Performance Results of `ickp` — A Consistent Checkpointer on the iPSC/860

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:

Scalable High Performance Computing Conference
Knoxville, TN
May 23-25, 1994

Performance Results of `ickp` — A Consistent Checkpointer on the iPSC/860

James S. Plank

Department of Computer Science
University of Tennessee
Knoxville, TN 37996

Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

This paper presents performance results of `ickp`*, a program that checkpoints applications written on the Intel iPSC/860.* `ickp` *is the first implementation of general-purpose checkpointing on a multicomputer.* `ickp` *implements three consistent checkpointing algorithms, as well as two optimizations, and recovery. This paper displays the results of testing* `ickp` *on many benchmark programs. These benchmarks are useful iPSC/860 programs written by other scientists, and not toy benchmarks used to test the checkpointer.*

An alpha release of `ickp` *has been made to the Intel community.*

Introduction

Checkpointing has received a great deal of attention recently as increasing parallelism in distributed systems and supercomputers render programs more prone to failure. This paper shows the results of `ickp`, a consistent checkpointer for the Intel iPSC/860. `ickp` is the first implementation of checkpointing on a multicomputer.

Significant results that can be drawn from this paper are as follows:

- Consistent checkpointing is a reasonable strategy for providing fault-tolerance and coarse-grained job-swapping on multicomputers.
- For today's multicomputers, the overhead of control messages for checkpointing is insignificant compared to the overhead of saving the checkpoint to stable storage.
- Main memory checkpointing is a good strategy for improving the overhead of checkpointing in all applications. Checkpoint compression is only an optimization when the contents of program's address space have enough structure to be compressed. As will be shown, this happens frequently, especially with Fortran programs.

Checkpointing is the act of saving the volatile state of a program (e.g. registers, memory) to stable storage (e.g. disk) so that after a failure, the program may be restored to the state of its most recent checkpoint. The major use of checkpointing is for fault-tolerance of long-running programs. By checkpointing at moderate intervals, like once an hour, a long-running program is guaranteed to lose a maximum of one hour of running time to each system failure. This is preferable to restarting the program from scratch at each failure.

A second use of checkpointing is for coarse-grained job-swapping. Some multicomputers, like the iPSC/860, have no facilities for job-swapping, rendering them difficult to share among users. This is especially a concern for the execution of long programs, whose running time may exceed the slice of time allotted to the user of a heavily shared machine. With checkpointing, a user can checkpoint his program just before his allocated time slice has expired, and then restore the checkpoint at his next time slice.

There has been a huge amount of research on algorithms for checkpointing in parallel and distributed systems [4, 7, 8, 11, 14, 15, 17, 18, 20, 23, 25, 26, 28, 29, 30, 31, 32]. However, there have been surprisingly few implementations. Scattered implementations of general-purpose checkpointing exist for uniprocessors [12, 21, 24], multiprocessors [19, 22], and distributed systems [3, 5, 10, 16], but none for multicomputers.

The goal of this paper is to experimentally justify checkpointing as a means for fault-tolerance and job-swapping on multicomputers. Additionally, we compare the results of three checkpointing algorithms de-

scribed in previous literature [7, 23, 27], as well as two optimizations to checkpointing.

Description of `ickp`

`ickp` is a checkpointing library for the Intel iPSC/860. It may be invoked in one of two ways: by setting a periodic interval at which to checkpoint, or by making a library call. Recovery is attained by loading the executable with a special set of command line arguments. `ickp` requires no changes to the application programmer's code or to the iPSC/860 operating system. A simple relinking of object code with the `ickp` library is all that is necessary.

`ickp` is designed to solve the special problems brought about by checkpointing a multicomputer, as opposed to checkpointing distributed systems. A multicomputer is a single machine comprised of individual processors that communicate solely by message-passing. The processors do not share memory, and as such, a multicomputer resembles a distributed system. However, it differs from a distributed system as the processors are usually all in the same box or room, and are typically connected by a high-speed, wormhole-routing interconnection network. A more thorough discussion of multicomputers may be found in [2].

Multicomputers are typically used in a different manner than distributed systems. Although the objective of both is to exploit coarse-grained parallelism, multicomputers are more centralized than distributed systems, and thus have the following distinct characteristics:

- The individual processors are uniform, and the interconnection network is typically faster than in distributed systems.
- The processors all share the same I/O system. Thus, all processors share the same file system and disks. For example, the iPSC/860 is often configured with the "Concurrent File System" (CFS), which consists of four I/O processors controlling four disks, which stripe data to achieve high bandwidth.
- Job control is very coarse-grained. For example, on the iPSC/860, users must exclusively allocate all of the processors which they desire to use. These processors may not be used by any other user until they are deallocated. There is no automatic job-swapping or multiprogramming.
- The failure model is wholesale: If a node in an allocated collection fails, then the rest are unusable.

These characteristics lead to certain types of programs:

- Programs tend to use the processors uniformly. In other words, they resemble SIMD programs, as all processors usually perform the same actions at the same time, even if they do not synchronize often. This is as opposed to programs for distributed systems, which often use a client-server paradigm to take account of processor and network non-uniformity.
- Programs tend to use I/O as sparingly as possible, as it is a potential bottleneck. A typical program for a multicomputer performs a sequence of disk reads, then a large amount of computation, followed by a sequence of disk writes. Programs are rarely interactive, as there is typically no multiprogramming to keep CPU utilization at a reasonably high level.
- There is often a good deal of spare physical memory. This is because processors are often allocated exclusively, as on the iPSC/860, and thus any memory not used by the currently running program remains unused until the program is over.

All of these factors favor "consistent" checkpointing over other types of checkpointing (i.e. optimistic [14, 13, 29], pessimistic [4, 5], or independent checkpointing [32]), as they are designed for distributed systems, where single-node failures, slow interconnection networks, distributed resources, non-uniform processors, and interactive programs are frequent, if not the norm. We describe consistent checkpointing as implemented by `ickp` below.

Checkpointing algorithms

A consistent checkpoint of a collection of processors consists of a processor state and a message state. The processor state is comprised of individual checkpoints for each processor in the system. The message state contains each message m such that m was sent by processor p_i before p_i's checkpoint, and received by p_j after p_j's checkpoint. There may be no messages m such that m is sent by processor p_i after p_i's checkpoint, and received by p_j before p_j's checkpoint. Otherwise, the checkpoint is inconsistent. More thorough discussions of consistent checkpoints may be found elsewhere [9, 17, 23, 32].

There has been much theoretical research on consistent checkpointing algorithms [1, 7, 8, 9, 17, 18, 20, 23, 28], most of which focuses on optimizing the number of messages sent by the system to assure consistency. We chose to implement three different algorithms in `ickp`. The complete description of each algorithm may be found elsewhere. We briefly describe each below:

Sync-and-stop (SNS): The SNS algorithm is a very simple checkpointing algorithm first defined by Plank [27]. It is basically a two-phase commit, which freezes the entire multicomputer to take a consistent checkpoint with no message state, and then unfreezes the machine when it's finished. The SNS algorithm requires $O(n)$ extra messages to ensure consistency, where n is the number of processors in the multicomputer.

Chandy-Lamport (CL): In 1985, Chandy and Lamport defined a consistent checkpointing algorithm upon which all others save the SNS algorithm are based [7]. Unlike the SNS algorithm, it doesn't freeze the entire multicomputer, and allows a message state to exist. Basically, it sends *marker* messages through every logical channel in a system to both ensure consistency and define the message state. In multicomputers with wormhole routing, this leads to $O(n^2)$ total messages, since every processor has a logical channel to every other processor.

Network Sweeping (NS): The NS algorithm was defined by Li, Naughton and Plank to reduce the overall number of messages of the CL algorithm on multicomputers [23]. This algorithm takes advantage of static wormhole routing functions to checkpoint with $O(l)$ messages, where l is the number of physical links in the multicomputer. For a hypercube like the iPSC/860, this is $O(n \log n)$ messages. For a mesh like the Intel Paragon, this is $O(n)$ messages.

Optimizations

There are basically two metrics by which one can measure a checkpointer's performance: checkpoint time, and checkpoint overhead. Checkpoint time is the total time that it takes for the checkpointer to complete a checkpoint, from start to finish. It is a measure of how long it takes the checkpointer to run, and also determines how many checkpoints can be taken over a period of time. Checkpoint overhead is the total time added to the application program as a result of the checkpoint. It is a measure of how invasive the checkpointer is. For the purposes of fault-tolerance, checkpoint overhead is the important metric: Since checkpoints are needed at relatively large intervals, like once an hour, it is less important that a checkpoint completes quickly, as long as it does so before the next checkpointing interval. For the purposes of job-swapping, however, it is important to keep both time and overhead low.

`ickp` implements two optional optimizations, whose purpose is to lower the checkpoint time and/or overhead. The common thread in both optimizations is that the major source of checkpoint overhead is performing the actual writing of the checkpoint to disk. Thus, both attempt to achieve better performance by reducing or hiding the latency of the disk writes.

Main Memory Checkpointing: In main memory checkpointing, extra physical memory is employed to delay the writing of the checkpoint to disk. If there is enough memory unused by the application, then at checkpoint time, the checkpointer simply copies the checkpoint to this extra physical memory, and then writes the copy to disk after restarting the application program. This overlaps the writing of the checkpoint with the execution of the application, and should drastically reduce checkpoint overhead. If the checkpoint does not fit into physical memory, then enough of the checkpoint is written to disk while the application is stopped that the rest can fit into main memory. Although this lessens the optimization, it should still show an improvement over the unoptimized case, since fewer bytes are being written to disk while the application is stopped.

Compression: An obvious optimization for checkpointing is to use compression to decrease the number of bytes that get written to disk. This assumes that the state of memory has a structure that allows it to be compressed. Although compression time is greater than memory-to-memory copy time, it has the potential to reduce the size of a checkpoint enough that a checkpoint that does not fit into main memory uncompressed may fit when compressed. Thus, compression has the potential to further optimize the main memory optimization.

To test compression, `ickp` implements a variation of a compression algorithm defined by Wheeler and called "Algorithm 1" in [6]. This algorithm maintains a hash table which predicts the next byte to compress based on a function of the previous three bytes. If the prediction is correct, a 1

bit is emitted. Otherwise, a 0 bit is emitted followed by the actual byte. `ickp`'s variation adds run-length encoding to the algorithm. The compression algorithm was chosen because of its fast on-line behavior. Run-length encoding was added so that large, patterned segments of memory, such as a page of zeros, can be compressed to a couple of bytes. Both the algorithm and variation are discussed in detail in [27].

Implementation and tests

As stated above, `ickp` was implemented on the Intel iPSC/860. We tested the performance of `ickp` on six test programs: A genetic algorithm visualization tool, a sparse matrix solver, a FFT program, a matrix reduction program (reduction to block Hessenberg form), a molecular dynamics program, and an iterative equation solver. All are real application programs written by iPSC/860 programmers to further their research. We chose them as they provide real-life examples of multicomputer programs, as opposed to artificial test programs written for the sole purpose of this project.

The graphs in Figures 1 and 2 contain performance results of checkpointing these programs on a 32-node iPSC/860, running a CFS implementation with four disks. Each bar on each graph represents the average of multiple program runs, where one checkpoint is taken per run. As stated above, checkpoint time is the average time to complete a checkpoint from start to finish, and overhead is the amount of time added to the running time of the program. The checkpoint sizes shown are per processor. In each test of the main-memory optimization, one extra megabyte of physical memory is reserved per processor for checkpointing.

The first interesting result of these experiments is the lack of improvement in checkpoint time and overhead due to the CL and NS algorithms. One might expect the performance of the CL and NS algorithms to differ due to the number of extra messages sent. However, this difference is negligible in relation to the overall checkpoint time and overhead. The average time to send a consistency message on the iPSC/860 is 0.001 second. Thus, the difference between each processor sending 32 messages in the CL algorithm versus five messages in the NS algorithm is on the order of 0.03 seconds. This is noise compared to the time to write the checkpoint to disk.

Second, neither the CL or NS algorithm appears to improve performance much over the simple SNS algorithm. The reason is that all three algorithms basically

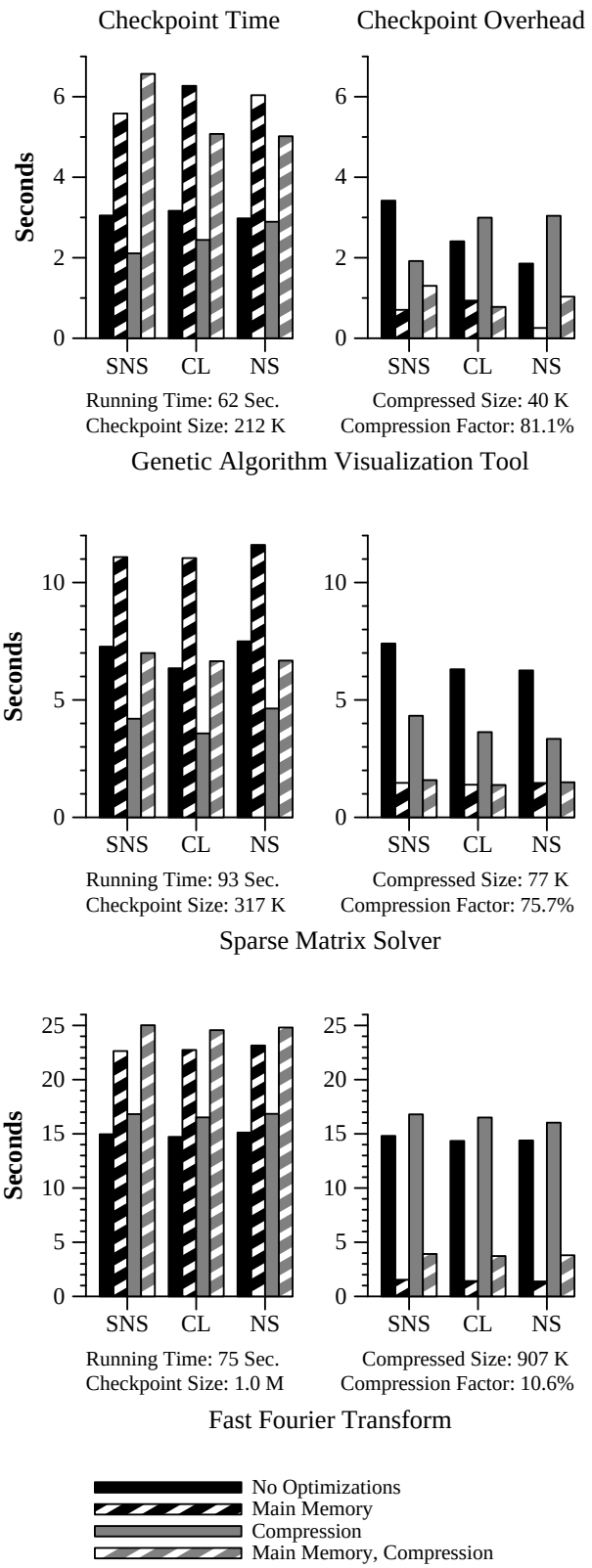

Figure 1: Performance of `ickp` on a 32-node iPSC/860

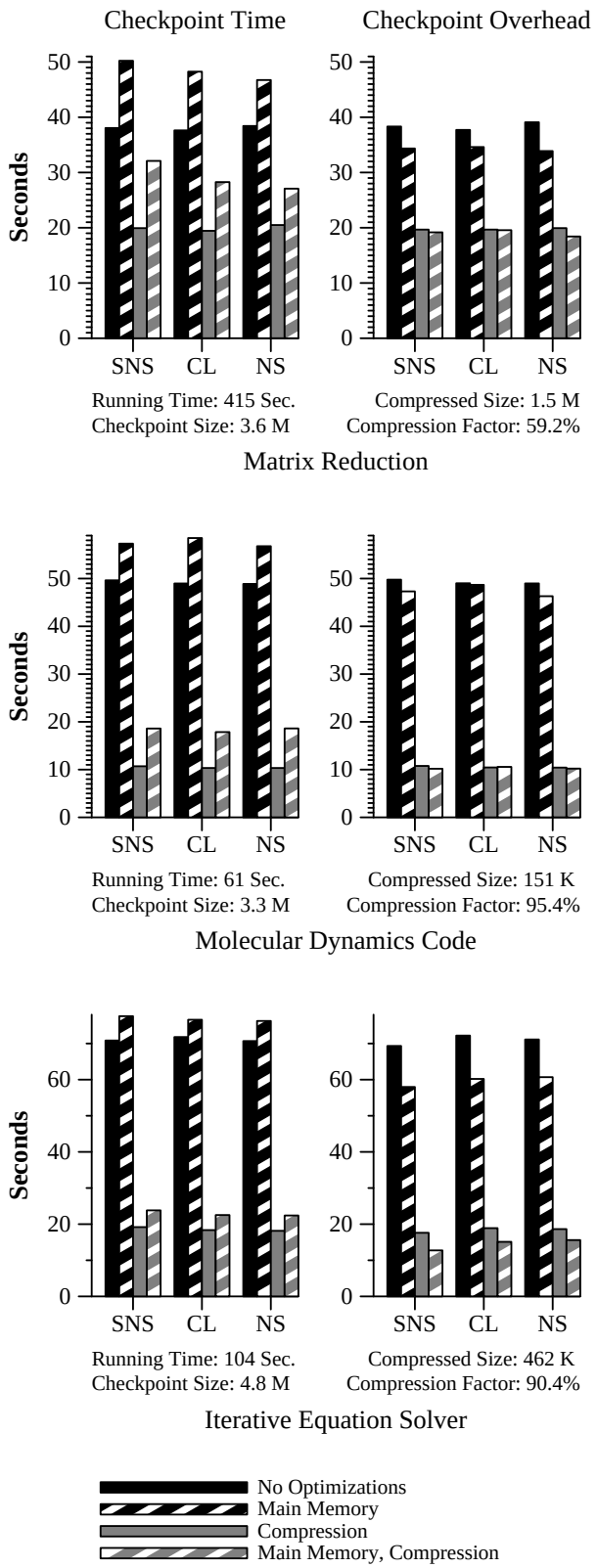

Figure 2: Performance of `ickp` on a 32-node iPSC/860

cause all nodes to checkpoint at once. This would not be true on a distributed system, where large message delays and non-uniform processor speeds would penalize the system-wide synchronization of the SNS algorithm. However, on a multicomputer, where message transmission is fast and processors are uniform, all three algorithms perform similarly.

It should be clear in these applications that both main memory checkpointing and compression are beneficial, although to varying extents. When there is enough extra physical memory to hold an entire checkpoint, as in the first three graphs, main-memory checkpointing keeps the overhead of checkpointing under two seconds. This is at the expense of checkpoint time, which is increased by as much as ten seconds in the FFT program. When the checkpoint size increases beyond the size of available extra physical memory, as in the last three graphs, the benefits of main-memory checkpointing decrease, but never to the point where it increases overhead. In fact, checkpointing without the main memory optimization is equivalent to checkpointing with the optimization, but with no extra memory.

Main memory checkpointing has a second advantage when the entire checkpoint fits into main memory. This is that it decreases overhead by eliminating contention for disks. Without this optimization, when all processors attempt to checkpoint at the same time, they end up serializing at the disks. For example, in our tests, there is an 8:1 processor to disk ratio. If the entire checkpoint fits into memory, each processor is free to write the checkpoint to disk at its leisure, and thus in the experiments, the processors order themselves and wait for their turn to write the main memory checkpoint to disk. This further increases checkpoint time, but eliminates overhead due to disk contention.

In the experiments, compression is a good optimization when the address space is compressible. Compression factors ranged in these tests from 10% to 96%, and most were over 50%. In the last three programs, the compression caused significant savings in checkpoint time and overhead, as the address spaces were very large and very compressible. The reason for this is that these were written in Fortran, where static allocation forces programmers to overestimate their allocation needs. The result is a significant amount of allocated but unused memory, which compresses away to almost nothing.

Conclusions

The experiments and results of checkpointing on the iPSC/860 allow us to draw the following conclusions:

- The performance of consistent checkpointing is sufficient to provide coarse-grained fault-tolerance and job-swapping on a multicomputer like the iPSC/860. For checkpointing on the order of once an hour, for either fault-tolerance or job-swapping, checkpoint times and overheads of under a minute are completely reasonable. The bottleneck in all cases is the time to write to disk, which on the iPSC/860 is on the order of three megabytes per second.

- The checkpointing algorithm makes little difference in terms of checkpoint time and overhead. In particular, the number of extra messages induced by the algorithm is not a concern because the disk-write times are four orders of magnitude higher than the message transmission time. As n scales to include 1000 processors or more, then the difference between algorithms should be more marked. These results corroborate a checkpointing implementation by Elnozahy, Johnson and Zwaenepoel [10] on a 16-processor distributed system, where the time to write to stable storage clearly dominates any other component of the checkpointing overhead.

- Main memory checkpointing is a viable optimization for improving checkpoint overhead in all cases. Thus it is ideal when checkpointing is used for fault-tolerance. It is especially useful as a way to reduce contention for shared disks.

- Compression is a good optimization of both checkpoint time and overhead when the address space is compressible. As our tests show, this happens often, most dramatically with Fortran programs where programmers tend to overallocate memory.

  Our results concerning compression are significant in comparison to results obtained by Li and Fuchs [21]. In their tests, they were able to get compression factors between 30% and 70% using LZW compression but observed a slowdown in checkpointing because the compression was too slow. In contrast, we were able to obtain a significant improvement over straightforward checkpointing with compression. We assume that the difference is due to a faster online compression algorithm.

Acknowledgements

The authors would like to thank the following people for donating their code and helping us test checkpointing: Tom Cundari, Jack Dongarra, Victor Eijkhout, Mootaz Elnozahy, Al Geist, David Johnson, Youngbae Kim, Michael Schmidt, and Michael Vose.

References

[1] M. Ahuja. Repeated global snapshots in asynchronous distributed systems. Technical Report OSU-CISRC-8/89 TR40, Ohio State University, August 1989.

[2] W. C. Athas and C. L. Seitz. Multicomputers: Muessage-passing concurrent computers. *Computer*, 21(8):9–24, August 1988.

[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] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[7] 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.

[8] 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.

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

[10] 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.

[11] E. N. Elnozahy and W. Zwaenepoel. One the use and implementation of message logging. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, 1994.

[12] 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.

[13] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

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

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

[16] 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.

[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] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the 4th International Conference on Data Engineering*, pages 154–163, February 1988.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] J. McDermid. Checkpointing and error recovery in distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 271–282, April 1982.

[26] J. S. Plank and K. Li. Faster checkpointing with $n+1$ parity. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, Austin, TX, June 1994.

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

[28] 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.

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

[30] Y. Tamir and C. H. Séquin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 13th International Conference on Parallel Processing*, pages 32–41, 1984.

[31] Y. M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154, October 1992.

[32] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1993.

Addendum

(This is not part of the paper as it appeared in SHPCC)

So, this paper is appearing as the following:

```
author          J. S. Plank and K. Li
title           Ickp --- A Consistent Checkpointer for Multicomputers
journal         IEEE Parallel and Distributed Technologies
volume          2
number          2
year            1994
```

In this version, we add a little more discussion concerning compression. Specifically, after looking into it, we see that the difference between the compression results of `ickp` and that of Li & Fuchs is not the speed of the implementation. Instead, we see the following (quoted from the journal version):

"Analytically, compression benefits checkpointing whenever:

$$\frac{B}{C} + \frac{(1-f)B}{D} < \frac{B}{C},$$

where C is the compression speed, f is the compression factor, D is the speed of disk writes, and B is the size of the checkpoint. Factoring out B and solving for f, we see that compression is beneficial whenever

$$f > \frac{D}{C}.$$

"In our system, each node compresses at 0.343 Mbytes per second. Because all nodes compress in parallel, the 32-processor system compresses at 11.0 Mbytes per second. The Concurrent File System writes at 2.16 Mbytes per second, giving us values of $C = 11.0$ and $D = 2.16$. Therefore, compression is beneficial whenever $f > 19.3\%$. This is a very low number.

"In contrast, Li and Fuchs also implemented checkpointing with compression on a uniprocessor [21]. Using the LZW compression algorithm, they reported values of $C = 0.034$ and $D = 0.100$, meaning that f must be greater than 100% for compression to be beneficial. Because this is impossible, compression could not improve checkpointing on their system."