In Proceedings of the Sixteenth Annual International Symposium on High Performance Computing Systems and Applications, 2002.

Compiler Support for Automatic Checkpointing*

Sung-Eun Choi
Advanced Computing Laboratory
Los Alamos National Laboratory†
Los Alamos, NM 87545
sungeun@lanl.gov

Steven J. Deitz
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195
deitz@cs.washington.edu

Abstract

Checkpointing is a key technology for applications on large cluster computer systems. As cluster sizes grow, component failures will become a normal part of operation, and applications will have to deal more directly with repeated failures during program runs. In this paper, we describe automatic checkpointing in the ZPL compiler and its advantages over traditional library- or system-based approaches that have no information about application behavior. We show that even naive compiler-inserted checkpoints can significantly reduce the size of the checkpoint recovery data, up to 73% in our application suite. We also introduce the notion of checkpoint ranges, *a range of code where processors can perform a local checkpoint at any time during the range. The compiler guarantees that these local checkpoints form a globally consistent checkpoint without global coordination by ensuring that there are no* in-flight *messages during the checkpoint range. Checkpoint ranges help further alleviate any additional network congestion caused by checkpointing.*

1. Introduction and Motivation

Cluster computer systems are extremely popular in industry, academia, and government. They rate very high on the *price-performance curve* due to the commodity nature of the computers, memory, and the network. Commodity pricing has also made it possible for non-government entities to build systems that are bigger and more powerful than any supercomputer ever built. The biggest problem with supercomputer-sized clusters is that they are comprised of many individually designed and tested systems (nodes). One of the many consequences of this is the unexpectedly high failure rate. While a vendor can make claims of individual system availability, multiply this by thousands of nodes and entire cluster availability is not very high. For example, in April 2000, Google, the web search engine, reported a node failure (unexpected reboot) rate of once every 20 minutes for their 6000 node cluster. For *oblivious* applications like web searches, failures can mostly be covered by simple redundancy; search data can be replicated and searches can be repeated. For more tightly coupled applications, a failure on one node will result in complete application failure. Clearly, application checkpointing is becoming more important.

Though checkpointing has long been part of many DOE applications running on supercomputers, the underlying assumptions are changing with the advent of terascale clusters. In particular, *a linear increase in the number of nodes in a cluster results in an exponential increase in failure rate.* It will no longer be sufficient to write all program data to a file once an "iteration." Applications will require more frequent checkpoints, and thus the overheads associated with checkpointing will be exacerbated.

While there is a rich research history in checkpointing support for applications, not much of the technology has made it into production use. The reasons behind this lack of acceptance can be argued, but ultimately the previous research will not be enough to save applications targeted at large clusters.

Most checkpointing systems provide support in an external library or at the operating system level. (We will discuss the compiler-based approaches in Section 4.) To reduce bandwidth requirements and checkpoint times, the systems focus efforts on reducing *individual* checkpoint size. The obvious problems with this approach are that it is highly sensitive to check-

*This research was funded by a grant from the Department of Energy's Office of Science and the Los Alamos Computer Science Institute. LANL LA-UR-02-1232.

†Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36.

point location (which in some systems is determined by a timer interrupt) and the fact that every page must be examined regardless of whether or not it will be saved (though this can be alleviated by caching the dirty state at the cost of increasing the time for every write). The more subtle problem is that the actual saved checkpoint size is equivalent to the entire process space. This is relevant for disk space requirements, but more importantly it is directly related to the time to recover from a checkpoint. It is not unreasonable that failures can occur during recovery, and thus reducing checkpoint recovery time is critical. On the proposed ASCI "Q" system at LANL, the vendor quoted 40 minutes *per node* to checkpoint or recover all of program memory!

In this paper, we argue for a language- and compiler-based approach for application checkpointing on cluster computer systems. As 64-bit architectures become commodity, the amount of addressable memory and thus a single node application's memory footprint can grow to multiple gigabytes. Multiply this by the number of nodes in the cluster and the system will be spending most of its time saving and restarting from checkpoint data. Providing support for automatic checkpointing in a high-level language enables the compiler to use standard analysis to find checkpoint locations which significantly reduce the amount of program state needed to restore the application after failure. Rather than using interrupt-based approaches that use approximations to the *optimal checkpoint interval* as a basis for when to checkpoint, we argue that carefully placed checkpoints in applications can greatly reduce the recovery checkpoint size as well as the individual checkpoint size.

The compiler-inserted checkpoints are only taken if a specified time interval has passed, and thus our work can incorporate much of the research on optimal checkpoint interval [11, 13, 19]. More importantly, our compiler-based technique is independent of the mechanism for saving the checkpoints. The language's runtime system can incorporate the use of any file system and other more interesting techniques such as diskless checkpointing [12] and Private Name Spaces [6, 7].

What makes our work unique is the inclusion of checkpointing in the language and compiler. This enables us to use accurate live variable analysis on high-level data structures such as arrays to significantly reduce the amount of user data saved. For our application suite, this results in up to a 73% reduction in checkpoint size. In addition, checkpointing parallel applications is simplified since the compiler is completely aware of all interprocessor communication, and thus inserts checkpoints only when there are no *in-flight* messages. This eliminates the need for message logging. Moreover, this results in *checkpoint ranges*, a communication-free range of code during which processors can perform a local checkpoint at anytime. These local checkpoints are guaranteed to form a globally consistent checkpoint without global coordination. In implementations that use the network for saving checkpoint data, checkpoint ranges help further alleviate any additional network congestion caused by checkpointing.

The remainder of the paper is organized as follows. In Section 2, we present the challenges of automatic checkpointing and the advantages of using a language- and compiler-based approach. In Section 3, we give a qualitative analysis of our approach for several benchmark programs. In Section 4, we discuss related work and in Section 5, we conclude and give directions for future work.

2. Automatic Checkpointing of Parallel Programs

Using a compiler for a high-level language affords us many advantages over traditional checkpointing techniques. The most useful of these is the availability of complete knowledge of the language's data structures and information from static analysis. In this section, we briefly describe the traditional approaches to checkpointing both sequential and parallel applications. Next, we discuss the two ideas that distinguish our work from existing systems: compiler-inserted checkpoints using live array variable analysis and the use of checkpointing ranges.

2.1. Traditional approaches to checkpointing

Traditional checkpointing systems are provided as either operating system or library support. Checkpoints are taken either automatically, via interrupts, or at programmer specified locations in the code. The major bottleneck for any technique is the checkpoint size. Thus the memory system is used to only write pages of memory that have changed since the last checkpoint, *i.e.*, dirty pages. This has been shown to greatly reduce the size of *individual* checkpoints. The amount of network bandwidth and disk space (or checkpoint memory in the case of diskless checkpointing) required to store the restartable checkpoint is still equal to the entire process space, or at least proportional to it if the checkpoint is compressed. Most importantly, that entire process space must be used to restore the application when recovering with checkpoint data, during which time cluster failures can occur.

The major difficulty with checkpointing parallel applications is interproccess communication. During a checkpoint, in-flight messages may result in an inconsistent checkpoint state because no one processor will save the data in these messages. To solve this problem, several systems have been developed to support message logging and rollback-recovery from the message log [1, 3].

2.2. Compiler-inserted checkpoints

In ZPL [14], arrays are first class citizens. The compiler and runtime system manage array allocation and deallocation, insert necessary array temporaries, etc. The compiler's internal representation of arrays and statements accessing arrays are not *scalarized* until code generation [2]. In other words, individual array elements are not recognized by the compiler as scalars and the loop nests used to iterate over arrays do not fall under the category of normal control flow. Maintaining a high-level array representation and semantic information during compilation ensures accuracy in analysis. This is how the compiler is able to extract parallelism, insert communication, and it is also how it can select good checkpoint locations.

Checkpoints are inserted into a program by first performing live array variable analysis followed by detection of communication-free checkpoint ranges. Currently, checkpoints are placed at the single location in the range with the fewest live array variables. One additional precaution must be made to insure that a globally consistent checkpoint is saved. In ZPL, there is a mechanism called *shattered control flow*, where the program briefly diverges from the SPMD programming model. Checkpoints cannot be placed within shattered control flow. Note that because the loops that iterate over arrays are not apparent yet, a checkpoint will never be placed in an inner loop, avoiding another problem with interrupt-based scheme.

Note that this technique is applicable to any language in which arrays are consider first class citizens; compilers for such languages (*e.g.*, Fortran 90) already perform the basic analysis necessary for automatic checkpointing. There is an immediate benefit for sequential programs. If the compiler also perform automatic parallelization, parallel programs can also benefit from being able to easily identify communication-free checkpoint ranges. For user-inserted comunication routines, live variable analysis is conservative around these routines and communication-free ranges must be explicitly identified. In Section 5 we discuss our other work in this area.

2.2.1 Comparison with memory system-based techniques

As discussed in the previous section, the most common technique for checkpointing data is to use the virtual memory system to write only dirty pages. This has been shown to dramatically reduce individual checkpoint size. The biggest advantage to this technique is that it enables applications to use checkpointing without special compilation and is thus fully general. For applications with relatively small memory footprints, this is undoubtedly the best solution. For any applications with large memory requirements, regardless of the working set size, memory system-based techniques can be little better than a complete core dump.

Figure 1 demonstrates this point with a checkpoint range taken from a code that performs heat dissipation over a metal plate. The checkpoint range begines at line 2 and ends at line 12. (For illustration purposes, the loops for iterating arrays have been inserted.) First consider performing a single checkpoint in the range. The live variable analysis identifies two "good" locations for a checkpoint both with a single live array: before line 2 (A is live) and before line 5 (T is live). A memory system-based approach would identify both array memories as *dirty* and write out all the pages associated with them. In addition, a memory system-based approach that is interrupt driven could potentially perform a checkpoint in an inner loop, adversely impacting locality optimizations that may have been performed by a low-level compiler.

If the compiler choses to perform two checkpoints during this range, it is possible that both approaches would save the same number of arrays. The compiler would perform the checkpoints before line 2 and before line 5. This illustrates the key advantage of compiler-based checkpointing: reduction in space required to save the checkpoint (total amount of recovery data) and the network bandwidth required to recover from the checkpoint. The compiler-based approach will only restore from one of the checkpoints and thus, only one of the arrays will be restored. Regardless of the size of the individual checkpoint, *memory system-based approaches must save and restore the entire memory footprint of the application.*

In parallel, different processors can perform the checkpoint at any location in the range (before lines 2, 5, or 9). Because there are no in-flight messages, there is no lost state. Therefore regardless of which specific checkpoint location is used by each individual processor, the recovered checkpoint restores a globally consistent state. In other words, *the checkpoint only need be consistent with respect to communication, not computation.* Note that this approach may diffuse tem-

```
1  while err > delta
      nearest neighbor communication for
         boundary values of A
2     for i = 1..N            live arrays: {A}
3        for j = 1..N
4           T[i,j] = (A[i-1,j] + A[i,j+1] + A[i+1,j] + A[i,j-1])/4
5     for i = 1..N            live arrays: {T}
6        for j = 1..N
7           A[i,j] = T[i,j]
8     err = 0
9     for i = 1..N            live arrays: {T, A}
10       for j = 1..N
11          if err < abs(T[i,j] - A[i,j])
12             err = abs(T[i,j] - A[i,j])
      collective communication for reduction
```

Figure 1. A checkpoint range in heat dissipation code. A single compiler-inserted checkpoint would occur before line 2 or 5, where only one of the array variables is live. A memory system-based approach would see both array memories as *dirty* and would write out both arrays.

poral hot spots in network traffic, at the cost of possibly increasing the size of the checkpoint.

2.2.2 Checkpoint interval specification

At runtime, the user can specify the checkpoint interval as a command line parameter. The chosen interval should be based on the problem size, expected failure rate, expected time to write the checkpoint data, and other factors. There has been a great deal of work in estimating the *optimal checkpoint interval*, and users are referred to the literature on that matter. When a checkpoint location is reached, if the elapsed time since the last checkpoint is equal to or greater than the specified interval, a checkpoint is taken. All checkpoints use the same interval clock. Consequently, generous placement of checkpoints does not necessarily result in frequent checkpoints.

When an application is run on multiple processors, the interval clock must be synchronized to ensure that the local checkpoints create a globally consistent checkpoint. In the most strict sense, a broadcast should be performed at every checkpoint location whether or not the checkpoint is taken (*i.e.*, the interval has not expired). Obviously, this can severely impact program performance.

Note that this problem also exists in other parallel application checkpointing systems, but again we have the advantage being compiler-based. Clocks can be synchronized during global operations such as reductions and scatter-gather (which are provided as operators in ZPL) by piggybacking the synchronization data (time and last checkpoint taken *by range*) on the communication that is inserted for the operation. A minimum reduction is performed on the synchronization data. The clocks are reset to the earliest time, and if the "minimum" checkpoint range number does not match the last local checkpoint range number, the new checkpoint is discarded. If clock skew cannot be adequately resolved, *i.e.*, local checkpoints are being discarded too frequently, we will revert back to our current heuristic which performs a simple exponential-like back-off with respect to the broadcast. The heuristic performs a broadcast at every checkpoint taken, but not every time the checkpoint is "passed." A broadcast is performed at the first and second checkpoints passed. If the interval has not expired, the next checkpoint is skipped and a broadcast is performed at the following checkpoint. If at this checkpoint, the interval has not expired, the next two checkpoints are skipped, etc.

3. Qualitative Analysis

In this section, we discuss the overhead of performing checkpointing from user level as well as the reduction in amount of recovery checkpoint data with respect to a single checkpoint location.

Cluster file systems are currently the most trailing technology in large cluster systems, yet they are one of the key components for efficient checkpointing. *There is no reasonable cluster file system in existence today.* Our current parallel implementation writes checkpoint data per processor to a file on an NFS mounted file system or a local disk, both inadequate for large-scale clustering environment. NFS is a recognized poor solution for clusters for a number of reasons including scalability to even just hundreds of nodes. Local storage is faster, but because nodes in terascale clusters are expected to fail continuously, quick recovery is essential and physically relocating a local disk from a failed node is not an option.

Our approach is independent of the file system or other backing store used for saving checkpoints, and we will address our plans to improve checkpointing with respect to this in the next section.

3.1. Overhead of checkpointing from user-space

A potential drawback of compiler-inserted checkpoints over memory system-based approaches is that

	AMD Athlon 1.0 GHz linux 2.4.9	**Alpha EV68** 833 MHz linux 2.4.9	**Intel PIII** 733 MHz linux 2.4.10
system (50 MB)	19.0	16.8	27.2
user (50 MB)	19.4	18.6	28.2
user (37.5 MB)	14.5	13.8	20.6
user (25 MB)	9.7	9.4	13.4

Table 1. The cost of checkpointing from user-space. The difference in the time (in seconds) to write a file from user-space versus using the memory system is negligible. These difference are quickly negated when the volume of data is reduced, thus reducing the checkpoint size will always be better than system-based I/O.

the operating system can write out pages much more quickly than writing a (local) file from a user-space program. We performed an experiment that showed the overhead to be very small for three processors running the Linux operating system.

For each processor type we ran four benchmarks, each wrote out a large chunk of contiguous data ten times to a file on the local disk. The *user* benchmark performed a `write`, followed by an `fsync` to force the writes to be committed (followed by an `lseek` to reset the file descriptor). The *system* benchmark performed an `msync` to write out dirty pages to a memory mapped file. Table 3.1 shows the time in seconds for an AMD Athlon, Compaq Alpha EV68, and Intel Pentium III (Coppermine) all running the 2.4 Linux kernel. The different between *user* and *system* is under 5% for all three processors. For reference, we measured the time for the *user* benchmark when the data size was reduced by 25% and 50%. As expected, the time is proportional to the data size, and thus even the slightest reduction in the amount of data saved during a checkpoint will negate the cost of performing the checkpoint in the user-space program. Reducing the checkpoint size will always be better than system-based I/O.

These numbers are optimistic with respect to the memory system-based approach because they do not include the overhead for maintaining the dirty bits. Furthermore, writing to an NFS-mounted file or other globally visible file will completely neutralize the cost of writing the data from user-space.

3.2. Reduction in checkpoint recovery data

Table 3.2 shows the reduction in recovery data due to compiler inserted checkpointing for a set of six applications written in ZPL. For illustrative purposes, only a single checkpoint is inserted in every "outer" loop nest. The larger, production codes (planetary integrator, CLAWPACK, NAS Multigrid) had more than one loop identified as an outer loop due to "warm-up" and initialization code.

For each checkpoint in each application, two numbers are given: *worst* and *compiler*. The *worst* column contains the total number of local and global arrays that are allocated at the checkpoint; this is equivalent to the number of arrays that would need to be restored when using a memory system-based approach to checkpointing. The *compiler* column contains the total number of local and global arrays that the compiler determined to be live; this is the number of arrays that are saved and restored from the checkpoint. The compiler can reduce the number of arrays saved by as much as 73%.

4. Related Work

Compiler support for checkpointing is not a new idea. Plank *et al.* proposed a scheme in which the compiler would assist the operating system in checkpointing using techniques described in this paper such as live variable analysis [10]. But the compiler was not responsible for checkpointing, only for providing information to a system. In addition, without a high-level representation, the analysis may not be as accurate as in the ZPL compiler.

Our work is most similar to that done by Strumpen on the *porch* compiler [17, 16]. We follow the *porch* idea of transforming a standard program into a fault-tolerant program. However, our work differs due to its focus on a high-level language and parallel computing. We are better able to identify live arrays and can handle parallel execution, but *porch* is a compiler to transform C programs and is more general.

application	*array size (8-bytes doubles)*	*arrays recovered worst*	*arrays recovered compiler*	*percent reduced*
time-sliced planetary integrator	N	17	13	24%
		17	12	29%
		17	12	29%
		17	11	35%
		17	10	41%
shallow water equations	$N \times N$	14	6	57%
simple hydrodynamics	$N \times N$	27	16	41%
CLAWPACK elasticity equations (finite volume method)	$N \times N$	24	9	63%
		11	3	73%
		11	3	73%
SWEEP 3D	$N \times N \times N$	21	18	14%
NAS Multigrid	$N \times N \times N$	3	3	0%
		3	2	33%
		3	1	67%

Table 2. Reduction in checkpoint recovery data due to compiler inserted checkpointing. The *worst* column shows the number of arrays recovered in a checkpoint using a memory system-based approach; the *compiler* column shows the number of arrays actually needed to recover from a checkpoint.

5. Conclusions and Future Work

In this paper, we argue for automatic checkpointing in a high-level language and compiler. The need for this is motivated by the emergence of large clusters of commodity computers, but even in sequential applications with large memory footprints, compiler-inserted checkpoints can reduce the amount of checkpoint data significantly. While a traditional memory system-based approach to checkpointing may be able to reduce the size of individual checkpoints, the total amount of data saved will include the entire memory footprint. Thus the recovery process will require restoration of the entire process space. A language- and compiler-based approach only saves the minimal amount of data required for recovery at each checkpoint, greatly reducing the space and network bandwidth requirements as well as recovery time. In addition, the compiler can eliminate the need for message logging by only inserting checkpoints when there are no in-flight messages. This allows us to identify *checkpoint ranges*, where any checkpoint taken in a particular range of statements will contribute to a globally consistent checkpoint.

We have implemented automatic checkpointing in the ZPL language and compiler. The ideas are applicable to any language in which arrays are considered first class citizens; most compilers for such languages (*e.g.*, Fortran 90) already perform the basic analysis necessary for automatic checkpointing. We have shown that saving the checkpoint data from a user-space program incurs little penalty ($< 5\%$) over using a memory system-based approach for the same amount of data, and as such, even the slightest reduction in the amount of memory written can make up for the penalty. Finally, our analysis shows that for a set of benchmark programs and real applications, the reduction in checkpoint recovery data size can be as much as 73% over the memory system-based approach.

5.1. Future Work

The motivation behind this work is to enable applications to survive failures, potentially without restarting all nodes from a checkpoint. We are involved with a cluster monitoring project called Supermon [8, 15], a high speed cluster monitoring tool. We are currently adding low level support to also return hardware monitoring data such as CPU temperature and fan speed. This information will be collected and interpreted by a status process with the goal of predicting hardware failures. When a node failure may be imminent, we will migrate the running ZPL (node) program to another working node.

This research is part of a larger project aimed at building more reliable and efficient terascale clusters. Our current clusters run a single-system image version of Linux called BProc [5]. BProc supplies fast (user) directed process migration. We will combine BProc's direct migration facilities with ZPL checkpointing to quickly migrate and restart the process. In addition we are investigating the use of restartable network connections such as CIPE (Cryto IP Encapsulation) [18]. Though CIPE was designed for building encrypting IP routers, it also provides persistent network connections even after disconnecting from the network. It is an IP layer and thus can transparently replace the existing IP layer. CIPE is currently available for Linux (and Windows).

Despite our arguments against memory system-based approaches, we recognize that a language- and compiler-based solution is not always appropriate. For example, there is a large base of legacy applications at LANL and other sites which will not be rewritten for new architectures. In addition, there are a number of parallel codes that are not memory bound and would best be served by a memory system-based approach. In conjunction with the compiler work, we are investigating using state information from the messaging library (MPI) for detecting communication free ranges in SPMD style programs. Specifically, we will use LA-MPI, a standard compliant MPI 1.2 implementation from LANL that performs reliable and resilient message [4]. LA-MPI maintains queues of message state data for the various states that messages can be in before they are delivered to the recipient'ss memory. When these queues are all empty, there are no in-flight messages, and we are in a communication-free checkpoint range. Within these ranges, we will use BProc's Virtual Memory Area (VMA) dumper to save program memory state to a local RAM disk which will be dribbled out to a file server. The VMA dumper is very fast because it essentially performs a memory-to-memory copy with copy-on-write optimizations. For the legacy applications with large memory footprints, we will need to balance the rate at which the checkpoint is dribbled out to the file server and the rate at which the program is touching pages, and thus copying the old pages and increasing memory usage.

Finally, as alluded to in the previous section, there are no good solutions for cluster file systems. All existing solutions provide a global view of the file system at the cost of decreasing reliability (all file servers must be up for *any* file to be accessible) and/or increasing interdependence (the file servers must be up for the cluster to be up). We are currently implementing a Plan 9 [9] style *Private Name Spaces* file system for our clusters called V9FS [6, 7]. Private Name Spaces decouple files from the file system; users have access to files (Name Spaces) and not the entire file system that the files live on. In addition to benefits with respect to security, Private Name Spaces eliminate the unnecessary side affects of sharing an entire file system. We will be testing ZPL application checkpointing on a cluster testbed with V9FS using both a disk-based file server as well as the V9FS network RAM disk. The network RAM disk is a very attractive solution because it provides a transparent way to perform diskless checkpointing.

Acknowledgements. We would like to thank Erik Hendriks, Ron Minnich, and Matt Sottile.

References

[1] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, casual, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149 – 159, 1998.

[2] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1996.

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

[4] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network failure tolerant message passing system for terascale clusters. *Submitted for publication*, 2002.

[5] E. A. Hendriks. The Beowulf distributed process space. *Submitted for publication*, 2002.

[6] R. Minnich. 9.2.u: A user-mode private name space system for unix. Technical report, Defense Advanced Research Projects Agency, September 1998.

[7] R. Minnich. Private namespaces for linux. *Dr. Dobb's Journal*, December 2001.

[8] R. Minnich and K. Reid. Supermon: High performance monitoring for linux clusters. In *The Fifth Annual Linux Showcase and Conference*, 2001.

[9] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[10] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[11] J. S. Plank and W. Elwasif. Experimental assessment of workstation failures and their impact on checkpiongint systems. In *International Symposium on Fault-tolerant Computing*, 1998.

[12] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):300–310, 1998.

[13] J. S. Plank and M. Thomason. The average availability of parallel checkpointing systems and its importance

in selecting runtim parameters. In *International Symposium on Fault-tolerant Computing*, pages 250 – 259, 1999.

[14] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[15] M. J. Sottile and R. G. Minnich. Supermon: Cluster monitoring as if performance mattered. *Submitted for publication*, 2002.

[16] V. Strumpen. Compiler technology for portable checkpoints. Technical report, Submitted for publication, 1998.

[17] V. Strumpen and B. Ramkumar. Portable checkpointing and recovery in heterogeneous environments. Technical report, University of Iowa (UI-ECE TR-96.6.1), June 1996.

[18] O. Titz. Crypto IP encapsulation protocol. 1999.

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