Application-level Checkpointing for Shared Memory Programs

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.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming, Parallel Programming

General Terms: Reliability, Experimentation

Keywords: Fault-tolerance, Checkpointing, Shared-memory Programs, OpenMP

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.

Fault tolerance has been studied extensively by the distributed systems community. However, the goal in that context is to ensure high availability of critical systems like air traffic control systems and web servers, so most solutions rely on some form of redundancy. The goal of fault toler-

*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.

ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.

ance in the context of high-performance computing is different - it is to minimize the expected time to completion of the program, given some probability of hardware failure. High real-time availability is not a goal, so redundancy is not the right approach to finding good solutions.

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[1]. Most existing systems for checkpointing such as Condor [14] 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; for example, the Condor documentation states that "Linux is a difficult platform to support...The Condor team tries to provide support for various releases of the Red Hat distribution of Linux [but] we do not provide any guarantees about this." [8]. 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 [24] including parallel regions, locks, and barriers. Since OpenMP is the dominant standard for shared-memory programming, this choice provides us access to a large number of different experimental platforms. 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). Of course, our approach is not tied to OpenMP and is applicable to any shared-memory programming model or API.

The system described here builds on our previous work on ALC for message-passing programs [5, 4]. 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. There are two main problems: (i) how do threads save their local states and the global state, and (ii) how do we coordinate state-saving between different threads? These questions are addressed in Sections 4 and 5 respectively. In Section 6, we present experimental results. Finally, we discuss ongoing work in Section 7.

2. PRIOR WORK

Alvisi et al. [11] 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. Almost all of this work uses system-level checkpointing, but there are major differences in the protocols used to co-ordinate processes for taking checkpoints. *Blocking* techniques bring all processes to a stop before taking a global checkpoint. Hardware blocking was used on the IBM SP-2 to take system-level checkpoints. Software blocking techniques take checkpoints when processes reach a global barrier [27]. In *non-blocking* checkpointing, a global coordination protocol implemented by exchanging control messages is used to orchestrate the state saving of individual processes. Usually, a distinguished process called the *initiator* is responsible for initiating and monitoring the protocol; processes communicate with other processes to co-ordinate the taking of checkpoints but make no assumptions about the states of other processes. The Chandy-Lamport protocol is perhaps the most well-known non-blocking co-ordination protocol [6].

In the high-performance computing applications community, hand-coded application-level checkpointing at global barriers is the norm. The Dome project explored hand-coded ALC within the context of an object-oriented language for computational science applications [3]. Recently, our research group has pioneered preprocessor-based approaches for implementing ALC (semi-)automatically [5, 4]. In addition to showing that existing SLC protocols like the Chandy-Lamport protocol do not work with ALC, we have designed and implemented novel protocols that do.

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 [25] is an example of a

[1]Strictly speaking, CPR provides a solution only for *fail-stop* faults, a fault model in which failing processors just hang without doing harmful things allowed by more complex *Byzantine* fault models in which a processor can send erroneous messages or corrupt shared data [16]

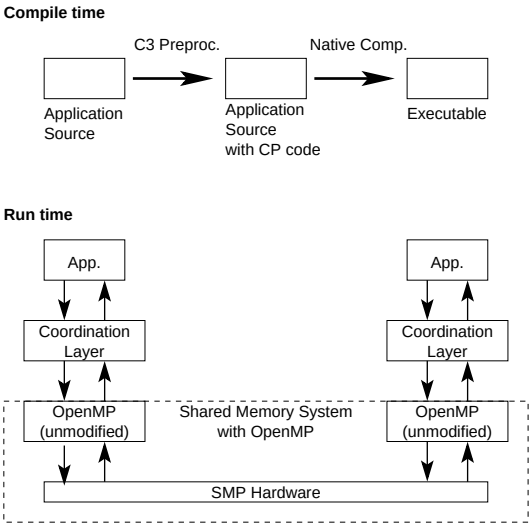

Figure 1: Overview of the C^3 system.

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. ReVive [19] is another approach to hardware shared memory fault tolerance. Based on a combination of message logging and checkpointing it also provides high efficiency at the cost of portability.

On the software side, Dieter et al. [9] and the *Berkeley Labs Linux Checkpoint/Restart* [10] provide checkpointing for SMP systems. The former approach augments the native thread library to coordinate checkpoints across the machine and implements a special protocol for synchronization primitives, similar to the one presented in this paper. The latter system uses dynamically loadable kernel modules to directly control the thread scheduler and force consistency among all threads. In contrast to our solution, however, both approaches are bound to a particular thread library and kernel version, are non-portable, and require root privileges for their installation.

In addition, several projects have explored checkpointing for software distributed shared memory (SW-DSM) [15, 23, 7, 13, 28]. 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 coordinated by the protocol described in Section 3.1.

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 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 (we discuss synchronization operations in Section 5). The only effect of the protocol from the perspective of the application is to synchronize all threads and enforce a consistent view

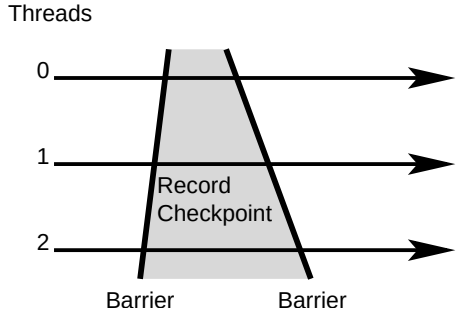

Figure 2: High-level view of checkpointing protocol

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.

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.

Having discussed the overall checkpointing approach, we now deal with two remaining questions.

1. How do we identify and save all the private and shared state in the system? This is addressed in Section 4.

2. How do we deal with synchronization constructs such as locks and barriers in the application program? This is addressed in Section 5.

4. SAVING STATE

The application-level state of a program consists of its heap, global variables, local variables, and call stack. OpenMP distinguishes between private and shared variables. Private variables are local to a thread, so they can only be saved by that thread. Shared variables on the other hand are visible to all threads, so they can be saved by any thread. In OpenMP, the heap is always shared while local and global variables may be either shared or private. The call stack of each thread is always private.

In saving the application state, C^3 follows an approach similar to that taken by Dome [1] and used in our previous work on distributed-memory checkpointing [4, 5]. The checkpoints produced by this method are not portable in the sense that they cannot be restarted on a different architecture or configuration; there is ongoing work on this in our group.

4.1 Heap

To save the heap, we use our own implementation of the heap library to keep track of all dynamic memory allocations and de-allocations. At checkpoint time, any thread can identify and save exactly those portions of the heap that are currently allocated to the application. On recovery, the library restores heap locations to the values saved in the checkpoint. Heap objects are restored to their original addresses, so the problem of relocating pointers does not arise.

4.2 Call Stack

Saving the state of the call stack is more intricate because the position in the program where the checkpoint was taken has to be recorded without reference to machine-specific constructs like program counters. To accomplish this, we use a special `pc_stack`. At the start of the program, the `pc_stack` is empty. Before every function call from which a `potentialCheckpoint()` call can be reached, the compiler inserts an instruction to push the unique ID of the call onto the `pc_stack`, followed by a goto label. Each function call is also followed by an instruction to pop the call ID off the `pc_stack`. Thus, at any given point in time the `pc_stack` will contain the chain of calls that led the program's execution from `main()` to the current function. An example of this transformation is shown in Figure 3.

On restart, the `pc_stack` is used to restore the original call stack. At the top of each function, the compiler places code that looks at the position in the `pc_stack` corresponding to the current function. The value at that position identifies the function call that was made inside this function. The code then jumps to the label that was placed right before that function call, skipping all the work that was done in the body of the function up to that point and perform the call. This process is repeated in each function until control is returned to the `potentialCheckpoint()` call where the checkpoint was taken. When the `potentialCheckpoint()` call return, the call stack will look just as it did at the time of the checkpoint.

This approach works for single-threaded applications.

```
func1() {
   target = read_pc_stack();
   switch(target) {
      case 0: goto label_0;
      case 1: goto label_1;
      ...
   }
   ...
   push(pc_stack,0);
label_0:
   func2();
   pop(pc_stack);
   ...
   push(pc_stack,1);
label_1:
   omp_set_num_threads(read_original_num_threads())
   #pragma omp parallel
      { parallel code }
   pop(pc_stack);
   ...
}
```

Figure 3: Recording the dynamic call sequence.

However, shared memory applications use multiple threads, and in restoring the call stack, the system needs to restore the threads themselves. In OpenMP, the syntax for thread creation uses a pragma to declare that a particular block of code should be run in parallel, and a call to `omp_set_max_threads()` to specify the number of threads to be used. Thus, to recreate threads on recovery, we treat these parallelization pragmas just like function calls. We precede them with a `pc_stack` push and a label, and follow them by a `pc_stack` pop, as shown in Figure 3. Furthermore, `omp_set_max_threads()` is used to ensure that the number of threads on recovery is the same as the number of threads during the original execution. Finally, since each thread has its own call stack, a separate `pc_stack` needs to be maintained for each thread.

4.3 Local Variables

Although the algorithm described above restores the function calls on the call stack, the local variables within those functions are left uninitialized. Therefore, technique is needed to save and restore the variables themselves. Since the variables are stored on the call stack, we C^3 uses a stack of "stack value descriptors", called the `svd_stack`. There is one `svd_stack` for each thread. A stack value descriptor contains the variable's size as well as a pointer to it.

At the start of every function from which a `potentialCheckpoint()` can be reached, the compiler inserts code to push the descriptor (size and pointer) of each function argument and all the local variables onto the `svd_stack`. At the end of each function, all the previously pushed descriptors are popped. Thus, at any given point in the program's execution, the `svd_stack` contains the addresses and sizes of all the local variables of the functions above the current point in the call stack. At checkpoint time, these addresses and sizes are used to access the values of all the local variables for the purpose of checkpointing them. An example of this transformation is shown in Figure 4.

In OpenMP, only local variables common to all threads can be declared as shared. Therefore, thread 0 will have every shared local variable on its `svd_stack`, and it is responsible for saving them.

On restart, after the system has recreated the call stack and control returns to the original call to `potentialCheckpoint()`, the restored `svd_stack` is used to restore the state of all the local variables. Note that if the call stack starts at the same address as it did during the original execution, the addresses of all local variables will be the same as before, thus ensuring that all of the program's original pointers are still valid. While the OpenMP standard does not guarantee that thread stacks start at specific locations, it has been our experience that in most implementations stack start points do not differ by more than several words between different executions of the same program. Therefore, we provide the illusion of an unchanged stack start address by using `alloca()` to pad the start of the stack with a variable amount of data depending on where the thread stack is placed by the operating system.

Global variables are handled in a similar manner. Because they are available throughout the execution of the program, it is possible to push them onto the `svd_stack` at the start of `main()`. Local function variables that have been declared static have lifetimes identical to those of global variables. Thus, during preprocessing the compiler turns them into global variables and checkpoints them accordingly.

```
func1(char q){
   int x;
   push(svd_stack, &q, sizeof(char));
   push(svd_stack, &x, sizeof(int));
   ...
   func2();
   ...
   pop();
   pop();
}
```

Figure 4: Recording the location of stack variables.

5. SYNCHRONIZATION CONSTRUCTS

A blocking protocol like the one described in Section 3.1 must ensure that its use does not introduce deadlocks into the program. Deadlock may occur if synchronization constructs like locks and barriers are used, because a thread may be blocked until some synchronization condition has been met, but all other threads may be waiting for that thread at a barrier, wanting to take a checkpoint.

In some situations, the checkpoint protocol itself may fail. Figure 5 shows an example of this problem, created when an application barrier crosses a checkpoint region. In this figure, threads 1 and 2 take a checkpoint before they encounter the application barrier, whereas thread 0 reaches this barrier before it reaches the next potential checkpoint location. In this situation, threads 1 and 2 will reach their first checkpoint barrier while thread 0 is waiting on its application barrier. In OpenMP, any barrier call on a thread will match a barrier call on another thread even if the two barriers are not in the same locations in the source code. As a result, all three threads will pass their respective barriers. However, while threads 1 and 2 record their checkpoints, thread 0 will continue computing, potentially polluting the checkpoint.

One approach to addressing these problems is to make this the responsibility of the programmer. This is possible as long as the system uses a deterministic algorithm to decide where to checkpoint (recall that in our approach, `potentialCheckpoint()` calls only indicate places where checkpoints *may* be taken). While this solution is simple, our goal is to provide the programmer with as automated a solution as possible. Therefore we have developed protocols to deal automatically with these problems. These are described next.

5.1 Barriers

The problem with barriers is actually deeper than is illustrated in Figure 5. According to the OpenMP spec (as well as other shared memory APIs), no thread may go past a barrier until every other thread has at least reached the barrier. Even if we could somehow resolve the problem described above, consider what would happen upon recovery. Thread 0 would recover in a state after the application barrier while threads 1 and 2 would recover in a state before the barrier, which is a violation of OpenMP barrier semantics.

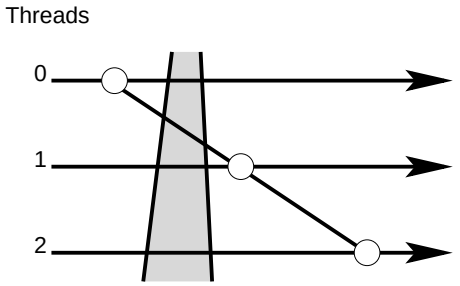

Figure 5: Checkpoint line crossed by a barrier.

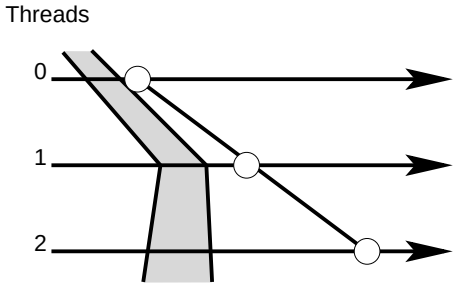

Figure 6: Resolving Barrier/Checkpoint Conflict.

```
ccc_barrier(){
   #pragma omp barrier
   while(checkpointFlag){
      // only do this if checkpoint started while
      // waiting on application barrier
      save application state
      checkpointFlag=FALSE
      #pragma omp barrier

      // trying to wait on application barrier again
      #pragma omp barrier
   }
}

potentialCheckpoint(){
   // update checkpointFlag
   #pragma omp flush(checkpointFlag)
   // if time to checkpoint or others checkpointed
   if (checkpointFlag or initiateCheckpoint()){
      checkpointFlag = true;
      #pragma omp barrier
      save application state
      checkpointFlag = FALSE
      #pragma omp barrier
   }
}
```

Figure 7: Pseudocode for barrier implementation.

The solution is to ensure that no checkpointing region ever crosses an application barrier. This is done by associating a `potentialCheckpoint()` call with every call to an application barrier. Intuitively, the idea is to make sure that if a situation like the one in Figure 5 occurs, thread 0 gets to know that it is at an application barrier while some other thread wants to take a checkpoint. We must ensure that thread 0 immediately takes a checkpoint and then goes back to waiting at the application barrier.

One problem in implementing this approach is that by the time another thread decides to take a checkpoint, thread 0 may already be blocked on its application barrier. The only way to get it to notice the ongoing checkpointing is to return control to the thread, which can only be done by allowing it to pass the barrier.

For this purpose, we introduce a global `checkpointFlag` variable to inform threads of the fact that a checkpoint is ongoing. This flag is initialized to FALSE. When a thread decides to take a checkpoint (in the pseudo-code of Figure 7, we assume this decision is encapsulated in a function called `initiateCheckpoint()`, which returns true if the thread should take a checkpoint), it first sets the global `checkpointFlag` to TRUE and then calls the first checkpoint barrier. Once all threads have either reached their first checkpoint barriers or an application barrier, all threads will be released from their respective barriers. Each thread that was trying to pass an application barrier now looks at the `checkpointFlag`. If it is set to TRUE, then at least one thread must have begun the global checkpoint. If it is FALSE, then all threads must have passed an application barrier and no checkpoint has been requested by any thread.

If the `checkpointFlag` is discovered to be TRUE, then any thread that has not already begun a checkpoint must begin one immediately. Fortunately, since it has already passed a barrier, this barrier can be treated as the first checkpoint barrier and the thread must simply execute the remainder of the checkpoint protocol: record its portion of the checkpoint and call the second checkpoint barrier.

Once the checkpoint has been recorded, the `checkpointFlag` variable is reset to FALSE by all threads before the second barrier is passed (indeed, it is sufficient for any one thread to reset the flag). Threads resume work after passing the second checkpoint barrier. In particular, threads that were originally waiting on application barriers before they were forced to take a checkpoint go back to waiting on these barriers until all threads reach an application barrier or until another global checkpoint is started in which case the process described above is repeated.

The pseudocode for `potentialCheckpoint()` and the code that we use to replace OpenMP barriers is given in Figure 7.

This protocol ensures that no global checkpoint is ever crossed by an application barrier. In particular, the problem shown in Figure 5 would be solved automatically as shown in Figure 6. Thread 0 will take a checkpoint after waking up from its barrier and realizing that other threads have begun the global checkpoint. After taking a checkpoint, thread 0 once again synchronizes at a barrier and this time only wakes up after threads 1 and 2 have both reached an application barrier.

5.2 Locks

Locks present synchronization problems similar to those of barriers. Furthermore, locks introduce an additional complication: if a thread was holding a lock at checkpoint time,

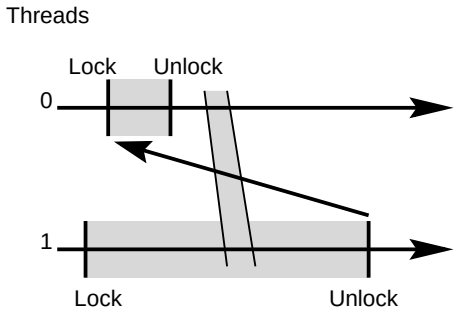

Figure 8: Checkpoint line crossed by a lock acquisition.

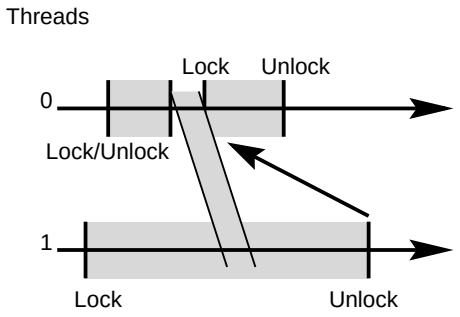

Figure 9: Resolving Lock/Checkpoint Conflict.

the protocol has to ensure that that thread is holding the same lock on recovery.

An example of the general problem is shown in Figure 8. Thread 1 has acquired a lock and wishes to take a checkpoint before releasing this lock. Meanwhile, thread 0 wants to acquire the same lock before it takes a checkpoint. Unfortunately, thread 1 already has this lock and is waiting for thread 0 to reach its first checkpoint barrier before it can release the lock, resulting in deadlock. Notice that if thread 0 could be forced to take a checkpoint just before it tries to acquire the lock, there would be no deadlock.

To resolve these problems, we employ a technique similar to the one described above for barriers. We associate a `lockCheckpointFlag` with every lock. Before a thread calls the first barrier of its checkpoint, it sets the `lockCheckpointFlag` of every lock that it is holding to TRUE and then releases all of its locks, taking care to remember which locks it was originally holding. The release of locks will unblock any thread that is waiting for one of these locks, such as thread 0 above.

Upon lock acquisition, each thread checks the value of the lock's `lockCheckpointFlag`. If the flag is FALSE, it knows that the lock was acquired because the application itself had released it and no further action is required. However, if the flag is TRUE, the thread knows that it must have acquired the lock not because it was released by the application but because the lock owner decided to take a checkpoint. In this case, the acquiring thread must also take a checkpoint, so it performs the following actions.

- It releases the lock it just acquired.
- It sets the lock flags for its locks to TRUE.
- It releases its locks, noting which locks it was holding.

```
ccc_set_lock(lock){
   omp_set_lock(lock)
   while(lock.lockCheckpointFlag){
      // only do this if checkpoint started while
      // waiting to acquire lock
      #pragma omp barrier
         for all held locks
            lock.lockCheckpointFlag=TRUE
         record which locks are being held
         release all locks

         save application state
         save lock state
         for all locks that were held
            reacquire lock
            lock.lockCheckpointFlag=FALSE
      #pragma omp barrier

      // try to acquire the lock again
      omp_set_lock(lock)
   }
}

potentialCheckpoint(){
   #pragma omp barrier
      for all held locks
         lock.lockCheckpointFlag=TRUE
      remember which locks are held
      release all locks

      save application state
      save lock state
      for all locks that were held
         reacquire lock
         lock.lockCheckpointFlag=FALSE
   #pragma omp barrier
}
```

Figure 10: Pseudocode for lock implementation.

- It executes the protocol to checkpoint its state.

This process of potentially forcing checkpoints before acquiring locks prevents deadlocks. In our example, Figure 8 would be transformed into Figure 9.

When taking a checkpoint, each thread records the locks it is currently holding. It uses this information on recovery to reacquire its locks. Furthermore, before returning control to the application, the coordination protocol must restore the state of all locks as follows.

- All lock flags are set to FALSE.
- Each thread reacquires the locks that it had before the checkpoint.

The second step appears to have potential for deadlock. However, note that after the first checkpoint barrier is passed, it is known that all the locks that were previously held by some thread have been released by their owners. Thus, when individual threads reacquire the locks that they held before the checkpoint, no deadlock can occur.

The resulting pseudocode for locks is given in Figure 10.

One remaining synchronization concern regarding locks is that in OpenMP there are in fact two ways to acquire a lock: the blocking acquire provided by `omp_set_lock()` as well as a non-blocking acquire provided by `omp_test_lock()`. `omp_test_lock()` simply checks if the lock is available, acquiring it if it is, and returning a notice if it is not. The fact that `omp_test_lock()` is guaranteed to return within a bounded amount of time guarantees that an individual call to `omp_test_lock()` will not cause deadlock. However, it is possible to use multiple such calls to implement a blocking lock acquire, which may cause a deadlock in the manner described above. A loop that keeps calling `omp_test_lock()` until it finally acquires the lock is an example of this.

While it is not possible to detect all such code patterns at compile-time, this detection is hardly necessary. If after calling `omp_test_lock()` a thread discovers that the lock in question has had its `lockCheckpointFlag` set to TRUE, it can force a checkpoint immediately in the manner described above. If the call to `omp_test_lock()` did not acquire the lock, it skips the step where the lock would be released. Thus, if `omp_test_lock()` really is being called repeatedly, this forcing of checkpoints before `omp_test_locks` will avoid deadlocks just like it did for `omp_set_lock()`.

While this algorithm is correct, it can be too conservative when it comes to applications that do not use `omp_test_lock()` to implement a blocking lock acquire. In such applications checkpoints would be forced when they are not necessary, potentially paying a price by taking a checkpoint at a suboptimal location (which may have more state than really needs to be saved). Therefore, our compromise solution is to set a constant `c` such that if a thread calls `omp_test_lock()` unsuccessfully on the same lock more than `c` times in a row, then it will check its `lockCheckpointFlag` and force a checkpoint if `lockCheckpointFlag` turns out to be TRUE. Otherwise, the application is allowed reach its next checkpoint location on its own.

5.2.1 Managing OpenMP Lock Objects

Another problem that must be addressed is that in OpenMP, locks at the application level are simply pointers to objects of type `omp_lock_t`. On recovery, the system needs to recreate all the locks that existed at checkpoint time. However if it merely tried calling `omp_init_lock` to recreate the locks, there would be no guarantee that the OpenMP implementation would place these locks at the same locations in memory.

To solve this problem, we need to add a layer of indirection between the application and OpenMP's lock handling routines. This layer wraps each lock object in a new struct that contains a unique ID, a pointer to the corresponding OpenMP lock object and a recovery number that indicates how fresh the lock pointer is. Every time the application restarts, a global recovery number gets incremented. When a lock function is used, the lock's recovery number is compared to the global recovery number and if they are different, the lock's unique ID is looked up in a table to get the current pointer to the lock. The lock's wrapper struct is then updated with the new recovery number and OpenMP lock pointer. This up-to-date lock pointer in then used in future interactions with OpenMP's lock management routines. In this way, the system can allow OpenMP to create brand new locks on recovery while providing the application with the illusion of persistent locks.

```
// waiting thread:
#pragma omp flush
while(flag==FALSE)
{
   #pragma omp flush
}

// releasing thread:
flag = TRUE;
#pragma omp flush
```

Figure 11: Example of a spinlock.

5.3 Spinlocks

The last method for creating a dependency between two threads is by using spinlocks implemented via shared reads and writes. In OpenMP, such a spinlock might be implemented as shown in Figure 11. In this example, the waiting thread keeps looping until the releasing thread sets a flag to TRUE. `#pragma omp flush` is simply OpenMP's version of the local memory fence common to many shared-memory APIs and is used here to ensure that the write on the releasing thread is visible at the waiting thread.

As in the lock example of Section 5.2, the introduction of a checkpoint can lead to deadlock. However, the problem of detecting when a piece of code implements such a spinlock is extremely difficult. Furthermore, any runtime detection scheme would need to monitor individual reads and writes, resulting in significant overheads. Therefore, our checkpointing protocol does not support applications that use spinlocks implemented via shared reads and writes. This is a restriction that would be shared by any application-level blocking protocol that does not track individual reads and writes.

5.4 Alternatives

We have discussed the use of forcing checkpoints before synchronization operations such as barriers and lock acquires to avoid deadlocks. A number of alternate approaches are possible. Though we did not choose to implement them, we will discuss them here to provide some perspective on the solution space.

5.4.1 Aborting checkpoints

Suppose we have a situation as in Section 5.1. As we discussed, our protocol does not permit the recovery line to cross the application barrier, so the solution was to force checkpoints in order to push the checkpoint line before the barrier. In the checkpoint abort algorithm, the checkpoint line gets pushed after the application barrier. Instead of forcing a checkpoint on thread 0, this algorithm aborts the checkpoint on threads 1 and 2. This can be implemented by an algorithm only a little more complex than that used for checkpoint forcing.

The advantages of the checkpoint abort algorithm are that (i) it never forces checkpoints at inconvenient points for a thread, such as when the application state is large, and (ii) it requires placing fewer additional `potentialCheckpoint()` calls in the code. However, it may require the application to unnecessarily wait on checkpoints that will be aborted and

it cannot guarantee that a checkpoint will ever be taken since it is possible for the potential checkpoint locations to be arranged in such a way that any choice of checkpoint locations will cross an application barrier.

5.4.2 Serialization

Another alternative is a protocol that allows recovery lines to straddle barriers and lock dependences. The resulting inconsistency is resolved by serializing the execution of all threads so as to make their updates to the shared address space deterministic.

An outline of the serialization solution for barriers is as follows. When all threads block on either a checkpoint barrier or an application barrier, the threads that want to checkpoint are allowed to record their private state and thread 0 records the shared state. The threads that were sitting at their checkpoints are now allowed to advance one at a time up to their next application barrier (we call them the pre-barrier threads). At this point, all threads have reached the application barrier but some still have not reached their next checkpoint location, where they could record their private state. Thus, all remaining threads are executed one at a time until each one reaches its next checkpoint location (we call these the post-barrier threads). When all these remaining threads have recorded their private state, thread 0 records all the pieces of the shared state that changed since the time when it was recorded at the beginning of the checkpoint.

On recovery, the system restores the original shared state and then has each pre-barrier thread execute one at a time up to the barrier. At this point, all the pre-barrier threads will have reached the barrier and the protocol now needs to bring the shared state to its configuration at the time when the post-barrier threads took their checkpoints. This is done by applying the changes to the shared state made by the post-barrier threads when they did their serial execution. At the end, the application recovers to a state where all the pre-barrier threads are at the barrier and all the post-barrier threads are at their post-barrier checkpoints, a consistent and valid configuration.

While this protocol has the advantage of no inconveniently forced checkpoints or wasted time due to aborted checkpoints, it does have the problem of slowing the system down by a factor of n (where n is the number of threads) for the duration of the execution of the protocol.

6. 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 such as maintaining the `pc_stack` and the `svd_stack`. 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 OpenMP codes from the SPLASH-2 suite [29]. 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.

benchmark	Checkpoint location
fft	in `FFT1D()`, before FFT on each column
radix	in `slave_sort()`, after each barrier
lu-c	at end of `lu()` outermost loop
barnes	in `SlaveStart()` after each time step
ocean-c	in `slave()` after every step
radiosity	in `process_tasks()` before every task
raytrace	in `RayTrace()` before every job bundle
water-nsquared	in `MDMAIN()` at the end of each time step
water-spatial	in `MDMAIN()` at the end of each time step

Table 1: Characteristics of SPLASH-2 Benchmarks

Table 1 describes the locations in the other SPLASH-2 codes where we placed `potentialCheckpoint()` calls. In our studies, the longest-running code from this set finished execution in about 5 minutes, so hardware failures are not an impediment to running these codes in practice. Nevertheless, the SPLASH-2 suite is believed to illustrate the behavior of scientific codes, so the measured overheads for these code give some insights into the overheads that would be seen by users of the C^3 system.

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. We also performed some comparative studies with the Berkeley Checkpoint/Restart system (BLCR) [10], which is a system-level checkpointing system. However, this system runs only on Linux, so the comparisons are restricted to that platform.

6.1 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 2.

6.1.1 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 2. 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-squared` 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

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 2: SPLASH-2 Linux Experiments

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.

6.1.2 Checkpoint Sizes

The next set of experiments measured the sizes of checkpoints, the overheads of saving checkpoints to disk, and the time to recover from failure. We also compared our results with the BLCR system-level checkpointing system.

To understand the experimental set-up, it is necessary to note a peculiarity of the SPLASH-2 benchmarks. One feature of these benchmarks is that most memory is allocated near the beginning of execution, and then written to over the rest of the execution. For example, `barnes` and `radix` allocate 130MB and 765MB of memory respectively at the start of execution, but do not write to all of this memory until they near the end of their execution. On Linux, these initial calls to `malloc` do not allocate any physical pages but merely reserve address space; a physical page is allocated for a logical page only when that logical page is written to for the first time. Since BLCR is designed to work only with Linux, it optimizes state-saving by saving only those pages that were actually allocated to the application by the kernel. In particular, for some of the SPLASH benchmarks, this means that the size of the checkpoint taken by BLCR will increase if the checkpoint is taken later in the execution of that program. The C^3 system on the other hand checkpoints all the data `malloc()`-ed by an application regardless of whether it has actually been touched by the application.

Therefore, for the C^3 runs, we set the timer so that each application checkpointed mid-way through its execution. When using BLCR, we manually initiated checkpoints at various times during each benchmark's execution because it is difficult to take system-level checkpoints at precise points in the execution of a parallel application.

The resulting checkpoint sizes are shown in Table 3. Where a range of checkpoint sizes is reported for BLCR, it means that the checkpoint sizes increased throughout the execution of the program from the lower number to the higher number.

For most of the codes, the difference between the checkpoint sizes of the two systems is minimal. The codes for which this is not true, such as `radix` and `barnes`, are codes which write over an extended period of time to memory that is allocated upfront, as described above. We find these results encouraging because we have not devoted any energy so far in our project to reducing the size of the saved state. Our ongoing work towards this goal is exploring two approaches.

- Currently the C^3 system takes full checkpoints. We are incorporating incremental checkpointing into our system, which will permit the system to save only those data that have been modified since the last checkpoint. This mechanism has many advantages; among other things, it will address the inefficiency vis-a-vis BLCR, highlighted by the results in Table 3.

- We are also investigating the use of compiler techniques to exclude some data from being saved at a checkpoint because it can be recomputed during recovery. This is in the spirit of Beck et al, who have explored the use of programmer directives towards this end [17].

6.1.3 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. The difference is the number of seconds it takes to take a checkpoint, which includes not only the time to write the data to disk but also miscellaneous effects such the impact that taking a checkpoint has on future cache behavior.

To measure the overhead of a single recovery, we first measure the time of execution from the start of the program util after the single checkpoint completes (the program is 'killed' after this checkpoint). 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 3. 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.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Recovery	BLCR Checkpoint Size (MB)
fft	765	43	22	770
lu-c	191	2	5	192
radix	768	43	24	284-764
barnes	569	4	10	55-130
ocean-c	56	1	4	52
radiosity	32	0	1	29
raytrace	68	0	2	33
water-nsquared	4	1	0	5
water-spatial	3	0	0	3

Table 3: Overhead of Checkpoint and Recovery on Linux.

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 4: Characteristics and Results of SPLASH-2 Alpha Experiments

6.2 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 4.

6.2.1 Execution Time Overhead

We measured the overheads of instrumentation on Lemieux using the same methodology we used for Linux. Table 4 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. The state-saving mechanism described in Section 4 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 mechanisms described in Section 4 to circumvent this problem.

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

6.2.2 Checkpoint Sizes

Table 5 shows the checkpoint sizes for checkpoints created by the C^3 system. Note that the problem sizes used on Lemieux are different from the problems sizes we used on the Linux machine, so the sizes of checkpoint files are different on the two systems. We do not know of any system-level checkpointing system for Alpha/Tru64 that support multi-threaded programs, so we were not able to compare these checkpoint sizes with those of an SLC solution.

6.2.3 Checkpoint and Recovery Overhead

Table 5 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 3.

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 5: Overhead of each checkpoint and recovery on Alpha.

Benchmark	C^3 Checkpoint Size (MB)	Condor Checkpoint Size (MB)
sp	80	79
cg	428	427
bt	307	306
mg	435	435
ft	856	855
lu	45	44
ep	2	1
179.art	4	3
181.mcf	96	95
183.equake	42	46

Table 6: Comparison of C3 and Condor Checkpoint Sizes.

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. By comparison if `water-nsquared` were run on Alpha using the Linux parameters, it would have a 70s recovery overhead. 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.

6.3 Discussion

We ran a additional set of experiments comparing the checkpoint sizes produced by C^3 against those produced by the Condor [14] uniprocessor SLC system for Linux. The checkpoint sizes produced by both systems, on codes selected from the NAS OMP 2.3 (run in uniprocessor mode) and the SPEC 2000 CPU benchmarks suites are show in Table 6. It can be seen that the checkpoint sizes are very similar for both systems.

When we began this work, we invested considerable time in refining the protocol described in Section 3.1 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 as described in Section 4 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.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented an implementation of a blocking, co-ordinated 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 [5, 26], 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. We have shown that the sizes of checkpoints taken by our system are mostly comparable to those of system-level checkpoints; in principle, the size of our checkpoints may be reduced by using compiler analysis techniques [17].

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 [4] to produce a fault tolerance solution for programs using both message-passing and shared-memory constructs.

Acknowledgements

This work started as a term project in a graduate course on software for high-performance computing. Shafat Zaman was a member of that term project. We are also indebted to Kamen Yotov, who implemented the BCCK compiler substrate, which was used to implement the pre-processor used in this work.

8. REFERENCES

[1] A. Beguelin, E. Seligman and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1995.

[3] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997. Also available as `http://citeseer.nj.nec.com/beguelin97application.html`.

[4] 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.

[5] 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.

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

[7] R. Christodoulopoulou, R. Azimi, and A. Bilas. Dynamic data replication: an approach to providing fault-tolerant shared memory clusters. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, February 2003.

[8] Condor. http://www.cs.wisc.edu/condor/manual.

[9] 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.

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

[11] 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, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[12] P. Guedes and M. Castro. Distributed shared object memory. In *Proceedings of WWOS*, 1993.

[13] D. Hecht and C. Katsinis. Fault-Tolerant Distributed Shared Memory on a Broadcast-Based Interconnection Network. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, May 2000.

[14] 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.

[15] 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.

[16] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[17] J.S. Plank M. Beck and G. Kingsley. Compiler-Assisted Checkpointing. Technical Report Technical Report CS-94-269, University of Tennessee, December 1994.

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

[19] Z. Zhang M. Prvulovic and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared memory multiprocessors. In *International Conference on Computer Architecture*, 2002.

[20] K. Kusano M. Sato, S. Satoh and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, September 1999.

[21] Message Passing Interface Forum (MPIF). MPI: A message-passing interface standard. Technical Report, University of Tennessee, Knoxville, June 1995.

[22] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, C. Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html.

[23] 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.

[24] 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/.

[25] 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.

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

[27] 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`.

[28] F. Sultan, T.D. Nguyen, and L. Iftode. Scalable fault-tolerant distributed shared memory. In *Proceedings of Supercomputing 2000*, November 2000.

[29] 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.