery time is roughly equal to the checkpoint latency. It should be obvious that for applications and environments that may exhibit frequent failures (for example in the shared pool of privately owned workstations), the CFS-based strategies are far too slow to deliver a decent degree of performance. The same may be said for the RS strategies.

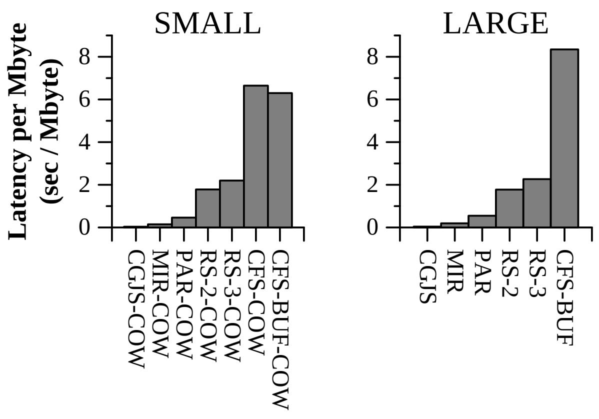

Figure 6. Latency per megabyte of checkpointing

6.3. Disk Space Consumed

Columns 8 and 9 of Table 2 show the total disk spaced consumed (local and central) by each checkpointing strategy. One notable feature of this data is the enormous size of the checkpointing files. As processors continue to get more local memory and workstation networks continue to scale, checkpoint sizes will continue to grow. Thus, disk space can indeed be a concern. The CGJS and CFS strategies both minimize the disk space consumed. The PAR and RS strategies follow closely behind. The MIR and CFS-BUF strategies both use twice the minimal amount of disk space, which may limit their effectiveness in some systems.

For example, the local disks in our labs each contain 550 Mbytes of storage, enough to hold five checkpoints of an application that uses all 96 Mbytes of memory. As mentioned in Section 3, there are times when a coordinated checkpointer needs to hold two checkpoints on disk. This means that on our system, we cannot extend the MIR strategy to tolerate two or more processor failures.

6.4. Fault Coverage

The last column of Table 2 shows the fault coverage of each checkpointing strategy. This has been discussed previously in Sections 2 and 5, and warrants no further discussion.

6.5. Impact on Shared Resources

There are two shared resources that can be affected by checkpointing: the central file server and the network. To test the effect of checkpointing on the central file server, we measured the bandwidth of disk writes from a separate processor to the central disk while the checkpointer was checkpointing. The results are presented in Figure 7.

The results are as expected. While the PAR and RS algorithms degrade the disk performance by 34 percent, the CFS strategies degrade the performance by 88 percent. Considering this degradation lasts for a long time (over 2 hours in the LARGE tests), the CFS strategies should be considered unusable in an environment where impact on shared resources is a concern. In contrast, the PAR and RS algorithms cause a slight degradation for a shorter period of time, and should be satisfactory for these environments. The CGJS and MIR algorithms have no impact on the central file server.

To test the impact on the network, we measured the bandwidth of sending 20 Mbyte messages between two non-participating processors on the network while the

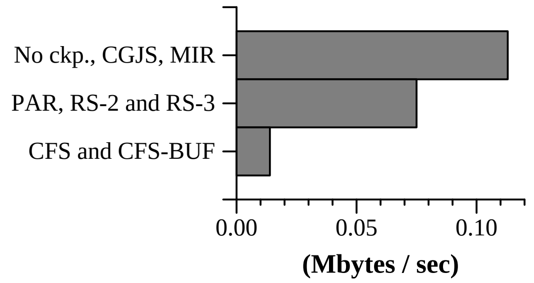

Figure 7. Disk bandwidth during checkpointing

other processors were checkpointing the LARGE instances. During the "control" run, the bandwidth was measured while the program executed with no checkpointing. These measurements were taken on both the fast Ethernet and the regular Ethernet and are presented in Figure 8. The lines above and below the marks represent the standard deviations of the measurements.

Figure 8 shows that the point-to-point performance of the fast Ethernet remains relatively unaffected by the message traffic of checkpointing. Since the network is switched, the two testing processors communicate without interference from the checkpointing processors. This does not say that the network is unaffected. Were we to test the communication performance of more pairs of processors, we expect that MIR, PAR and RS checkpointing would have more of a negative effect.

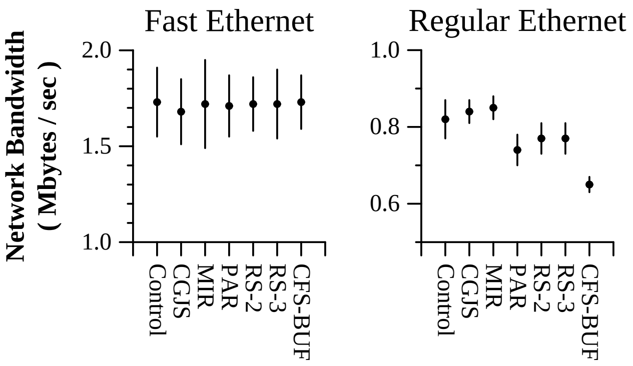

Figure 8. Network bandwidth during checkpointing

The regular Ethernet is affected adversely by the PAR, RS, and CFS-BUF strategies. This is due to writing checkpoint files to the central file server. PAR shows less of an effect than CFS-BUF because it interleaves the calculation of the parity checkpoint with the writing of than checkpoint to disk. Thus, it leaves

Transient Failures	Frequent Failures	Disk Space Is a Concern	Minimize Impact on Shared Resources	One Failure	Multiple Failures	Best Strategy
Yes	Either	Either	Either	Either	Either	CGJS (CGJS-COW)
No	Either	No	Either	Yes	No	MIR (MIR-COW)
No	Either	Yes	Either	Yes	No	PAR (PAR-COW)
No	No	Either	No	No	Yes	CFS-BUF (CFS-BUF-COW)
No	Yes	Either	Either	No	Yes	RS (RS-COW)
No	Either	Either	Yes	No	Yes	RS (RS-COW)

Table 3. Recommended checkpointing strategies for different performance parameters

the Ethernet idle more. The RS tests show even less of an effect that PAR because they perform more calculations per checkpointed byte than in the PAR test.

7. Related Work

As stated in section 2, coordinated checkpointing and checkpoint consistency are well-studied problems. The paper by Elnozahy, Johnson and Wang puts most of this work in perspective and provides a good starting point for studying these areas of research [7].

Combining local and global checkpoints to trade off reliability and performance has been studied by Vaidya [34]. This paper provides an excellent motivation for implementing single-site fault-tolerance in coordinated checkpointing systems as an alternative to CFS checkpointing. Vaidya's method for providing single-site fault-tolerance is the MIR strategy.

Checkpointing schemes that rely solely on checkpoints and encodings in memory have been developed and discussed by several researchers [6, 26, 29]. These schemes provide resilience to single processor failures with low overhead. The limitation of these schemes is their dependence on good error detection. In order to use the in-memory checkpoints following a processor failure, the remaining processors must be able to recognize the failure, agree on it, and then restore themselves from their state in memory. If any of these steps fail, the in-memory checkpoints become useless. It has been our experience that many message-passing libraries (PVM, for example) tolerate "clean" failures (e.g. system shutdowns) very well, but tend to hang or abort when unexpected failures occur. In such cases, periodic checkpointing to disk provides an alternative that is easier to program, and is more reliable than saving encodings in memory. As message-passing libraries evolve to perform better error detection, in-memory checkpointing techniques will become more useful.

This paper uses Reed-Solomon coding to tolerate multiple processor failures. Although this is the best general-purpose method for tolerating any number of failures, there are better methods for specific numbers. For example, EVENODD parity [2] is a method for tolerating two processor failures using only parity operations. As such, it is faster than using Reed-Solomon coding for two processor failures. EVENODD parity was not used for this experiment, but should be used in preference to Reed-Solomon coding for two processor fault-tolerance.

Finally, there are other techniques that have been presented and implemented to reduce checkpoint overhead and latency, including incremental checkpointing [10, 38], memory exclusion [25], compression [27] and compiler assistance [17, 24]. These techniques should affect all checkpointing strategies equally, and will not alter any of the conclusions in this paper.

8. Conclusion

This paper has explored several strategies for taking coordinated checkpoints on a network of workstations. While current tools for coordinated checkpointing implement simple strategies (CFS and CGJS), an application of RAID techniques can yield better performance for a smaller amount of fault coverage. This is especially true when checkpoint latency, recovery time, and impact on shared resources are major concerns.

Based on the results of this paper, we make the recommendations in Table 3 concerning which checkpointing strategy is best for various performance criteria. It it our hope that future implementations of coordinated checkpointing tools will include all the strategies mentioned in Table 3 in order to give the user maximum flexibility in tuning the performance of checkpointing to his or her specific network.

9. Acknowledgment

The author thanks the following people for their help with this research and this paper: Randy Bond, Mootaz Elnozahy, W. Kent Fuchs, Chris Jepeway, Mark Jones, Youngbae Kim, Kai Li, Chad Mynhier, Michael Puening and the anonymous referees.

References

[1] A. Bauch, B. Bieker, and E. Maehle. Backward error recovery in the dynamical reconfigurable multiprocessor system DAMP. In *Proc. of the 1992 Workshop on Fault-Tolerant Par. and Dist. Sys.*, pages 36–43, Amherst, 1992.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual Int. Symp. on Comp. Arch.*, pages 245—254, Chicago, IL, April 1994.

[3] 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.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comp. Sys.*, 3(1):3–75, February 1985.

[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*, 1994.

[6] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *26th Int. Symp. on Fault-Tolerant Comp.*, Sendai, June 1996.

[7] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. To appear, 1996.

[8] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symp. on Rel. Dist. Sys.*, pages 39–47, October 1992.

[9] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *24th Int. Symp. on Fault-Tolerant Comp.*, pages 298–307, Austin, TX, June 1994.

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

[11] 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.

[12] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *12th Symp. on Rel. Dist. Sys.*, October 1993.

[13] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[14] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Soft. Eng.*, SE-13(1):23–31, January 1987.

[15] T. H. Lai and T. H. Yang. On distributed snapshots. *Inf. Proc. Letters*, 25:153–158, May 1987.

[16] J. León, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Tech. Rep. CMU-CS-93-124, Carnegie Mellon Univ., February 1993.

[17] C-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-assisted full checkpointing. *Soft. – Practice and Exp.*, 24(10):871–886, October 1994.

[18] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. on Par. and Dist. Sys.*, 5(8):874–879, August 1994.

[19] D. D. E. Long, J. L. Carroll, and C. J. Park. A study of the reliability of internet sites. In *10th Symp. on Rel. Dist. Sys.*, pages 177–186, October 1991.

[20] Message Passing Interface Forum. MPI: A message-passing interface standard. *Int. Journal of Supercomputer App.*, 8(3/4), 1994.

[21] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. on Soft. Eng.*, 18(4):319–328, April 1992.

[22] J. S. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton Univ., January 1993.

[23] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Tech. Rep. UT-CS-96-332, Univ. of Tenn., July 1996.

[24] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Tech. Comm. on Op. Sys. and App. Env.*, 7(4):10–14, Winter 1995.

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

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

[27] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Par. & Dist. Tech.*, 2(2):62–67, Summer 1994.

[28] T. J. E. Schwarz and W. A. Burkhard. RAID organization and performance. In *Proc. of the 12th Int. Conf. on Dist. Comp. Sys.*, pages 318–325, Yokohama, June 1992.

[29] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scal. High Perf. Comp. Conf.*, pages 694–701, Knoxville, TN, May 1994.

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

[31] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th Int. Par. Proc. Symp.*, April 1996.

[32] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. on Comp. Sys.*, 3(3):204–226, August 1985.

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

[34] N. H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Conf. on Meas. and Mod. of Comp. Sys.*, Ottawa, May 1995.

[35] N. H. Vaidya. On checkpoint latency. In *Pac. Rim Int. Symp. on Fault-Tolerant Sys.*, Newport Beach, December 1995.

[36] S. Venkatesan. Message-optimal incremental snapshots. In *Proc. of The Ninth Int. Conf. on Dist. Comp. Sys.*, pages 53–60, Newport Beach, June 1989.

[37] Y. M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *11th Symp. on Rel. Dist. Sys.*, pages 147–154, October 1992.

[38] P. R. Wilson and T. G Moher. Demonic memory for process histories. In *SIGPLAN '89 Conf. on Prog. Lang. Design and Imp.*, pages 330–343, June 1989.

[39] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symp. on Par. and Dist. Proc.*, Dallas, TX, December 1993.