Checkpointing Shared Memory Programs at the Application-level

Greg Bronevetsky, Daniel Marques, Keshav Pingali*
Department of Computer Science
Cornell University
Ithaca, NY 14853
{bronevet,marques,pingali}@cs.cornell.edu

Peter Szwed
School of Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853
pkszwed@csl.cornell.edu

Martin Schulz†
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
schulzm@llnl.gov

ABSTRACT

Trends in high-performance computing are making it necessary for long-running applications to tolerate hardware faults. The most commonly used approach is checkpoint and restart (CPR) - the state of the computation is saved periodically on disk, and when a failure occurs, the computation is restarted from the last saved state. At present, it is the responsibility of the programmer to instrument applications for CPR.

Our group is investigating the use of compiler technology to instrument codes to make them self-checkpointing and self-restarting, thereby providing an automatic solution to the problem of making long-running scientific applications resilient to hardware faults. Our previous work focused on message-passing programs.

In this paper, we describe such a system for shared-memory programs running on symmetric multiprocessors. This system has two components: (i) a pre-compiler for source-to-source modification of applications, and (ii) a runtime system that implements a protocol for coordinating CPR among the threads of the parallel application. For the sake of concreteness, we focus on a non-trivial subset of OpenMP that includes barriers and locks.

One of the advantages of this approach is that the ability to tolerate faults becomes embedded within the application itself, so applications become self-checkpointing and self-restarting on any platform. We demonstrate this by showing that our transformed benchmarks can checkpoint and restart on three different platforms (Windows/x86, Linux/x86, and Tru64/Alpha). Our experiments show that the overhead introduced by this approach is usually quite small; they also suggest ways in which the current implementation can be tuned to reduced overheads further.

*This research was supported by DARPA Contract NBCH30390004 and NSF Grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

†Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

1. INTRODUCTION

The problem of making long-running computational science programs resilient to hardware faults has become critical. This is because many computational science programs such as protein-folding codes using *ab initio* methods are now designed to run for weeks or months on even the fastest available computers. However, these machines are becoming bigger and more complex, so the mean time between failures (MTBF) of the underlying hardware is becoming less than the running times of many programs. Therefore, unless the programs can tolerate hardware faults, they are unlikely to run to completion.

The most commonly used approach in the high-performance computing arena is checkpoint and restart (CPR). The state of the program is saved periodically during execution on stable storage; when a hardware fault is detected, the computation is shut down and the program is restarted from the last checkpoint. Most existing systems for checkpointing such as Condor [7] take System-Level Checkpoints (SLC), which are essentially core-dump-style snapshots of the computational state of the machine. A disadvantage of SLC is that it is very machine and OS-specific. Furthermore, system-level checkpoints by definition cannot be restarted on a platform different from the one on which they were created.

In most programs however, there are a few key data structures from which the entire computational state can be recovered; for example, in an n-body application, it is sufficient to save the positions and velocities of all the particles at the end of a time step. In Application-Level Checkpointing (ALC), the application program is written so that it saves and restores its own state. This has several advantages. First, applications become self-checkpointing and self-restarting, eliminating the extreme dependence of SLC implementations on particular machines and operating systems. Second, if the checkpoints are created appropriately, they can be restarted on a different platform. Finally, in some applications, the size of the saved state can be reduced dramatically. For example, for protein-folding applications on the IBM Blue Gene machine, an application-level checkpoint is a few megabytes in size

whereas a full system-level checkpoint is a few terabytes. For applications on most platforms, such as the IBM Blue Gene and the ASCI machines, hand-implemented ALC is the default.

In this paper, we describe a semi-automatic system for providing ALC for shared-memory programs, particularly in the context of Symmetric Multi-Processor (SMP) systems. Applications programmers need only instrument a program with calls to a function called `potentialCheckpoint()` at places in the program where it may be desirable to take a checkpoint (for example, because the amount of live state there is small). Our Cornell Checkpointing Compiler (C^3) tool then automatically instruments the code so that it can save and restore its own state. We focus on shared-memory programs written in a subset of OpenMP [10] including parallel regions, locks, and barriers. We have successfully tested our checkpoint/restart mechanism on a variety of OpenMP platforms including Windows/x86 (Intel compiler), Linux/x86 (Intel compiler), and Tru64/Alpha (Compaq/HP compiler).

The system described here builds on our previous work on ALC for message-passing programs [2, 1]. By combining the shared-memory work described here with our previous work on message-passing programs, it is possible obtain fault tolerance for hybrid applications that use both message-passing and shared-memory communication.

The remainder of this paper is structured as follows. In Section 2, we briefly discuss prior work in this area. In Section 3, we introduce our approach and how our tool is used. In Section 4, we present experimental results. Finally, we discuss ongoing work in Section 5.

2. PRIOR WORK

Alvisi et al. [6] is an excellent survey of techniques developed by the distributed systems community for recovering from fail-stop faults.

The bulk of the work on CPR of parallel applications has focused on message-passing programs. Most of this work deals with SLC approaches, such as [13] [3] and thus results in solutions where the message passing library must be modified in order to allow checkpointing to take place. At the application-level, most solutions are hand-coded checkpointing routines run at global barriers. Recently, our research group has pioneered preprocessor-based approaches for implementing ALC (semi-)automatically [2, 1].

Checkpointing for shared memory systems has not been studied as extensively. The main reason for this is that shared memory architectures were traditionally limited in their size and hence fault tolerance was not a major concern. With growing system sizes, the availability of large-scale NUMA systems, and the use of smaller SMP configurations as building blocks for large-scale MPPs, checkpointing for shared memory is growing in importance.

Existing approaches for shared memory have been restricted to SLC and are bound to particular shared memory implementations. Both hardware and software approaches have been proposed. SafetyNet [11] is an example of a hardware implementation. It inserts buffers near processor caches and memories to log changes in local processor memories as well as messages between processors. While very efficient (SafetyNet can take 10K checkpoints per second), SafetyNet requires changes to the system hardware and is therefore not portable. Furthermore, because it keeps its logs inside regular RAM or at best battery-backed RAM rather than some kind of stable storage, SafetyNet is limited in the kinds of failures it is capable of dealing with.

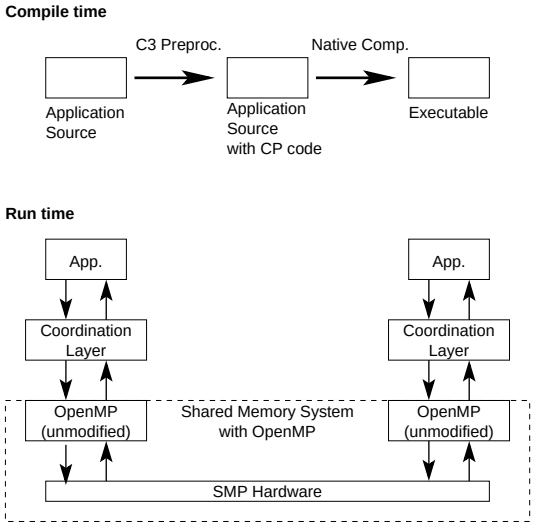

Figure 1: Overview of the C^3 system.

On the software side, Dieter et al. [4] and the *Berkeley Labs Linux Checkpoint/Restart* [5] provide checkpointing for SMP systems. Both approaches modify specific systems and are thus bound to them, rendering these solutions non-portable.

In addition, several projects have explored checkpointing for software distributed shared memory (SW-DSM) [8, 9]. They are all implemented within the SW-DSM system itself and exploit internal information about the state of the shared memory to generate consistent checkpoints. They are therefore also bound to a particular shared memory implementation and do not offer a general and portable solution.

3. OVERVIEW OF APPROACH

Figure 1 describes our approach. The C^3 pre-compiler reads C/OpenMP application source files and instruments them to perform application-level saving of shared and thread-private state. The only modification that programmers must make to source files is to insert calls to a function called `potentialCheckpoint()` at points in the program where a checkpoint may be taken. Ideally, these should be points in the program where the amount of live state is small.

It is important to note that checkpoints do not have to be taken every time a `potentialCheckpoint()` call is reached; instead, a simple rule such as "checkpoint only if a certain quantum of time has elapsed since the last checkpoint" is used to decide whether to take a checkpoint at a given location. Checkpoints taken by individual threads are kept consistent by our coordination protocol.

The output of the pre-compiler is compiled with the native compiler on the hardware platform, and linked with a library that implements a coordination layer for generating consistent snapshots of the state of the computation. This layer sits between the application and the OpenMP runtime layer, and intercepts all calls from the instrumented application program to the OpenMP library. This design permits us to implement the coordination protocol without modifying the underlying OpenMP implementation. This promotes modularity, eliminates the need for

access to OpenMP library code, which is proprietary on some systems, and allows us to easily migrate from one OpenMP implementation to another. Furthermore, it is relatively straightforward to combine our shared-memory checkpointer with existing application-level checkpointers for MPI programs to provide fault tolerance for hybrid MPI/OpenMP applications.

3.1 Tool Usage

C^3 can be used as a pass before an application's source code is run through the system's native compiler. The process of generating a fault tolerant application can be broken down into several steps. This process is easily automated and can be hidden inside a script, in much the same way that the details of linking with an MPI library are often hidden inside a mpicc script.

- Use the native preprocessor to translate the original source code into its corresponding pure C form. This involves applying `defines`, resolving `ifdefs` and inserting into the source code the files specified by `include` statements.
- The resulting preprocessed files are then given to C^3, which instruments them in a way that allows them to record their own state.
- The instrumented fault-tolerant files are fed to the native C compiler and linked to the C^3 coordination layer that keeps track of the application's interactions with OpenMP and coordinates the threads' checkpoints

In practice a user would use a single script to do all of the above actions, providing a list of files to be compiled and receiving a fault tolerant executable in return.

3.2 Protocol

We use a blocking protocol to co-ordinate the saving of state by the individual threads. This protocol has three phases, shown pictorially in Figure 2.

1. Each thread calls a barrier.
2. Each thread saves its private state. Thread 0 also saves the system's shared state.
3. Each thread calls a second barrier.

We assume that a barrier is a memory fence, which is typical among shared memory APIs. It is easy to see that if the application does not itself use synchronization operations such as barriers, its input-output behavior will not be changed by using this protocol to take checkpoints. The only effect of the protocol from the perspective of the application is to synchronize all threads and enforce a consistent view of the shared state by using a memory fence operation (normally implemented implicitly within the barrier). This state may not be identical to the system's state had a checkpoint not been taken. However, it is a legal state that the system could have entered since all consistency models only define the latest point at which a memory fence operation can take place, not the earliest (that is, it is always legal to include an additional memory fence operation). Furthermore, it is obvious that the state visible to each thread immediately after the checkpoint is identical to the state saved in the checkpoint.

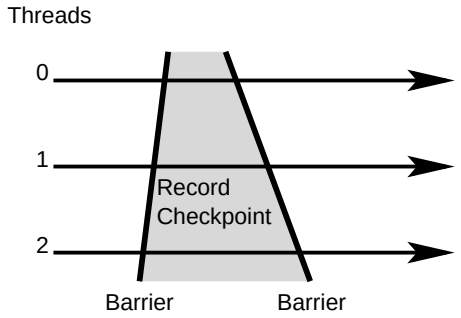

Figure 2: High-level view of checkpointing protocol

These properties ensure that we can restart the program by restoring all shared memory locations to their checkpointed values. Intuitively, if it was legal to flush all caches and set every thread's view of the shared memory to that memory image, then by restoring the entire shared address space to the image and flushing all the caches, we will return the system to an equivalent state.

The recovery algorithm follows from this, and is described below.

1. All threads restore their private variables to their checkpointed values and thread 0 restores all the shared addresses to their checkpointed values.
2. Every thread calls a barrier.
   This recovery barrier is necessary to make sure that the entire application state has been restored before any thread is allowed to access it.
3. Every thread continues execution.

Our protocol inserts additional barriers into the execution of the program and it is possible for these barriers to cross the application's own barriers and lock acquisitions. In such cases the checkpointing process may be corrupted or a deadlock may occur. To deal with this problem our protocol may force checkpoints to happen before the application's barriers and lock acquires, ensuring that no checkpoint conflicts with the application's causal interactions.

4. EXPERIMENTAL EVALUATION

Application-level checkpointing increases the running times of applications in two different ways. Even if no checkpoints are taken, the instrumented code executes more instructions than the original application to perform bookkeeping operations . Furthermore, if checkpoints are taken, writing the checkpoints to disk adds to the execution time of the program. In this section, we present experimental results that measure these two overheads for the C^3 system.

For our benchmark programs, we decided to use the codes from the SPLASH-2 suite [14] that we converted to run on OpenMP. We omitted the `cholesky` benchmark because it ran for only a few seconds, which was too short for accurate overhead measurement. We also omitted `volrend` because of licensing issues with the tiff library, and `fmm` because we could not get even the unmodified benchmark to run on our platforms.

One of the major strengths of application-level checkpointing is that the instrumented code is as portable as the original code. To demonstrate this, we ran the instrumented SPLASH-2 benchmarks on three different platforms: a 2-way Athlon machine running Linux, a 4-way Compaq Alphaserver running Tru64 UNIX, and an 8-way Unisys SMP system running Windows. In this section, we present overhead results on the first two platforms; we were not able to complete the experiments on the third platform in time for inclusion in this paper.

Benchmark	Problem size	Uninstrumented run time	C^3-instrumented run time 0 checkpoints taken	C^3-instrumentation overhead
fft	2^{24} data points	20s	20s	0%
lu-c	5000×5000 matrix	110s	110s	0%
radix	100,000,000 keys, radix=512	30s	31s	3%
barnes	16384 bodies, 15 steps	103s	106s	3%
ocean-c	514×514 ocean, 600 steps	162s	162s	0%
radiosity	Large Room	8s	8s	0%
raytrace	Car Model, 64MB RAM	32s	34s	6%
water-nsquared	4096 molecules, 60 steps	260s	223s	-14%
water-spatial	4096 molecules, 60 steps	156s	141s	-9%

Table 1: SPLASH-2 Linux Experiments

4.1 Linux/x86 Experiments

The Linux experiments were conducted on a 2-way 1.733GHz Athlon SMP with 1GB of RAM. The operating system was SUSE 8.0 with a 2.4.20 kernel. The applications were compiled with the Intel C++ Compiler Version 7.1. All experiments were run using both processors (i.e. P=2). Checkpoints were recorded to the local disk. The key parameters of the benchmarks used in the Linux experiments are shown in Table 1.

4.1.1 Execution Time Overhead

In this experiment, we measured the running times of (i) the original codes, and (ii) the instrumented codes without checkpointing. Times were measured using the Unix `time` command. Each experiment was repeated five times, and the average is reported in Table 1. From the spread of these running times, we estimate that the noise in these measurements is roughly 2-3%. The table shows that for most codes, the overhead introduced by C^3 was within this noise margin. For two applications, `water-nsquared` and `water-spatial`, the instrumented codes ran faster than the original, unmodified applications. Further experimentation showed that this unexpected improvement arose largely from the superior performance of our heap implementation compared to the native heap implementation on this system. We concluded that the overhead of C^3 instrumentation code for the SPLASH-2 benchmarks on the Linux platform is small, and that it is dominated by other effects such as the quality of the heap implementation.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Recovery
fft	765	43	22
lu-c	191	2	5
radix	768	43	24
barnes	569	4	10
ocean-c	56	1	4
radiosity	32	0	1
raytrace	68	0	2
water-nsquared	4	1	0
water-spatial	3	0	0

Table 2: Overhead of Checkpoint and Recovery on Linux.

4.1.2 Checkpoint and Recovery Overhead

Finally, we measured the execution time overhead of taking a single checkpoint and performing a single recovery. These numbers can be used in formulas containing particular checkpointing frequencies and hardware failure probabilities to derive the overheads for a long-running application.

To measure the overhead of taking a single checkpoint, we ran the C^3-transformed version of each benchmark without taking a checkpoint and compared its execution time to the time it took to run the same benchmark and taking a single checkpoint.

To measure the overhead of a single recovery, we first measure the time of execution from the start of the program until after the single checkpoint completes. Then we add to this the time measured from the beginning of a restart from this checkpoint to the end of the program. Finally, from this sum, we subtract the execution time for the complete program that takes a single checkpoint.

The results are shown in Table 2. The time to take checkpoints is fairly low for most applications, and is significant only for applications for which checkpoint sizes are very large (`fft` and `radix`). As mentioned before, these checkpoints were saved to local disk on the machine. If they were saved to a networked file system, we would expect the overheads to be larger.

4.2 Alpha/Tru64 Experiments

The Alpha experiments were conducted at the Pittsburgh Supercomputing Center on the Lemieux cluster. This cluster is composed of 750 Compaq Alphaserver ES45 nodes. Each node is an SMP with 4 1Ghz EV68 processors and 4GB of memory. The operating system is Compaq Tru64 UNIX V5.1A. All codes were run on all 4 processors of a single node (i.e. P=4). Checkpoints were recorded to system scratch space, which is a networked file system available from all nodes. The key parameters of the SPLASH-2 benchmarks used in the Alpha experiments are shown in Table 3.

4.2.1 Execution Time Overhead

We measured the overheads of instrumentation on Lemieux using the same methodology we used for Linux. Table 3 shows the results.

These results show that except for `radix` and `ocean-c`, the overheads due to C^3's transformations are either negligible or negative. The overheads in `radix` and `ocean-c` arise from two different problems that we are currently addressing.

The overhead in `radix` comes from some of the details of how C^3 performs its transformations. Our state-saving mechanism computes addresses of all local and global variables, which may prevent the compiler from allocating these variables to a register. For `radix`, it appears that this inability to register-allocate certain variables leads to a noticeable loss of performance. We are currently re-designing the mechanism to circumvent this problem.

Our experiments also showed that the overhead in `ocean-c` execution comes from our heap implementation

Benchmark	Problem size	Uninstrumented run time	C^3-instrumented run time 0 checkpoints taken	C^3-instrumentation overhead
fft	2^{26} data points	68s	67s	-2%
lu-c	12000×12000 matrix	719s	724s	1%
radix	300,000,000 keys, radix=512	61s	70s	15%
ocean-c	1026×ocean, 600 steps	153s	183s	20%
radiosity	Large Room	13s	12s	-9%
raytrace	Car Model, 1GB RAM	20s	20.4s	2%
water-nsquared	12167 molecules, 10 steps	136s	140s	3%
water-spatial	17576 molecules, 40 steps	214s	218s	2%

Table 3: Characteristics and Results of SPLASH-2 Alpha Experiments

(replacing our heap implementation with the native heap eliminated this overhead). While this implementation has been optimized for Linux, it is not as optimized for Alpha. This tuning is underway.

4.2.2 Checkpoint and Recovery Overhead

Table 4 shows the checkpoint time and the recovery time for the different applications. It can be seen that there is a correlation between the sizes of the checkpoints and the amount of time it takes to perform the checkpoint. In these experiments, the checkpoint files were written to the system scratch space rather than to a local disk, so for codes that take larger checkpoints, the overheads observed on Lemieux are higher than the overheads on the Linux system shown in Table 2.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Recovery
fft	3074	363	32
lu-c	1103	136	7
radix	2294	285	36
ocean-c	224	68	*
radiosity	43	8	1
raytrace	1033	137	7
water-nsquared	16	3.75	388
water-spatial	12	3.5	17

Table 4: Overhead of each checkpoint and recovery on Alpha.

The only code with a high recovery overhead is `water-nsquared`, and it highlighted an inefficiency in our current implementation. Note that `water-nsquared` takes 3.5 seconds to record a 16MB checkpoint but takes 388 seconds to recover. The reason for this is that `water-nsquared` `malloc()-s` a large number of individual objects: 194K. This in comparison to the 18K objects that `water-spatial` allocates or the 65K allocated by `water-nsquared` given the input parameters used on Linux. C^3's checkpointing code is optimized to use buffering when writing these objects to a checkpoint, but its recovery code does not have such optimizations, so it performs one file read for every one of these objects. The cost of that many file reads, even to buffered files is very high and results in a long recovery time. Our next implementation of the C^3 system will optimize reading the checkpoint files to eliminate this inefficiency.

`Ocean-c`'s recovery overhead was measured to be negative. However this negative overhead was within the variability of the timing results in this experiment, so it appears to be an artifact of the fluctuations inherent to a networked file system.

4.3 Discussion

When we began this work, we invested considerable time in refining our coordination protocol because we thought that the execution of the protocol would increase the running time of the application significantly. Indeed, much of the literature on fault-tolerance focuses on protocol optimizations such as reducing the number of messages required to implement a given protocol.

Our experiments showed that the overheads are largely due to other factors, summarized below.

- The performance of some codes is very sensitive to the memory allocator. Overall, we obtained good results on the Linux system because we have tuned our allocator for this system; on Lemieux, where the tuning work is still ongoing, some codes such as `ocean-c` had higher overheads.
- The instrumentation of code to enable state-saving prevents register allocation of some variables in codes like `radix` on Lemieux. This is relatively easy to fix by introducing new temporaries, and it is being implemented in our preprocessor.
- For codes that produce large checkpoint files, the time to write out these files dominates the checkpoint time. We are exploring incremental checkpointing, as well as compiler analysis, to reduce the amount of saved state.
- Finally, recovery time for codes that create a lot of small objects, such as `water-nsquared` on Lemieux, needs to be reduced by better management of file I/O.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented an implementation of a blocking, coordinated checkpointing protocol for application-level checkpointing (ALC) of shared-memory programs using locks and barriers. The implementation has two components: (i) a pre-compiler that automatically instruments C/OpenMP programs so that they become self-checkpointing and self-restarting, and (ii) a runtime layer that implements the co-ordination protocol. Experiments with SPLASH-2 benchmarks show that the overheads introduced by our implementation are small. The implementation can be used to checkpoint shared-memory programs; it can also be used in concert with a system for checkpointing message-passing programs, such as [2, 1, 12], to provide a solution for checkpointing hybrid message-passing/shared-memory programs.

Our ALC approach has the advantage that programs instrumented by our pre-compiler become self-checkpointing and self-restarting, so they become fault-tolerant in a

platform-independent manner. This is a major advantage over system-level checkpointing approaches, which are very sensitive to the architecture and operating-system. We have demonstrated this platform-independence by running on a variety of platforms.

In the future, we intend to extend (C^3) to deal with a broader set of shared-memory constructs. In particular, we intend to support the full OpenMP standard. Furthermore, we intend to couple (C^3) with the MPI checkpointer described in [1] to produce a fault tolerance solution for programs using both message-passing and shared-memory constructs.

6. REFERENCES

[1] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *Proceedings of the 2003 International Conference on Supercomputing*, pages 234–243, June 2003.

[2] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 84–94, June 2003.

[3] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *IEEE Transactions on Computing Systems*, 3(1):63–75, 1985.

[4] W. Dieter and Jr. J. Lumpp. A user-level checkpointing library for POSIX threads programs. In *Proceedings of 1999 Symposium on Fault-Tolerant Computing Systems (FTCS)*, June 1999.

[5] J. Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. http://www.nersc.gov/research/FTG/checkpoint/reports.html.

[6] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[7] T. Tannenbaum J. B. M. Litzkow and M. Livny. Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. Technical Report Technical Report 1346, University of Wisconsin-Madison, 1997.

[8] Angkul Kongmunvattan, S. Tanchatchawal, and N. Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *Proceedings of the International Conference on Distributed Computer Systems (ICDCS 2000)*, 2000.

[9] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Symposium on Principles of Distributed Computing Systems (PDCS)*, 1994.

[10] OpenMP Architecture Review Board. *OpenMP C and C++ Application, Program Interface*, Version 1.0, Document Number 004–2229–01 edition, October 1998. Available from http://www.openmp.org/.

[11] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA 2002)*, July 2002.

[12] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of International Parallel Processing Symposium(IPPS)*, 1996.

[13] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996. Also available at `http://citeseer.nj.nec.com/stellner96cocheck.html`.

[14] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture 1995*, pages 24–36, June 1995.