Faster Checkpointing with $N + 1$ Parity

James S. Plank\textsuperscript{*} \hspace{1cm} Kai Li\textsuperscript{†}

University of Tennessee Technical Report CS-93-219
December, 1993
Submitted to FTCS-24

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 statically set 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 [Pla93].

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 [AL81]. With the proliferation of large parallel and distributed systems, checkpointing has been the method of choice for providing fault-tolerance [Bir86, BBG+89, Joh89, KMBT91, EZ92, Pla93]. 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.

\textsuperscript{*}plank@cs.utk.edu, Department of Computer Science, University of Tennessee, Knoxville TN 37996.

\textsuperscript{†}li@princeton.edu, Department of Computer Science, Princeton University, Princeton NJ 08544.
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 [FB89, EJZ92], compiler support [LF90], compression [LF90, Pla93], copy-on-write [LN90], non-volatile RAM [KMBT91], and pre-copying [EJZ92]. 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 [Gib90] to provide reliability in disk arrays. \( N + 1 \) parity was proposed by Plank [Pla93] 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 with 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. We show that this is a far more reasonable approach.

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 [BDG+93]. 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 [SY85, LY87, Joh89, LNP92, EZ92, Pla93], 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 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 \textit{read-only} or \textit{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 [CL85, KT87, LY87, CT90, LNP92]. 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 [Pla93] 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 [Gib90]. 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 (⊕) 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:

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

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 $p_i$ 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 pagel in its address space:

- Calculate \( \text{diff}_k = \text{page}_k \oplus \text{buf}_k \), where \( \text{buf}_k \) is the saved copy of \( \text{page}_k \) in the processor’s primary checkpointing buffer.

- Send \( \text{diff}_k \) to the checkpoint processor, which \( \text{XOR} \)’s it with its own copy of \( \text{page}_k \). This has the effect of subtracting \( \text{buf}_k \) from the parity page and adding \( \text{page}_k \).

- Set the protection of \( \text{page}_k \) 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 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.

![Figure 1: State at checkpoint 0.](image)

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.

![Diagram of processor states after checkpoint 0](image)

**Figure 2:** State slightly after checkpoint 0.

Since processor $P_1$ has used up its primary checkpointing buffer, checkpoint 1 must be started. To do so, $P_1$ goes through any synchronization necessary for the checkpoint to be consistent. When it is time to take its local checkpoint, $P_1$ XORs 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.

![Diagram of processor states during checkpoint 1](image)

**Figure 3:** Processor $P_1$ starts checkpoint 1.

If an application processor fails at this point, then the processors may again roll back to check-
point 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 $P_5$ 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: Processors $P_2, P_3$ and $P_4$ take checkpoint 1.

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: $p_{1}, \ldots, p_{2m}$, and $p_{n+1}, \ldots, p_{2m+n}$. 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 [PFL93]. We outline it in the following paragraph:

The calculations resemble Reed-Solomon codes [MS77, vL82]: Instead of performing bitwise arithmetic as the checkpoint processor does in the algorithm of the previous sections, each proces-
sor 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 [PFL93]. 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 far 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:

- The time to process page faults.
- The time to coordinate checkpoints for consistency.
- The time to send pages to the checkpoint processor.
- The frequency of checkpointing.

We do not analyze the first two components as they are the same for this algorithm as for other incremental consistent checkpointing algorithms [EJZ92, Pla93]. They should not amount to as much overhead as the third component. This component, the time to send pages to the checkpoint processor, depends on the speed of the interconnection network, and the number of bytes sent.

We notice that this component may be improved by a simple optimization: This comes from the fact that each processor sends $\text{diff}_k = \text{page}_k \oplus \text{buf}_k$ to the parity processor. This is as opposed to normal incremental checkpointing algorithms [FB89, EJZ92] that send $\text{page}_k$ to stable storage during an incremental checkpoint. The benefit of sending $\text{diff}_k$ is that all bytes of $\text{page}_k$ which have not been changed since the previous checkpoint will be zero in $\text{diff}_k$. This allows us to optimize the algorithm by sending only the non-zero bytes of $\text{diff}_k$, 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, were one to attempt to use $N + 1$ parity for *non*-incremental checkpointing, as suggested in Plank's thesis [Pla93], each processor would need to hold two extra copies of its checkpoint in main memory. This would require that the application processors allocate two thirds of their physical memory for the sole use of checkpointing. We consider this to be an unreasonable assumption.

The algorithm presented here alleviates this problem with incremental checkpointing: Extra space is required only for the pages of each processor's memory that have changed since the previous checkpoint. We allow the user to specify an upper limit on this space (this is $M$ in the algorithm) and when it is exhausted, a new checkpoint must be taken. There are two reasons to believe that this is a more reasonable approach.

First are previous results from Elnozahy, Johnson and Zwaenepoel [EJZ92]. 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. For the fine-grained checkpointing intervals that we propose (on the order of every second or every few seconds), the memory requirements (i.e. $M$) should be much smaller.

Second, we have instrumented some distributed programs in PVM [BDG+93] to record their behavior under this algorithm for varying values of $M$. The results in Table 1 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. The first row of the table shows the per processor CPU time of the multiplication with no checkpointing. All other rows show average per processor values, where each test was executed three times. Each application processor used 7.2 megabytes of memory without checkpointing. The processors have 8 kilobyte pages.

The data shows that for this program, a value of 800 kilobytes for $M$ yields both a reasonable checkpointing interval of around four seconds, and a reasonable **diff**'d checkpoint size of 172 kilobytes. Moreover it shows that encoding $diff_k$ to include only non-zero bytes results in a significant amount of compression.
<table>
<thead>
<tr>
<th>$M$ (bytes)</th>
<th>Average Time Between Checkpoints (CPU Seconds)</th>
<th>Number of Checkpoints</th>
<th>Incremental Checkpoint Size (bytes)</th>
<th>diff'd Checkpoint Size (bytes)</th>
<th>Compression Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>352.23</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>160K</td>
<td>0.48</td>
<td>763</td>
<td>80K</td>
<td>19K</td>
<td>76</td>
</tr>
<tr>
<td>400K</td>
<td>2.00</td>
<td>180</td>
<td>200K</td>
<td>78K</td>
<td>61</td>
</tr>
<tr>
<td>800K</td>
<td>4.33</td>
<td>82</td>
<td>400K</td>
<td>172K</td>
<td>57</td>
</tr>
<tr>
<td>1.6M</td>
<td>8.98</td>
<td>39</td>
<td>800K</td>
<td>300K</td>
<td>55</td>
</tr>
<tr>
<td>3.2M</td>
<td>19.96</td>
<td>17</td>
<td>1.6M</td>
<td>816K</td>
<td>49</td>
</tr>
<tr>
<td>4.8M</td>
<td>31.28</td>
<td>10</td>
<td>2.4M</td>
<td>1.3M</td>
<td>45</td>
</tr>
<tr>
<td>8.0M</td>
<td>80.11</td>
<td>3</td>
<td>4.0M</td>
<td>2.3M</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 1: Results of instrumenting a matrix multiply on 7 workstations

6 Related Work

Checkpointing is a well-documented topic in fault-tolerance. In parallel and distributed systems, the field has been divided into pessimistic [BBG83, PP83, BBG+89], optimistic [SY85, Joh89], and consistent checkpointing [CL85, SK86, LY87, KT87, Ahu89, CT90, CJ91, KMBT91, LNP92]. Implementations have been tested on unprocessors [FB89, LF90, LS92], multiprocessors [LNP90, LMJ92], distributed systems [KMBT91, EJZ92], and multicomputers [Pla93]. 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 [JZ87]. 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” [KCZ92]. 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 Garth Gibson in his design and implementation of RAID disk arrays [Gib90]. Gibson also addresses multiple-site failures, and gives an algorithm for for tolerating 2-site failures with $2n^{\frac{1}{2}}$ extra disks. This algorithm scales to tolerate $m$-site failures with $mn^{\frac{1}{m}}$ extra disks. Reed-Solomon codes are mentioned as a way to reduce the number of extra disks, but dismissed because of the extra complexity that such codes would require.
in the disk hardware. This isn’t a problem in our system because the unit of fault-tolerance is a processor, well-capable of handling the extra calculations.

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. In the discussion above, we argue that a fixed amount of extra memory is reasonable for this algorithm’s efficient operation. Results from Elnozahy, Johnson and Zwaenepoel [EJZ92], as well as our own instrumentation of a distributed matrix multiply corroborate this claim.

We have begun working to convert the above instrumentation into a full-blown checkpointing implementation on PVM. While PVM supports a wide variety of processors, our implementation will focus on DEC alpha workstations. This is because the alpha combines high performance, user-level access to page protection and fast page fault handling. The goal of this implementation will be to provide general fault-tolerance for PVM applications on DEC alphas, and to assess the overhead of checkpointing with \( N + 1 \) parity, as compared to checkpointing implementations to stable storage [LF90, LNP90, EJZ92, Pla93].

References


