Faster Checkpointing with $N+1$ Parity

James S. Plank

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

Kai Li

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

Appearing in:

24th Annual International Symposium on Fault-Tolerant Computing
Austin, Texas
June 15-17, 1994

Faster Checkpointing with $N + 1$ Parity

James S. Plank

Department of Computer Science
University of Tennessee
Knoxville, TN 37920

Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

This paper presents a way to perform fast, incremental checkpointing of multicomputers and distributed systems by using $N + 1$ parity. A basic algorithm is described that uses two extra processors for checkpointing and enables the system to tolerate any single processor failure. The algorithm's speed comes from a combination of $N + 1$ parity, extra physical memory, and virtual memory hardware so that checkpoints need not be written to disk. This eliminates the most time-consuming portion of checkpointing.

The algorithm requires each application processor to allocate a fixed amount of extra memory for checkpointing. This amount may be set statically by the programmer, and need not be equal to the size of the processor's writable address space. This alleviates a major restriction of previous checkpointing algorithms using $N + 1$ parity [28].

Finally, we outline how to extend our algorithm to tolerate any m processor failures with the addition of $2m$ extra checkpointing processors.

1 Introduction

Checkpointing is an important topic in computer science as it is the only way to provide fault tolerance in a general-purpose computing environment [1]. With the proliferation of large parallel and distributed systems, checkpointing has been the method of choice for providing fault-tolerance [3, 5, 10, 14, 15, 27]. Checkpointing typically requires the saving of one or more processors' address spaces to stable storage so that after a failure, the machine's state may be restored to the saved checkpoint. Besides fault-tolerance, checkpointing has been used for process migration, job swapping and debugging.

The major overhead of checkpointing is writing the checkpoint to disk. Results of implementations have shown that the overriding concern in making checkpoints fast is either reducing or hiding the overhead of disk writing. This is especially a concern in parallel and distributed systems, where the number of processors is often vastly larger than the number of disks. Proposed solutions to reducing the effect of disk writing have been to use incremental checkpointing [9, 11, 33], compiler support [20], compression [20, 27], copy-on-write [21], non-volatile RAM [15], and pre-copying [9]. Although these methods succeed to varying degrees, they all default to the speed of the storage medium as the bottleneck in decreasing overhead.

In this paper, we present a set of incremental checkpointing algorithms that perform *no writing to disk*. Instead, they assume that no more than m processors fail in a parallel or distributed system at any one time, and describe how to recover from such failures. We will start with a detailed description of the algorithm when m equals one, and then describe how it can be modified for larger values of m. The bottom line is that with $2m$ extra processors, we can protect the system from any m processors failing.

The algorithm revolves around $N + 1$ parity, previously used by Gibson [12] to provide reliability in disk arrays. $N + 1$ parity was proposed by Plank [28] as a way to perform diskless checkpointing, but the proposed algorithm is non-incremental, and needs each processor to maintain two in-memory copies of local checkpoints. This forces each processor to allocate two thirds of its physical memory for the sole use of checkpointing, which is unreasonable.

The algorithm presented here alleviates this problem by using incremental checkpointing: Extra space is required only for the portions of each processor's memory that have changed since the previous checkpoint. We allow the user to specify an upper limit on this space and when it is consumed, a new checkpoint must be taken. This introduces a tradeoff between the extra memory required for checkpointing and the

number of checkpoints that must be taken. We evaluate this tradeoff in detail.

By omitting disk-writing from the checkpointing protocol, programmers should be able to checkpoint far more frequently than when they have to write to disk. Instead of checkpointing on the order of once an hour, programmers may checkpoint as frequently as once every second, or every few seconds. This should drastically reduce the amount of lost work due to processor failures.

Moreover, this algorithm allows one's computational model to be one of a continuously running parallel system. If a processor, or up to m processors fail, then they can be replaced instantly with any available processor or processors. The system does not have to be halted and restarted when the failing processors are reset. Moreover, the amount of work lost due to the failures is small – on the order of seconds. Thus, the algorithm may be used for process migration and/or load-balancing in a reconfigurable distributed system such as PVM [2]. Finally, as there is no reliance on disk, there are no problems concerning the availability of stable storage following a failure or migration.

Combined with an all-encompassing checkpointing method for wholesale system failures [10, 14, 18, 22, 28, 31], this algorithm provides an efficient composite system for fault-tolerant computing: The faster algorithm is used to checkpoint at small intervals, like once a second, and the all-encompassing method is used to checkpoint at large intervals, like once an hour. Thus, the more frequent case of a few processors failing (or being reclaimed by their owners in a distributed workstation environment) is dealt with swiftly, involving no disk interaction, and a minimal loss of computation. The rarer case of the whole system failing is handled as well, albeit more slowly, as it has more saved state from which to recover.

2 The Basic Algorithm

To describe the basic algorithm, we assume to have a collection of $n + 2$ processors: $p_1, \ldots, p_n$, p_c and p_b. Processors p_c and p_b are called the "checkpoint processor" and "backup processor" respectively. Both are dedicated solely to checkpointing. Processors $p_1, \ldots, p_n$ are free to execute applications, and are thus called "application processors." The application processors must reserve a fixed amount of memory for checkpointing. We denote this amount by M. Finally, we assume that the checkpointing mechanism is able to access the memory management unit (MMU) of each processor, enabling it to protect pages of memory as *read-only* or *read-write*, and to catch the resulting page faults.

The basic idea of the algorithm is as follows: At all points in time, there will be a valid consistent checkpoint maintained by the system in memory. Consistent checkpointing has been well-documented and well-studied [6, 8, 17, 18, 22]. A consistent checkpoint is comprised of a local checkpoint for each application processor, and a log of messages. To recover from a consistent checkpoint, each processor restores its execution to the state of its local checkpoint, and then messages are re-sent from the message log. For the sake of simplicity, we assume that the consistent checkpoint has no message state. For example, the processors can use the "Sync-and-Stop" checkpointing protocol [28] to guarantee no message state.

The consistent checkpoint is maintained cooperatively by all processors, $p_1, \ldots, p_n, p_c, p_b$, using $N + 1$ parity [12]. Specifically, each application processor will have a copy of its own local checkpoint in physical memory. The checkpoint processor will have a copy of the "parity checkpoint," which is defined as follows:

- Let the size of each application processor p_i's checkpoint be S_i.
- The checkpoint processor records each value of S_i, for $1 \leq i \leq n$.
- The size S_c of the parity checkpoint is the maximum S_i for $1 \leq i \leq n$.
- Let $b_{i,j}$ be the j-th byte of of p_i's checkpoint if $j \leq S_i$, and 0 otherwise.
- Each byte $b_{c,j}$ of the parity checkpoint is equal to the bitwise exclusive or ($\oplus$) of the other bytes: $b_{c,j} = b_{1,j} \oplus b_{2,j} \oplus \ldots \oplus b_{n,j}$, for $1 \leq j \leq S_c$.

The backup processor is used to keep a copy of the parity checkpoint when the checkpoint processor needs to update its copy.

Now, if any application processor p_i fails, then the system can be recovered to the state of the consistent checkpoint by having each non-failed processor restore its state to its local checkpoint, and by having the failed processor calculate its checkpoint from all the other checkpoints, and from the parity checkpoint. Specifically, it retrieves its value of S_i from the checkpoint processor (or from the backup processor if the checkpoint processor is changing its state). Then it calculates its checkpoint:

$$b_{i,j} = b_{1,j} \oplus \ldots \oplus b_{i-1,j} \oplus b_{i+1,j} \oplus \ldots \oplus b_{n,j} \oplus b_{c,j}, \quad \text{for } 1 \leq j \leq S_i$$

If the checkpoint processor fails, then it restores its state from the backup processor, or by recalculating the parity checkpoint from scratch. The backup processor may be restored similarly.

The actual algorithm works as follows: At the beginning of each application processor's execution, it takes checkpoint 0: It sends the size of its application's writable address space to the checkpoint processor, along with the contents of this space. Next, it protects all of its pages as *read-only*. The checkpoint processor records each value of S_i, and calculates the parity checkpoint from the contents of each processor's address space. When the checkpoint processor finishes calculating the parity checkpoint, it sends a copy to the backup processor, which stores it.

After sending p_c its address space, each application processor clears its M bytes of extra memory. This space is split in half, and each half is used as a checkpointing buffer. We will call them the *primary* and *secondary* checkpointing buffers. After designating the checkpointing buffers, the processor is free to start executing its application. When the application generates a page-fault by attempting to write a *read-only* page, the processor catches the fault, and copies the page to its primary checkpointing buffer. It then resets the page's protection to *read-write*, and returns from the fault.

If any processor fails during this time, the system may be restored to the most recent checkpoint. Each application processor's checkpoint consists of the *read-only* pages in its writable address space, and the pages in its primary checkpointing buffer. The processor can restore this checkpoint by copying (or mapping) the pages back from the buffer, reprotecting them as *read-only*, and then restarting. Obviously, if the checkpoint processor fails during this time, it can be restored from the backup processor, and if the backup processor fails, then it can be restored from the checkpoint processor.

Now, when any processor uses up all of its primary checkpointing buffer, then it must start a new global checkpoint. In other words, if the last completed checkpoint was checkpoint number c, then it starts checkpoint $c+1$. The processor performs any coordination required to make sure that the new checkpoint is consistent, and then takes its local checkpoint. To take the local checkpoint, it must do the following for each *read-write* protected page $page_k$ in its address space:

- Calculate $diff_k = page_k \oplus buf_k$, where buf_k is the saved copy of $page_k$ in the processor's primary checkpointing buffer.
- Send $diff_k$ to the checkpoint processor, which `XOR`'s it with its own copy of $page_k$. This has the effect of subtracting buf_k from the parity page and adding $page_k$.
- Set the protection of $page_k$ to be *read-only*.

After sending all the pages, the processor swaps the identity of its primary and secondary checkpointing buffers.

If an application processor fails during this period, the system can still restore itself to checkpoint c. First consider a non-failed application processor that has not started checkpoint $c+1$. It restores itself as described above, by copying or mapping all pages back from its primary checkpointing buffer, resetting the pages to *read-only*, and restarting the processor from this checkpoint. Suppose instead that the application processor has started checkpoint $c+1$. Then, it first restores itself to the state of local checkpoint $c+1$ by copying or mapping pages from the primary checkpointing buffer, and next, it restores itself to the state of checkpoint c by copying or mapping pages from the secondary checkpointing buffer. When all these pages are restored, then the processor's state is that of checkpoint c. The checkpoint processor restores itself to checkpoint c by copying the parity checkpoint from the backup processor. The backup processor does nothing. Once all non-failed processors have restored themselves, the failed processor can rebuild its state, and the system can continue from checkpoint c.

If the checkpoint processor fails during this period, then the application processors roll back to the state of checkpoint c, and the checkpoint processor restores itself from the backup processor. If the backup processor fails, then the processors roll back to the state of checkpoint c, and the checkpoint processor's checkpoint is recalculated, and then copied to the backup.

When all processors have finished taking their local checkpoints for global checkpoint $c+1$, the checkpoint processor sends a copy of its checkpoint to the backup processor, and the application processors may jettison their secondary checkpointing buffers.

3 An Example

In this section, we present an example of a six-processor system running this checkpointing algorithm. Processors $P_1, \ldots, P_4$ are the application processors. Processor P_5 is the checkpoint processor, and P_6 is the backup processor. Before starting the application, the processors take checkpoint 0: They protect their writable address spaces to be *read-only*, clear

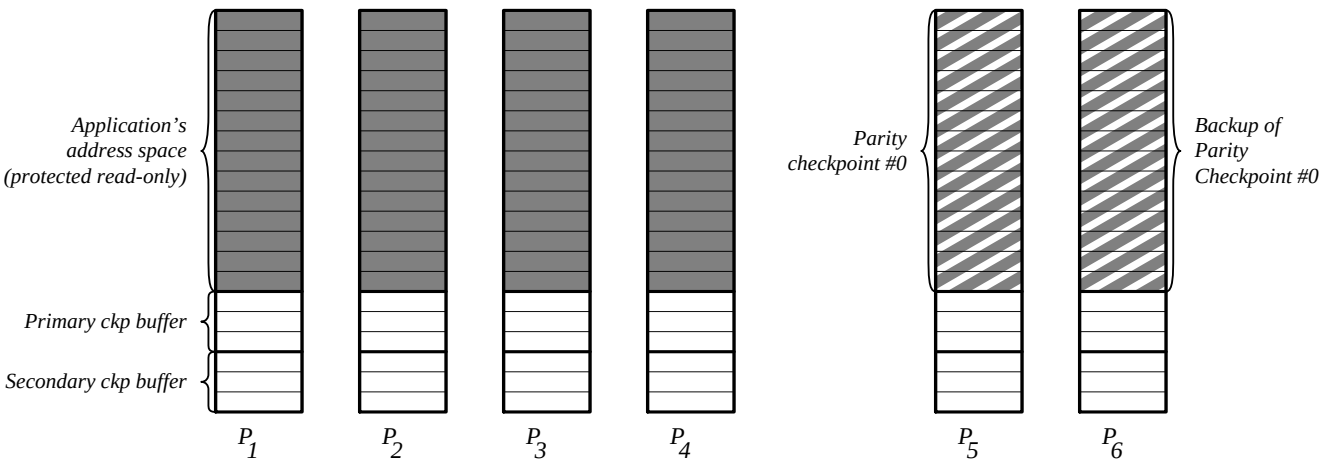

Figure 1: State at checkpoint 0.

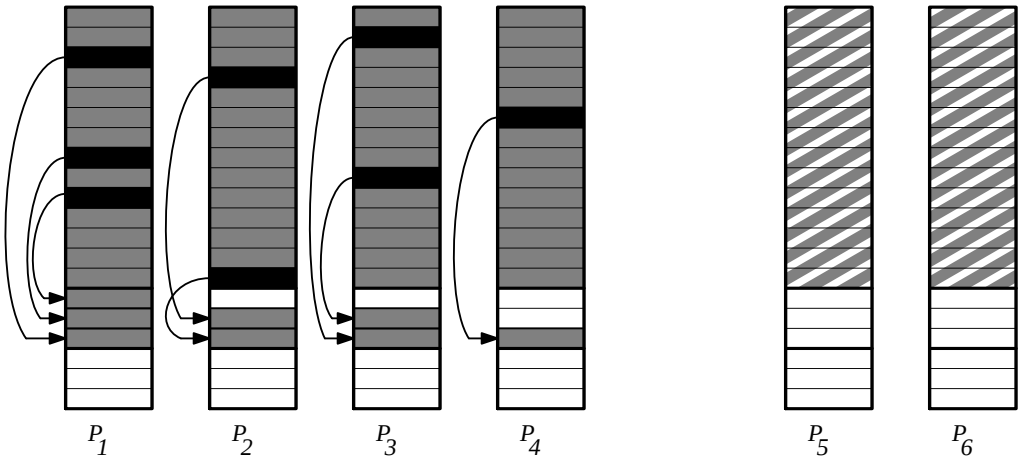

Figure 2: State slightly after checkpoint 0.

their checkpointing buffers, and send the contents of their address spaces to P_5. P_5 calculates the parity checkpoint, and then sends it to the backup processor, P_6. At this point, the system looks like Figure 1.

Next, the application processors run the application. When page faults occur, the faulting pages are copied to the processor's primary checkpointing buffer and set to *read-write*, so that the application may continue. The state of the system looks as in Figure 2. Processor P_1 has copied three pages to its primary checkpointing buffer. Processors P_2 and P_3 have copied two pages each, and P_4 has copied one. Were a failure occur to one of the application processors, then the others would restore themselves to checkpoint 0 by copying or mapping the pages back from the primary checkpointing buffer to the application's memory and reprotecting those pages as *read-only*. The failed processor may then reconstruct its checkpoint from the other application processors' checkpoints and from the parity checkpoint. If a non-application processor fails, then it may restore itself from the other non-application processor.

Since processor P_1 has used up its primary checkpointing buffer, checkpoint 1 must be started. P_1 goes through any synchronization necessary for the checkpoint to be consistent. When it is time to take its local checkpoint, P_1 `XOR`'s each changed page with its buffered copy and sends the results to P_5, which uses them to update the parity checkpoint. P_1 then protects its pages to be *read-only*, and swaps the identity of the primary and secondary checkpoint buffers. The state of the system is depicted in Figure 3.

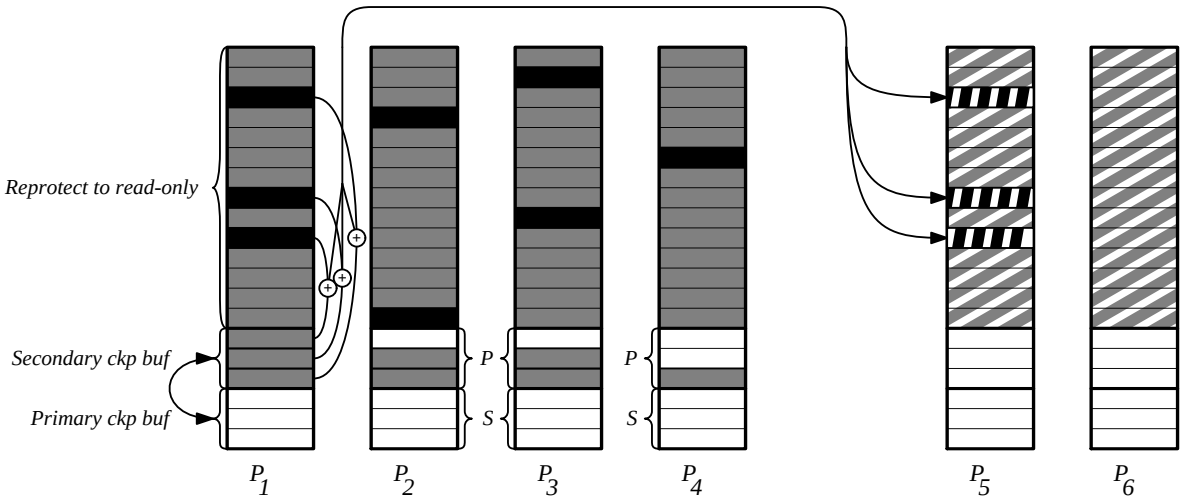

Figure 3: Processor P_1 starts checkpoint 1.

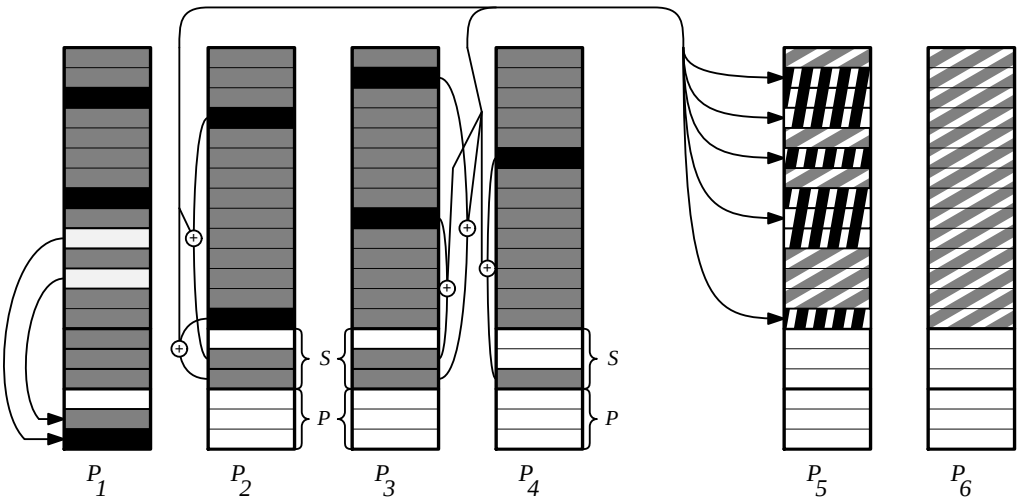

Figure 4: Processors P_2, P_3 and P_4 take checkpoint 1.

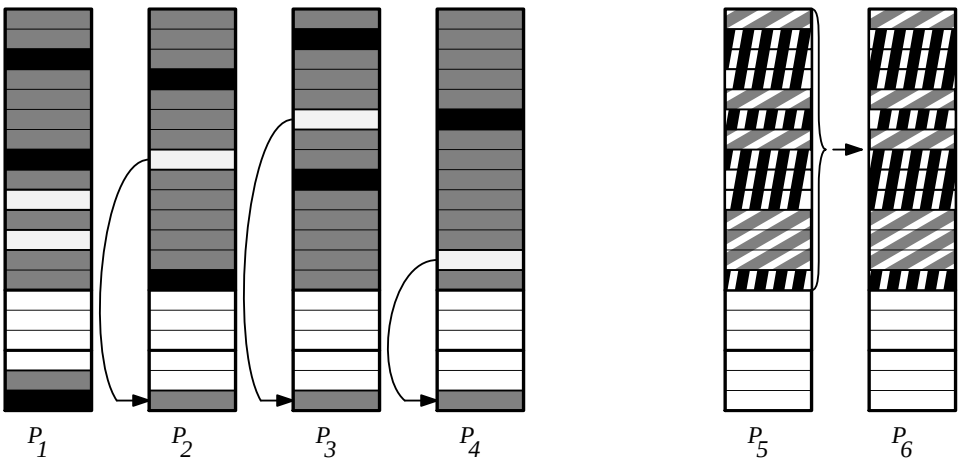

Figure 5: Checkpoint 1 is complete.

If an application processor fails at this point, then the processors may again roll back to checkpoint 0. P_1 is able to do this by using pages from its secondary checkpoint buffer. P_2, P_3 and P_4 use pages from their primary checkpoint buffer as before. The checkpoint in P_6 must be used, as P_5's checkpoint has been updated to reflect P_1's changes for checkpoint 1. If P_5 fails, then it copies its checkpoint from P_6, and the application processors roll back to checkpoint 0. If P_6 fails, then the processors again roll back to checkpoint 0, and the parity and backup checkpoints are calculated anew.

Figure 4 shows processors P_2, P_3 and P_4 taking their local checkpoints. They `XOR` their changed pages with the buffered copies and send the results to P_5. Then, they reprotect the pages and swap the identities of the primary and secondary checkpoint buffers. If a failure occurs during these activities, then the recovery is the same as in Figure 3: The processors still recover to checkpoint 0. Also during this time, processor P_1's application continues execution, and its pages are copied to the new primary checkpoint buffer. To restore itself to the state of checkpoint 0, it must copy or map pages first from the primary checkpoint buffer, and then from the secondary checkpoint buffer. As before, the parity checkpoint in the backup processor (P_6) must be used.

Finally, Figure 5 depicts the state when all the local checkpoints are finished: The parity checkpoint in processor P_5 is copied to processor P_6, and the application processors jettison their secondary checkpointing buffers. Any failure will now be restored to checkpoint 1.

4 Tolerating Failures of More Than One Processor

The above algorithm allows the system to tolerate any one processor failure with two extra checkpointing processors. In this section, we outline how to configure the system to tolerate any m processor failures with $2m$ extra checkpointing processors. Specifically, let there be $n + 2m$ processors in the system. As before, processors $p_1, \ldots, p_n$ are the application processors. The rest are split into checkpointing and backup processors: $p_{c1}, \ldots, p_{cm}$, and $p_{b1}, \ldots, p_{bm}$. The checkpointing and backup processors are paired up (p_{ci} is paired with p_{bi}), and related like the checkpoint and backup processors in the previous section: The backup processor p_{bi} contains the contents of p_{ci} at the time of the most recently committed checkpoint. This is so that there is a copy of p_{ci} from which to restore if a failure occurs while p_{ci} is being updated.

The application processors perform the same actions as in the above algorithm, with one difference: Instead of sending copies of their changed pages to just the one checkpoint processor, they send their changed pages to all m checkpoint processors. The checkpoint processors are like the original checkpoint processor above, except that they do not just calculate the bitwise parity of each page. Instead, each one calculates a different function of the bytes of the pages. This calculation is such that if any m processors in the entire system fail, the rest may be used to recalculate the values of the failed ones. The description of how each checkpoint processor makes its calculation requires too much detail for this paper. Instead, it may be found in [26]. We outline it in the following paragraph:

The calculations resemble Reed-Solomon codes [24, 32]: Instead of performing bitwise arithmetic as the checkpoint processor does in the algorithm of the previous sections, each processor breaks the pages into multi-bit words, and performs arithmetic on those words over a Galois Field. The number of bits per word depends on the size of n and m. Although more complex computationally than $N+1$ parity, this coding is not prohibitively complex: Instead of an exclusive-or for each byte, each processor must perform a few table lookups and some bitwise arithmetic. Recovery involves gaussian elimination of an $n \times n$ matrix, and then for each byte, more table lookups and bitwise arithmetic. Again, complete details may be found in [26]. Since each processor is devoted solely to checkpointing, it is well-situated to perform the computation for checkpointing and recovery, and the entire process should still be faster than checkpointing to disk.

5 Discussion

There are two types of overhead that the basic algorithm imposes on user programs. First is the time overhead of taking checkpoints, and second are the extra memory requirements, as manifested by the variable M.

The time overhead of checkpointing has the following components:

- Processing page faults.
- Coordinating checkpoints for consistency.
- Calculating each $diff_k$.

- Sending $diff_k$ to the checkpoint processor.
- The frequency of checkpointing.

We do not analyze the first two components as they are they same for this algorithm as for other incremental and consistent checkpointing algorithms [9, 27]. They should not amount to as much overhead as the third and fourth components. These components, the time to calculate each $diff_k$ and send it to the checkpoint processor, depend on the speed of the processor, the speed of the interconnection network, and the number of bytes sent.

We notice that the sum of these components may be improved by a simple optimization, stemming from the fact that each processor sends $diff_k = page_k \oplus buf_k$ to the parity processor. This is as opposed to normal incremental checkpointing algorithms [9, 11, 33] that send $page_k$ to stable storage during an incremental checkpoint. The benefit of sending $diff_k$ is that all bytes of $page_k$ which have not been changed since the previous checkpoint will be zero in $diff_k$. This allows us to optimize the algorithm by sending only the non-zero bytes of $diff_k$, thereby lowering the number of bytes sent to the checkpoint processor when only fractions of pages are altered. This technique – sending the `diff` of the changed pages – should be a marked improvement over blindly sending $diff_k$ when only a few bytes of a page are touched in a checkpointing interval.

The frequency of checkpointing is related to the extra memory requirements, and thus the two are discussed together. As stated in the Introduction, there is a tradeoff between the extra memory requirements and the frequency of checkpointing. At one extreme, if the processors can allocate enough extra memory for two whole checkpoints, as suggested in [28], then the primary checkpointing buffer will never become completely full, and thus the processors can checkpoint at the direction of the user – any checkpointing interval can be supported. At the other extreme, if the processors can allocate only a minimum of extra memory, such as five pages, for checkpointing, then the buffers will be filling up constantly, forcing checkpoints to be taken at very small intervals, and grinding the system to a halt.

Obviously, the ideal place between these extremes depends on the processors and on the program being executed. If the program exhibits great locality, then a small checkpointing buffer will be sufficient to keep the system from bogging down. If the processors have a large amount of available physical memory, then a large checkpointing buffer may be feasible. For programs whose locality is bad enough to constantly fill up the maximum size checkpointing buffers available on the application processors, this method may impose too much overhead to be reasonable.

One goal of our future work is to make an empirical evaluation of how this checkpointing performs on a variety of programs, networks, and processor characteristics. To provide a more limited evaluation we draw from two sources. First are previous results from Elnozahy, Johnson and Zwaenepoel [9]. They implemented incremental, consistent checkpointing to a central file server in a distributed system of 16 processors. Although they checkpointed at a coarse interval of two minutes, in six out of their eight test programs, incremental checkpoints consisted of less than half of the application's memory. This leads us to believe that at far finer checkpointing intervals (on the order of every second or every few seconds), the extra memory requirements (i.e. M) should be much smaller.

Second, we have instrumented some distributed programs in PVM [2] to record their behavior under this algorithm for varying values of M. The results in Figure 6 through Figure 8 display the instrumentation of multiplying two 1300x1300 matrices using eight DEC alpha workstations (six for the application, one to record the parity checkpoint, and one to backup the parity processor) connected via an Ethernet. Each application processor uses 7.2 megabytes of memory without checkpointing. The processors have 8 kilobyte pages and well over 24 megabytes of physical memory (thus space for checkpointing buffers is not a problem).

The data in Figure 6 show that matrix multiplication is a good candidate for checkpointing using this method. Its locality is such that with even a tiny buffer size of 160K (this is just ten pages for each of the primary and secondary checkpointing buffers), the overall running time of the program is increased by an average of only 40 seconds. This is under 10% overhead.

The fluctuations in running time seen by the maximum and minimum values for each point in Figure 6 are due to the fact that these tests were run using general-purpose machines in Princeton's Computer Science Department. Although the tests were run at night, the machines and network could not be allocated for exclusive testing. Thus, network and machine contention from other users has made its way into the graph.

Figure 7 displays the average number of checkpoints taken for the given checkpoint buffer size, and then average number of CPU seconds between checkpoints. As CPU seconds are not affected drastically by ex-

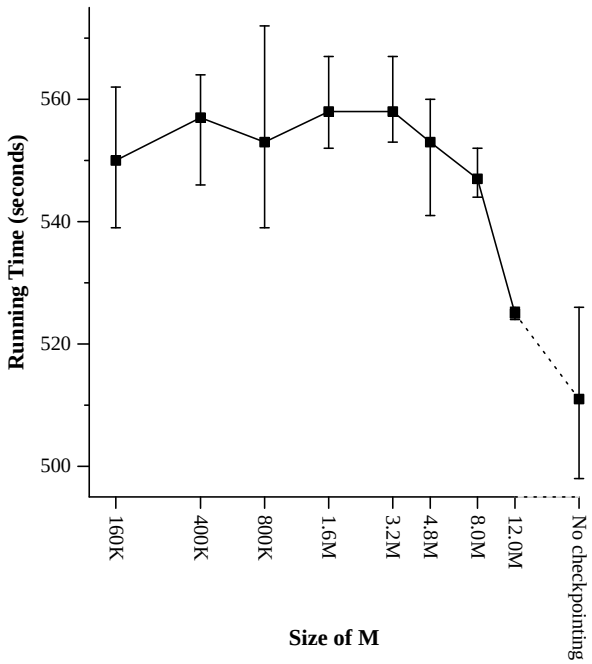

Figure 6: Running Time of Checkpointing

ternal processes, this data factors out the effects of the shared machines. Figure 7 corroborates Figure 6, showing that all checkpoint buffer sizes give reasonable inter-checkpoint intervals. Given this data, a user might choose 800K to be the size of M, as that necessitates a checkpoint for every four seconds of CPU time.

Figure 8 shows the success of compressing each $diff_k$ so that only the non-zero bytes are transmitted. As one would expect, as the checkpoint buffers get bigger, pages get written more between checkpoints, resulting in fewer zero bytes in $diff_k$, and thus less compressibility. Still, the compression ratios observed

M (bytes)	Number of Checkpoints	CPU Seconds Between Checkpoints
160K	763	0.48
400K	180	2.00
800K	82	4.33
1.6M	39	8.98
3.2M	17	19.96
4.8M	10	31.28
8.0M	3	80.11
12.0M	0	-

Figure 7: Number of Checkpoints

M (bytes)	Checkpoint Size (bytes)	Compressed Checkpoint Size (bytes)	Compression Ratio (%)
160K	80K	19K	76
400K	200K	78K	61
800K	400K	172K	57
1.6M	800K	360K	55
3.2M	1.6M	816K	49
4.8M	2.4M	1.3M	45
8.0M	4.0M	2.3M	42

Figure 8: Average Size of Checkpoints

show significant savings in the amount of bytes transmitted.

One might observe that if each processor is producing 172K checkpoints every four seconds, as in this example, then a local disk should be able to write the incremental checkpoints as fast as they are produced. This is true, and is mentioned in the Future Work below. However, in a system like the one tested, where all disks belong to a central file server and are shared among hundreds of users, disk contention is a significant issue, and checkpointing to processors will be more efficient.

6 Related Work

Checkpointing is a well-documented topic in fault-tolerance. In parallel and distributed systems, the field has been divided into pessimistic [4, 5, 29], optimistic [14, 31], and consistent checkpointing [6, 7, 8, 15, 17, 18, 22, 30]. Implementations have been tested on uniprocessors [11, 20, 23], multiprocessors [19, 21], distributed systems [9, 15], and multicomputers [27]. All of these systems have checkpointed to disk, and consequently taken efforts to minimize the overhead caused by disk writes.

Johnson and Zwaenepoel presented an algorithm to reconstruct the message state of a distributed system when at most one processor fails, with no disk-writing [13]. The algorithm has the sending processor save the message so that it may resend if the receiver fails. Processors save their own execution states in disk checkpoints.

Keleher, Cox and Zwaenepoel used `diff`'s to propagate updates to shared pages in their distributed shared memory system "Munin" [16]. As in this paper, the `diff`'s are used to lower the latency of transporting whole pages by sending fewer than a pageful

of bytes when possible.

$N+1$ parity was used to provide single-site fault-tolerance by Gibson in his design and implementation of RAID disk arrays [12]. Gibson also addresses multiple-site failures, and gives an algorithm for tolerating 2-site failures with $2n^{\frac{1}{2}}$ extra disks. This algorithm scales to tolerate m-site failures with $mn^{\frac{m-1}{m}}$ extra disks. Reed-Solomon codes were not employed to reduce the number of extra disks to m because of the extra complexity that such codes require in the disk controller hardware. This is not a problem in our system because the unit of fault-tolerance is a processor, well-capable of handling the extra calculations. RAID techniques were also combined with memory management in a design of disk arrays to support transaction recovery in [25].

7 Conclusion

We have presented a fast incremental checkpointing algorithm for distributed memory programming environments and multicomputers. This algorithm is unique as it checkpoints the entire system without using any stable storage. Therefore, its speed is not limited by the speed of secondary storage devices.

The basic algorithm presented above tolerates the failure of any one processor with the addition of two checkpointing processors. This algorithm generalizes so that any m processor failures can be tolerated with the addition of $2m$ checkpointing processors.

Of concern in this algorithm is the amount of extra memory per processor required for checkpointing, and how it relates to the frequency of checkpointing. In the discussion above, we argue that a fixed amount of extra memory is reasonable for many applications. Results from Elnozahy, Johnson and Zwaenepoel [9], as well as our own instrumentation of a distributed matrix multiply corroborate this claim.

8 Future Work

For future work, we are continuing to assess the performance of this algorithm on more test programs. The result of this work should be to quantify what values of M are reasonable for programs of differing locality patterns.

This work is being extended in three different directions. First is to write a full-blown checkpointing implementation on PVM using a combination of this method and an all-encompassing checkpointing method to disk. The goal is to provide fast fault-tolerance and process migration (for load balancing) using this method, and still be able to tolerate wholesale system failures. Second, we are working with Rob Netzer and Jain Xu to exploring whether or not we can improve the performance of incremental checkpointing (to disk) by using a checkpoint buffer and compressing `diff`'s as specified in this paper. Finally, we are working with Jack Dongarra and Youngbae Kim to explore how diskless checkpointing can be mixed with application-oriented checkpointing to achieve fault-tolerance without the reliance on page-protection hardware. The goal in all of these project is assess the overhead of checkpointing with methods described in this paper, as compared to standard checkpointing implementations to stable storage [9, 20, 21, 27].

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[2] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. Heterogeneous network computing. In *Sixth SIAM Conference on Parallel Processing*, 1993.

[3] K. P. Birman. ISIS: A system for fault-tolerant distributed computing. Technical Report TR 86-744, Cornell University, April 1986.

[4] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, GA, October 1983.

[5] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):3–75, February 1985.

[7] F. Cristian and F. Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.

[8] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report TR 90-1101, Cornell University, February 1990.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In

Proceedings of the 11th Symposium on Reliable Distributed Systems, 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 Special Issue on Fault-Tolerant Computing*, 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, Jan 1989.

[12] Garth Alan Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California, Berkeley, December 1990.

[13] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

[14] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.

[15] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. Technical Report IR-258, Vrije Universiteit, Amsterdam, October 1991.

[16] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[17] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[18] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[19] L. A. Laranjeira, M. Malek, and R. Jenevein. Experimental evaluation of techniques for fault-tolerance. Technical Report TR-92-32, Department of Computer Sciences, University of Texas at Austin, July 1992.

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

[21] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, Washington, Mar 1990.

[22] K. Li, J. F. Naughton, and J. S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Processing*, 20(3), June 1992.

[23] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Conference Proceedings, Usenix Winter 1992 Technical Conference*, pages 283–290, January 1992.

[24] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[25] A. N. Mourad, W. K. Fuchs, and D. G. Saab. Database recovery using redundant disk arrays. *Journal of Parallel and Distributed Computing*, January 1993.

[26] J. S. Plank, J. Friedman, and K. Li. A failure correction technique for parallel storage devices with minimal device overhead. Submitted. Contact `plank@cs.utk.edu` for a copy, 1994.

[27] J. S. Plank and K. Li. Performance results of *ickp* — a consistent checkpointer on the iPSC/860. Submitted to *Scalable High Performance Computing Conference*, 1994.

[28] James S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, January 1993.

[29] M. L. Powell and D. L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100–109, October 1983.

[30] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of The Sixth International Conference on Distributed Computing Systems*, pages 382–388, Cambridge, Massachusetts, May 1986. IEEE Computer Society.

[31] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[32] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.

[33] 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, Portland, Oregon, June 1989.