Application-level Checkpointing for OpenMP Programs

Greg Bronevetsky, Keshav Pingali, Paul Stodghill
{bronevet,pingali,stodghil}@cs.cornell.edu
Department of Computer Science, Cornell University, Ithaca, NY 14853.

ABSTRACT
It is becoming important for long-running scientific applications to tolerate hardware faults. The most commonly used approach is checkpoint and restart (CPR) - the state of the computation is saved periodically to disk, and when a failure occurs, the computation is restarted from the last saved state. One common way of doing this, called System-level Checkpointing (SLC), requires modifying the Operating System and the communication libraries to permit the saving of the state of the entire parallel application. Unfortunately, this approach has poor portability since a checkpoint for one system rarely works on a different system. The only portable alternative is for the programmer to manually modify their program to enable CPR, a very labor-intensive task.

We are investigating the use of compiler technology to instrument codes to make them self-checkpointing and self-restarting, thereby providing an automatic solution for CPR that is portable to any system. In this paper, we describe an implementation of our checkpointing techniques for OpenMP shared-memory programs. 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. 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 two different platforms: Linux/x86, and Tru64/Alpha. Our experiments show that the overhead introduced by this approach is generally small; they also suggest ways in which the current implementation can be tuned to reduced overheads further.

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 solution 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 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 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.” 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 OpenMP applications. Applications programmers need only instrument a program with calls to a
function called \texttt{ccc\_potential\_checkpoint} 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 (\textit{C$^3$}) tool then automatically instruments the code so that it can save and restore its own state using any implementation of OpenMP. Our focus has been to support the basic OpenMP 2.5 specification, including support for all required OpenMP constructs and trivial support for nested parallelism (nested parallel regions run with only one thread). We have successfully tested our checkpoint/restart mechanism on multiple OpenMP platforms including x86/Linux (Intel compiler) and Alpha/Tru64 (Compaq/HP compiler).

The system described here builds on our previous work on ALC for message-passing programs [32] [7] [8] and for generic shared memory programs [10]. By combining the OpenMP checkpointer 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.

Alvisi et al. [17] 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, and almost all of this work uses SLC. 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 [33]. In non-blocking checkpointing, a global coordination protocol implemented by exchanging control messages is used to synchronize the state of all processes. Usually, a distinguished process called the initiator is responsible for initiating and monitoring the checkpoint; processes communicate with other processes to coordinate 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 [11].

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 [5]. Recently, our research group has pioneered preprocessor-based approaches for implementing ALC (semi-)automatically [32] [7] [8] [10]. 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. 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 [34] 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), ReVive [26] is another approach to hardware shared memory fault tolerance. Based on a combination of message logging and checkpointing. Both of these systems provides high efficiency, but because they are hardware-based, they do not provide portable solutions.

Dieter et al. [15] and the Berkeley Labs Linux Checkpoint/Restart [16] provide software-based checkpointing for SMP systems. The former 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 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, and are non-portable.

Checkpointing for software distributed shared memory has been explored in (SW-DSM) [22] [30] [12] [20] [36]. All are implemented within the SW-DSM system itself and exploit internal information about the state of the shared memory to generate consistent checkpoints. They are bound to a particular shared memory implementation and not portable.

3. \textbf{OVERVIEW OF APPROACH}

3.1 \textbf{Architecture}

Figure 1 describes our approach. The \textit{C$^3$} pre-compiler reads C/OpenMP application source files and instruments them to perform application-level saving of shared and thread-private state. This allows checkpointing to become a property of the application rather than of the underlying system and would thus be available on a platform on which the application is executed. The only modification that programmers must make to source files is to insert calls to a function called \texttt{ccc\_potential\_checkpoint} at points in the

![Figure 1: Overview of the \textit{C$^3$} System](image-url)
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 `ccc_potential_checkpoint` 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 below.

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 library 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 many 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 checkpointer for MPI programs to provide fault tolerance for hybrid MPI/OpenMP applications.

When checkpointing a parallel application it is important to have some mechanism to coordinate the state saving tasks performed by different threads. Our approach in this paper is to focus on a blocking checkpointing protocol. A blocking protocol is one where all threads synchronize on a barrier and save the application state as it was at that barrier. In particular, our basic protocol can be described as shown in Figure 2.

```
Global_Barrier
Save State
Global_Barrier
```

Figure 2: Blocking Checkpointing Protocol

The two barriers ensure that the application is not running while the checkpoint is being recorded. Furthermore, the first barrier ensures that any writes to shared data that were initiated before the checkpoint will have completed by the time the first barrier returns.

Finally, we note that the use of external libraries may make the “Save State” step difficult. If the library has state, this state must be saved and restored, but there is no general mechanism for accessing the state in arbitrary library codes. This problem arises in checkpointing OpenMP codes because the OpenMP implementation has state that needs to be saved and restored. We explore these issues in the next section.

3.2 State classification

To apply this basic approach to the general class of OpenMP programs, we need to describe what is done in the `Save State` step in Figure 2. We consider four categories of state that are present in a running OpenMP process,

- **Application state.** This includes the global and local variables that appear in the application source code, heap objects, and the stack activation frames that exist at the checkpoint.
- **Hidden state** This includes any state that is present within the OpenMP runtime system and that must be recreated on recovery. This state includes the locations of privatized and reduction variables, the binding between stack regions and thread numbers, and the scheduling of worksharing constructs.
- **Synchronization state.** This includes the state of any synchronization objects that a thread held or was blocked on at the checkpoint. These synchronization objects include barriers, locks, critical sections, and ordered regions in workshare constructs.

3.3 The Four R's of Checkpointing

Although the application depends on all of the above types of state, it may not have the same type of access to all of them. For example, the application has complete access to its process counter. It knows its value at all times and can use loops or gotos to move the process counter at will. The same is not true for heap objects or the ID of each thread.

While the application is completely aware of where its heap objects are located and the IDs of its threads, it has no power to choose where `malloc()` should place an object or what ID to give to a particular thread. Finally, some types of state like the location of reduction variables are completely hidden from the application until the reduction is complete.

Given the different types of state and the different amounts of control that the application has over them, we have employed four strategies for checkpointing and restarting these types of state:

- **Restore:** If the state can be directly manipulated by the application, then it can be directly saved at the checkpoint time and restored on restart. Global and local variables are examples of state that can be restored.
- **Replay:** If the state cannot be directly manipulated by the application but can be recreated using a sequence of deterministic operations, then on restart it can be regenerated by replaying these operations. OpenMP locks are examples of this. They cannot be restored by writing directly to the `omp_lock_t` structures, but their state can be recreated by making the appropriate calls to `omp_init_lock` and `omp_set_lock`.
- **Reimplement:** If the state cannot be recreated by replaying operations, then the operations need to be reimplemented so that the state can be saved and restored. Heap objects in C applications are examples of state that require reimplementation. The heap functions can be used to recreate heap objects on restart, but no sequence of calls can be made to ensure that these objects are allocated at the same addresses as when they were checkpointed. As a result, we have developed our reimplementation of the heap functions in order to ensure that heap objects are assigned the same addresses on restart.
- **Restrict:** If the state cannot be recreated by reimplementing the operations, then it cannot be supported. The application must be restricted from using this kind of state or to only using it in a restricted manner.
3.4 Discussion
In previous work, we describe a sequence of program transformations that add code to an application to save and restore the Application State. At a high-level, our system uses Restore to recreate the value of variables and heap objects, Replay to recreate activation frames, and we Reimplement the heap. We discuss our approach for recreating the stack in Section 4.1. More complete descriptions of these techniques can be found in [8, 7, 9].

The remaining categories of Hidden state and Synchronization state are described in detail in Sections 4 and 5, respectively.

In this paper, we do not describe in any detail how we handle the remaining constructs of OpenMP (e.g., master regions, atomic regions, flush directives, and the time functions). The only comment that is worth making here is that, while it is possible for a checkpoint location to appear inside of an atomic directive, the OpenMP specification allows the evaluation of complex expressions to be done outside of the atomic region. Since it would be impossible for a thread to participate in a blocking protocol from within an atomic region, we hoist complex expressions from atomic regions in order to prevent this from occurring.

4. HIDDEN STATE

4.1 Threads
The parallel directive is used for creating new threads and assigning to them regions of memory for their stacks. OpenMP also provides a number of routines for querying the runtime system for the current thread number, the number of threads currently running, and to changes these values. During a checkpoint enough information must be saved so that on restart the threads can be recreated and the runtime system can put into the same state that it was in at the checkpoint.

4.1.1 Recreate threads and their IDs
On restart the application must recreate the same number of threads that were present when the checkpoint was taken. While it is not possible for the application to directly create its own threads, OpenMP makes it easy to recreate threads by having the application pass through the same #pragma omp parallel directives that it passed thru during the original execution, using omp_set_num_threads to specify the number of threads to be created. Since OpenMP generates thread ID’s deterministically, we are guaranteed that this Replay mechanism will result in the creation of the same number of threads as during the original execution, with the same thread ID’s and the same nesting relationships.

4.1.2 Recreate thread to stack mapping
In order to restart successfully it is necessary that each thread be reassigned the same stack region that it was assigned in the original execution. The reason is that the checkpoint may contain pointers to objects on the stack: if these objects are assigned different locations on restart, then the pointers are invalid. One solution would be to update these pointers on restart, but this is only possible with stronger type information than C provides.

Although it is straightforward to recreate threads by re-executing parallel constructs on restart, OpenMP provides no guarantees about the location of the stacks assigned to each thread, nor does it allow an application to specify these locations. In theory, we are forced to reimplement parallel constructs, but in practice, OpenMP implementations appear to operate under some reasonable assumptions.

Suppose that we assume that when a parallel construct is re-executed on restart that set stacks allocated to the threads will be the same as during the original execution. In this case, thread 0 may be assigned the stack that was assigned to thread 1 in the original run. By providing a custom implementation of omp_get_thread_num, we can provide a mapping of stack regions to thread ids that is that same as during the original execution.

Let us relax this assumption to allow the stacks on restart to move relative to the original stacks by a small amount (< 1K). In this case it is possible to pad the start of each stack with a buffer and on restart use malloc() to pad it with the right number of bytes to make the starting point of the new stack line up to the old stack from the application’s point of view.

If neither of these assumptions holds, then it will be necessary to reimplement the mechanism for assigning stacks to threads. Under many versions of UNIX and Linux, this can be done using the mmap and setcontext system calls. There equivalent system calls under Windows as well.

To summarize, depending upon the assumptions that can be made about how a particular implementation assigns stacks to threads, it may be possible to use varying degrees of Replay and Reimplementation to ensure that threads are assigned the same stacks on restart. The most general and portable approach requires a complete reimplementation.

4.1.3 Recreate the stack contents
After we have recreated the threads and re-associated each thread with its correct stack and thread ID, the contents of each stack must be recreated. This consists of three tasks,

- Recreating the activation frames that were present on the stack when the checkpoint was taken,
- Ensuring that stack allocated variables are placed at the same locations, and
- Restoring the values of stack allocated variables.

We describe each in more detail below.

Recreating the activation frames. Since we allow the developer to place checkpoint locations at arbitrary points in the application source code, checkpoints can occur within nested function calls. Furthermore, checkpoints can be placed within regions that are encapsulated by OpenMP directives (e.g., within parallel regions). In a SLC systems, this execution context can be directly restored by writing the saved frames into the appropriate memory locations and restoring the value of the control registers.

ALC systems do not have direct access to the stack and registers, so Restore is not possible. Instead, the application source code must be selectively re-executed in order to Replay the creation of this execution context. This means adding statement labels to each function call that can reach a checkpoint location, and using these labels at restart in order to invoke the same functions whose frames were present when the checkpoint was taken. The details of how this is done can be found in our prior papers [8, 7] or in [9].

An important difference from the previous work is the fact that the execution context of a thread within an OpenMP
programs is more than just its activation frames; it also includes the OpenMP regions within which a thread is executing when it takes a checkpoint. The way that this additional context is recreated is similar to how activation frames are created. That is, statement labels are inserted so that on restart these regions will be entered in the same order as prior to checkpointing.

**Ensuring the location of stack allocated variables.** An application may create a pointer to a stack allocated variable that is saved in the checkpoint. Our system does not use a mechanism for retargetting these pointers on restart, so on restart it is necessary to ensure that all stack allocated variables are assigned to the same addresses. Otherwise, pointer values created prior to the checkpoint may become invalid after restart. In order to ensure that these variables are assigned the same addresses, it is necessary to ensure that each thread is assigned the same region for its stack, and we need to ensure that each activation frame starts at the same address. Ensuring that threads are assigned the same stack regions was discussed in Section 4.1.2.

If each thread is assigned the same starting address for its stack on restart, then creating the application activation frames by Replaying the appropriate application functions will ensure that the stack allocated variables will be assigned to the same locations. An additional complication occurs when a thread is executing within a region controlled by an OpenMP directive. When entering this region during the original execution, the OpenMP runtime system may have allocated a certain amount of storage on the stack. When reentering this region during restart, a different amount of storage may be allocated. The OpenMP specification is silent on the size of this of storage or whether this amount is deterministic. As a result, any local variables within the region may be assigned to different locations on restart. The solution is to use alloca to pad the stack during the original execution in much the same way that it was used to ensure the stacks start at the same address on restart.

**Restoring stack allocated variables.** Once the stack allocated application variables have been placed at their original memory locations, then it is possible for the application to directly restore their value from the checkpoint data.

For stack allocated variables used by the OpenMP runtime system this is not possible as their values may be tied to internal library state, which is not visible at the application-level. In many cases it may be possible to recreate this state using Replay. This is true for example with critical regions - a thread can reacquire a critical region simply by reentering it. For other instances of stack allocated state, it may not be possible. These are discussed in the next section.

### 4.2 Data

In an OpenMP application, there are variables whose locations cannot be deterministically assigned. These are variables that are explicitly privatized variables and reduction variables. The reason for this is that the specification does not explicitly say how these variables are allocated and assigned addresses.

Because of this, Replay is not an option and our system relies on Reimplementation. This is done via a program transformation that is performed by the precompiler. The details of the transformation can be found in [9], but the net effect is that the every program variable is assigned a location in the globals, stack or heap to which it can be deterministically reassigned on restart.

#### 4.3 Worksharing constructs

The final category of OpenMP constructs whose state is hidden to the application are the worksharing constructs, which include `single`, `sections`, and `for` constructs. These constructs assign work units to threads using various scheduling choices. The hidden state is this assignment and the completion status of each work unit.

In the most general sense, these constructs cannot be handled using a Replay mechanism. The reason is that the assignment of work units to threads is inherently nondeterministic. Consider the example shown in Figure 3. In this example, only one thread will be assigned to execute the body of the `single` construct. Suppose that in the original run of the application thread 3 is assigned thread. On restart, there is no way to re-execute the `single` construct and ensure that thread 3 would be assigned again.

```
#pragma parallel
{
    // pragma omp single
    {
        # pragma omp single
        {
            ... ccc_checkpoint(); ...
        }
    }
}
```

**Figure 3: Single Construct Example**

Because Replay is not possible, in the general case we must Reimplement the worksharing constructs. In this case, checkpointing the reimplemented worksharing constructs is simple; our implementation directly saves and restores the scheduling data structures.

However, rather than fully Reimplement worksharing constructs (which may result in reduced performance), it is possible to deal with them via a mixture of Replay and Reimplementation. Consider the example shown in Figure 4. Suppose that all of the threads take their checkpoint within this construct. On restart, we will need to 1. ensure that each thread will resume executing the same work unit within which it took the checkpoint and 2. make sure that the remaining work units will be allocated to threads. To deal with the first issue our compiler extracts the body of the for loop and directly jumps to the location inside the body where it took its checkpoint (Reimplementation). Once these work units have been completed, the native OpenMP for construct can be used to schedule the remaining work units (Replay). Since OpenMP requires that the loop inside a for construct have a very simple form, it will be necessary to map the remaining sparse iteration space into a dense form. This approach is illustrated in Figure 5.

Unfortunately, the situation with for constructs is not so simple. Consider the program in Figure 6. The nowait clause enables a thread to exit a for construct as soon as it has completed each of the work units that it has been assigned; it does not have to wait for the other threads to complete theirs. In this example, it is possible for thread 4 to take its checkpoint within the first for loop and for

---

2 For portable checkpointing, we do, in fact, use such a mechanism. See [18] for details.
5. SYNCHRONIZATION STATE

5.1 Synchronization and the blocking protocol

While it is easy to see how the basic protocol shown in Figure 2 can successfully save application state, the fact that it is a blocking protocol presents difficulties if the application contains synchronization constructs. The difficulty arises when one thread tries to checkpoint while holding a resource that another thread is waiting to acquire. This situation can result in deadlock, as shown in Figure 4.

In this situation, thread 0 will block at the first global barrier within the checkpoint protocol while thread 1 will block at the lock acquire. A similar situation happens with applications that use critical sections.

In OpenMP, there are two fundamental types of synchronization constructs. The first type involve the request and release of a resource. Locks, critical sections and ordered sections fall into this category. The second type involves the collective synchronization of threads. Barriers fall into this category.

5.2 Refining the Basic Protocol

One solution would be to use a mechanism for waking all blocked threads so that they could participate in taking the checkpoint. If we were to build our protocol within an existing OpenMP implementation, it might be possible to construct such a mechanism. However, our system is built at the application-level and OpenMP does not provide such a mechanism. We could achieve the same effect by having all threads rendezvous before blocking operations to determine whether or not a checkpoint was to be taken prior to blocking. This may result in threads waiting for a long time at the rendezvous waiting for other threads.

The approach that we take is to allow threads to block at synchronization points, but to ensure that (a) the blocked threads are unblocked in order to participate in the checkpointing, and (b) the threads that were unblocked are put back into a blocked state at the end of the checkpoint. Threads can be unblocked by releasing the resource that they were blocked on. In the case of locks, this means that the thread that holds the lock must release it. In the case of barriers, it means that all threads that are not waiting at a barrier must execute a barrier in order to release the threads blocked at any barrier. At the end of checkpoint, threads must reacquire the resources that they held at the beginning of the checkpoint and must re-execute any operation that they might have been blocked on when the protocol started.

Figure 6 shows the complete protocol framework.

When performing a checkpoint, a thread first releases all the resources it holds, which may include locks, critical regions and ordered regions. Then it ensures that any threads blocked at a barrier are released. After performing a global barrier, all of the threads save the application and hidden state. Before performing the second barrier, the thread reacquires all of the resources that it held before the checkpoint was initiated. The shared variable checkpoint initiated is used to inform awoken threads that the checkpoint protocol has been initiated. The parameter at barrier initiated is set to true when a checkpoint is being taken at an OpenMP barrier, and false otherwise. Its use will be described below.

Our protocol and its implementation do not use the native OpenMP barrier directive at all. This is because the OpenMP specification precludes the use of the barrier directive in certain constructs (e.g., work-sharing, critical, ordered or master). Because we allow the developer to place checkpoint locations within these constructs, we have reimplemented barriers in the routines ccc_global_barrier to avoid such violations.

To expand this high level idea into a complete protocol for OpenMP, we need to describe how locks, critical regions and ordered regions are released and reacquired and how barriers are released.

3For clarity, the code executed on restart is not shown.
shared */ int checkpoint_initiated = FALSE;

void ccc_potential_checkpoint()
{
    if(time_to_checkpoint())
        ccc_perform_checkpoint(FALSE);
}

void ccc_perform_checkpoint(bool at_barrier)
{
    checkpoint_initiated = TRUE;
    foreach r (resources_held_by_current_thread())
        release_xxx(r);
    if (!at_barrier) release_barrier();
    ccc_global_barrier();
    save_application_state();
    save_hidden_state();
    checkpoint_initiated = FALSE;
    foreach r (resources_held_by_current_thread())
        reacquire_xxx(r);
    ccc_global_barrier();
}

void ccc_global_barrier()
{
    /* Your favorite barrier implementation, */
    /* so long as it does not use */
    /* OpenMP's barrier. */
}

Figure 8: Checkpointing Protocol Framework

5.3 Incorporating Locks
When transforming the application source code using our pre-compiler, all calls to OpenMP locking routines (e.g., omp_set_lock, omp_unset_lock) are replaced with calls to routines in our runtime library (e.g., ccc_set_lock, ccc_unset_lock). The pseudocode for our locking routines and the release and reacquire routines required by the framework are shown in Figure 9.

The routines for releasing and reacquiring locks during checkpointing are trivial; they simply call the corresponding OpenMP routines.

Similarly, the routine ccc_set_lock calls omp_set_lock in order to acquire the lock. Once the lock has been acquired, the thread checks to see if the checkpointing protocol has been initiated. If so, it is assumed that the thread has been awoken so that it can take a checkpoint. In this case, it releases the lock that it was given in order to wake it up, performs a checkpoint and again attempts to acquire the lock. If the thread acquires the lock without the checkpointing protocol being initiated, then this fact is recorded and the lock acquisition is complete.

Releasing a lock is done by recording the fact that the thread is releasing the lock and calling omp_unset_lock to release the lock.

5.4 Incorporating Barriers
When transforming the application source code using our pre-compiler, all OpenMP barrier directives are replaced with calls to the ccc_barrier routines in our runtime library. The pseudocode for the our barrier routine and the release routine required by the framework are shown in Figure 10.

The flag threads_awoken is used to determine whether or not a barrier completed because all threads arrived at the barrier (=FALSE) or because a thread called release_barrier in order to awaken any thread that was blocked at the barrier (=TRUE).

The purpose of the at_barrier parameter to the procedure ccc_perform_checkpoint can now be explained. When
a thread is released from a barrier to take a checkpoint, it cannot call `release_barrier`. Doing so would result in deadlock. When `at_barrier` is true, this denotes that `ccc_perform_checkpoint` has been called from inside a `ccc_barrier` and the call to `release_barrier` should be skipped.

### 5.5 Incorporating Critical and Ordered Sections

Conceptually, critical and ordered sections are similar to locks, and the way that they are incorporated into the framework is similar to the code shown in Figure 9. There are several key differences.

The first is that there are no routines in OpenMP for explicitly acquiring or releasing critical or ordered sections. These are done implicitly by entering and exiting these regions. As a result of this fact, the routines `release_critical` and `release_ordered` must execute code to exit the enclosing regions. Similarly, `reacquire_critical` and `reacquire_ordered` must execute code to reenter the enclosing regions.

For example, in the program shown in Figure 9, suppose a checkpoint was taken inside the critical section in function bar() after bar() was called from foo(). Since the the checkpointing thread will be holding both critical sections A and B, it will need to first release them and later on reacquire them. In order to do this, it will need to first jump from the checkpoint location to the end of critical section B, exit B, jump to the end of critical section A, exit A, and finally take a checkpoint. In order to reacquire the critical sections it will need to do the same things in reverse: reenter critical section A, jump to the start of critical section B, reenter B, jump to the original checkpoint location and finish the checkpoint.

```c
bar()
{
    ...
    #pragma omp critical(Name_B)
    {
        ...
        ccc_potential_checkpoint();
        ...
    }
    ...
}
foo()
{
    ...
    #pragma omp critical(Name_A)
    {
        ...
        bar();
        ...
    }
    ...
}
```

**Figure 11:** Critical example

This can be done by using a set of control flags and introducing additional `return` and `goto` instructions into the application source code. `setjmp` and `longjmp` may be used as well. For lack of space, we have omitted the details from this paper, but they can be found in [9].

The second key difference is that when a critical or ordered section is reexecuted, either during restore or replay, it must be ensured that the local variables in the enclosed scope are given the same addresses that they had during the original execution. The OpenMP specification does not guarantee this, but it can be handled in a manner similar to the approach described in Section 5 for ensuring that stack regions are correctly reassigned on restart.

### 5.6 Implicit Synchronization

It is possible for an application to implement its own form of synchronization. For example, an application might use the `atomic` directive together with a spin or polling loop to implement barriers.

When an application contains this sort of implicit synchronization, our checkpointing framework will likely deadlock. The reason is fundamental - our protocol assumes that there is a mechanism for releasing resources and barriers. There is no way for our system to release threads that are blocked at application-implemented synchronization. Therefore, we use the Restrict methodology and prohibit the application from using its own form of synchronization.

### 5.7 Eager vs. Lazy

When a thread acquires a lock in `ccc_set_lock`, it will take a checkpoint if the `checkpoint_initiated` flag is set. This is true whether the thread was awoken by another thread calling `release_lock` or not. We call this an “eager” checkpointing mechanism. If instead of checking `checkpoint_initiated`, a thread were to check a flag set in `release_lock`, then it could continue executing until it attempted to acquire a resource that was release by another thread taken a checkpoint. We call this a “lazy” checkpointing mechanism. (The same classification applies to other similar resources like critical and ordered sections)

A lazy checkpointing mechanism can be made to behave like an eager checkpointing mechanism by inserting potential checkpoint locations into the application immediately before calls to `ccc_set_lock`, etc. There is performance trade-offs that can be made between these two approaches. For some applications, an eager approach may give better performance because threads spend less time waiting for other threads to checkpoint. For other applications, a lazy approach may give better performance because threads are more likely to take checkpoints only at the developer specified locations. For still other applications, a hybrid approach may give the best performance.

The system used for the experiments reported in Section 6 uses a pure lazy mechanism.

### 6. IMPLEMENTATION AND EXPERIMENTS

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 that keep track of local variables and OpenMP-specific state. Furthermore, if checkpoints are taken, writing the checkpoints to disk adds to the execution time of the program. Because different fault/migration environments will require different checkpointing frequencies,
it is important to measure the two overheads separately. In this section, we present experimental results that measure these overheads for the C³ system.

We decided to evaluate our system using the OpenMP versions 1 of the NAS Parallel Benchmarks 2. None of the codes in these benchmarks ran for longer than 30 minutes on our test systems, meaning that they were not themselves vulnerable to hardware failures. However, since these codes represent realistic computations, C³’s performance with these codes should be indicative of its performance on real-world large applications.

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 NAS benchmarks on two different platforms: a 2-way Athlon machine running Linux and a 4-way Compaq Alphaserver running Tru64 UNIX.

The Linux experiments were conducted on a 2-way 1.73GHz 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 8.0. All experiments were run using both processors. Checkpoints were recorded to the local disk.

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.

6.1 Checkpoint-free Overheads

In our first experiment, we measured the overhead of C³’s instrumentation of the application code. This requires the measurement of the running times of (i) the original codes, and (ii) the instrumented codes without checkpointing. Running times were taken from the output of each benchmark as this gives us the performance of the section of the code performing the computational behavior represented by that benchmark. Each experiment is the average of multiple runs. Results for the x86/Linux runs are shown in Table 1 while results for the Alpha/Tru64 runs are shown in Table 2 For each code we show results for the largest problem size that could run on that system. The Alpha/Tru64 table is missing results for MG and BT because those two codes fail to run on this platform.

The tables show that for most codes, the overhead introduced by C³ was generally small. There are a few exceptions:

- FT on x86 shows a 12% overhead due to C³’s transformations. Upon closer examination it turns out that FT is very sensitive to the use of alloca(). Indeed, the use of alloca in a number of positions inside FT’s code will result in this overhead while C³’s other code modifications appear to have little effect on FT’s running time. It is not yet clear in what way alloca() affects FT’s performance or whether we can use other stack management techniques to reduce the overhead.
- IS on Alpha shows a 13% overhead. However, this is for a running time of 1.24s, which is not significant. Unfortunately, IS’s memory requirements precluded us from running it on problem classes that would generate longer running times.

6.2 Per-checkpoint Overheads

For our second experiment we evaluated the overheads of checkpointing and restarting the application. Table 3 shows our results on x86/Linux and Table 4 shows them on Alpha/Tru64. For each code we list the following information:

- The size of the checkpoint, in MB
- The amount of time taken to perform one checkpoint, both in seconds and as a percentage of the original application’s runtime. This includes not just the time it took to record one checkpoint but also any miscellaneous effects on the application’s runtime as a result of taking the checkpoint.
- The amount of time taken to perform one restart, both in seconds and as a percentage of the original application’s runtime. The restart was performed after taking a single checkpoint, and the restart time was computed via the following formula: (upto_chkpt + restart_till_end) - 1chkpt_app, where
  - upto_chkpt - the time from the start of the application to the end of the checkpoint
  - restart_till_end - from the beginning of the restart execution until its termination
  - 1chkpt_app - the running time of the transformed application, taking 1 checkpoint

The first thing to note is that checkpoint and restart times do not correlate directly to checkpoint sizes. While this correlation would be intuitive, it is also true that the checkpoint and restart process is more involved than just saving the bits of the application to disk, seeing as it requires a notable amount of Replay and Reimplementation. In particular, it appears that the restart time of EP on both platforms is as high as it is relative to checkpoint overhead (6.3% and 7.7% per restart vs - .05% and -.4% per checkpoint) because checkpoints in EP are taken inside of a parallel for loop, meaning that for some part of the restart execution the parallel for iterations are being scheduled by our algorithm, which is less efficient than the native algorithm in the OpenMP library.

Another factor that may be affecting the results is the use of buffering on the part of the file system. It is not clear when the checkpoint data written to files is residing in memory and when it has definitely been written out to disk. Our experiments do not control for this effect but it may help to explain the lack of direct correlation between checkpoint sizes and the overheads of checkpointing and restarting.

Another interesting phenomenon is the fact that in some cases the checkpoint and restart overheads are negative. Because these negative overheads are so small relative to the runtime of the application, it is not clear whether they are real effects or noise. In general, while it is clear that the effect of a single checkpoint and a single restart are not limited to the number of instructions executed (in which case all overheads would be positive), it is not clear exactly what effect they have on different applications. This would be an interesting area to examine more deeply in the future.

Finally, it can be seen that for most of the codes the cost of taking a checkpoint and restarting is generally a small fraction of the overall program runtime. The only cases when
### 7. SUMMARY

In this paper we have described a protocol for checkpointing OpenMP applications at the application level. This approach makes checkpointing a property of the application rather than of any given system, allowing the application to checkpoint on any system, running with any implementation of OpenMP.

For this work we have extended the basic approach in [10] to support the OpenMP specification. While [10] only deals with barriers and locks, this work addresses a number of issues that arise when trying to checkpoint OpenMP constructs at the application-level. These include critical sections, privatization of variables, worksharing constructs (for, sections and single) and ordered sections among others.

We have described the implementation of our protocol and given a performance analysis based on several popular benchmarks. We have shown that code inserted by our system has a minimal impact on performance, and that checkpoint and restart costs are reasonable.

#### 7.1 Checkpointing Framework

In addition to presenting concrete techniques for checkpointing OpenMP applications, we present a general framework for describing and developing Application-level Checkpointing algorithms. This framework for State Recreation consists of the 4 R’s: Restore, Replay, Reimplement, and Restrict.

When trying to recreate a given piece of state a checkpoint must go through a series of questions. If it has direct access to read and modify this state, then it can simply Restore it. If it does not have direct access to this state but the state can be recreated via a sequence of deterministic operations, then we can use the Replay option by issuing those deterministic operations. If the piece of state was created using non-deterministic operations then Reimplement cannot be used and it is only possible to Restrict this piece of state and checkpoint this implementation directly. Finally, if Reimplementation proves to be too difficult or not possible then we can Restrict the set of applications to only those that either do not use the given piece of state or use it in a restricted fashion.

We believe that the 4R framework has wider applicability. Consider the job of a checkpointing algorithm inside the Operating System (a classic System-level checkpointer). It has direct read-write access to the application’s memory, meaning that it can use Restore to reconstruct its state. However, an OS-level checkpointer has very limited access to the hardware and must use Replay, Reimplement or Restrict to deal with the various complexities of its state. In particular, the Chandy-Lamport distributed snapshot protocol [11] is a way for system-level check pointers to use Replay to checkpoint the state of the network. Finally, this framework extends even to hardware-level check pointers that have direct access to software state but only limited access to the state of network wires or other system devices outside the given piece of hardware.

#### 7.2 Future Work

While this paper presents a method for checkpointing OpenMP applications at the application-level, our prior work [32, 8] has presented a method for checkpointing MPI applications at the application-level. There are many hybrid MPI/OpenMP applications that would benefit from a checkpointing system that combined the two approaches. Such a system would be able to checkpoint applications using MPI, OpenMP or both, running on any system, using any implementation or MPI and OpenMP.

Another area of active research is checkpointing in a het-

---

### Table 1: NAS x86/Linux Experiments

| Benchmark | Problem size | Uninstrumented run time | C

- instrumented run time | 0 checkpoints taken | C

- instrumentation overhead |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>A</td>
<td>1258s</td>
<td>1230s</td>
<td>2.20%</td>
</tr>
<tr>
<td>CG</td>
<td>B</td>
<td>1370s</td>
<td>1444s</td>
<td>5.41%</td>
</tr>
<tr>
<td>EP</td>
<td>C</td>
<td>1024s</td>
<td>1045s</td>
<td>2.03%</td>
</tr>
<tr>
<td>FT</td>
<td>A</td>
<td>33s</td>
<td>36s</td>
<td>11.00%</td>
</tr>
<tr>
<td>IS</td>
<td>A</td>
<td>23s</td>
<td>24s</td>
<td>5.31%</td>
</tr>
<tr>
<td>LU</td>
<td>A</td>
<td>1625s</td>
<td>1650s</td>
<td>1.53%</td>
</tr>
<tr>
<td>MG</td>
<td>A</td>
<td>315s</td>
<td>338s</td>
<td>12.75%</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
<td>666s</td>
<td>687s</td>
<td>3.24%</td>
</tr>
</tbody>
</table>

### Table 2: NAS Alpha/Tru64 Experiments

| Benchmark | Problem size | Uninstrumented run time | C

- instrumented run time | 0 checkpoints taken | C

- instrumentation overhead |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>B</td>
<td>889s</td>
<td>916s</td>
<td>2.99%</td>
</tr>
<tr>
<td>EP</td>
<td>C</td>
<td>338s</td>
<td>360s</td>
<td>6.51%</td>
</tr>
<tr>
<td>FT</td>
<td>B</td>
<td>97s</td>
<td>99s</td>
<td>2.60%</td>
</tr>
<tr>
<td>IS</td>
<td>A</td>
<td>124s</td>
<td>14s</td>
<td>12.75%</td>
</tr>
<tr>
<td>LU</td>
<td>B</td>
<td>326s</td>
<td>330s</td>
<td>1.16%</td>
</tr>
<tr>
<td>SP</td>
<td>B</td>
<td>625s</td>
<td>654s</td>
<td>4.73%</td>
</tr>
</tbody>
</table>
erogeneous computing environment (e.g., checkpoint on an IA-32 Windows machine and restart on an Alpha Tru64 machine). In this work, compile type analysis is used to ensure that every application variable or heap object is assigned a unique type. Information about this type is stored with the checkpoint and used to translate between machine representations on restart. In [15], we show that this can be this approach can be combined with our protocol for checkpointing MPI to provide portable checkpointing of MPI applications. We plan to investigate whether a similar approach can be used to develop a portable checkpoint system for OpenMP applications.

The checkpointing protocol presented in this paper is a blocking protocol: all threads synchronize at a barrier before taking a checkpoint. This limits the types of applications it can deal with (no applications that synchronize using variables), limits scalability of our solution and adds complications to the problem of checkpointing OpenMP constructs. We are working on non-blocking protocols for OpenMP checkpointing that would allow threads to checkpoint independently at different times and tie those individual snapshots together into a single recoverable checkpoint.

While our protocol provides a mechanism for checkpointing, it does not address any policy decisions about when checkpointing should be done. Such policy decisions may depend on the frequency of system failures (more failures require more frequent checkpoints), need for the application to migrate, system scheduling policy or the cost of taking checkpoints at different locations in the program. Such policy decisions are a complex area of research and it would be interesting to integrate them into our solution.

8. REFERENCES


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Checkpoint Size (MB)</th>
<th>Seconds per Checkpoint</th>
<th>Percent of Original Runtime</th>
<th>Seconds per Recovery</th>
<th>Percent of Original Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>4.25</td>
<td>2.2</td>
<td>1.7</td>
<td>1.9</td>
<td>4</td>
</tr>
<tr>
<td>IS</td>
<td>12.2</td>
<td>-.5</td>
<td>-.05</td>
<td>.9</td>
<td>48</td>
</tr>
<tr>
<td>LU</td>
<td>45</td>
<td>-.30</td>
<td>-1.8</td>
<td>17</td>
<td>47</td>
</tr>
<tr>
<td>MG</td>
<td>435</td>
<td>5.5</td>
<td>7.7</td>
<td>1.6</td>
<td>2.3</td>
</tr>
<tr>
<td>SP</td>
<td>80</td>
<td>-.25</td>
<td>.04</td>
<td>-11.5</td>
<td>-1.7</td>
</tr>
</tbody>
</table>

Table 3: Overhead of Checkpoint and Recovery on x86/Linux.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Checkpoint Size (MB)</th>
<th>Seconds per Checkpoint</th>
<th>Percent of Original Runtime</th>
<th>Seconds per Recovery</th>
<th>Percent of Original Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>1103</td>
<td>31</td>
<td>3.5</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>EP</td>
<td>4.3</td>
<td>1.4</td>
<td>4</td>
<td>7.7</td>
<td>2.3</td>
</tr>
<tr>
<td>FT</td>
<td>1683</td>
<td>31</td>
<td>3.2</td>
<td>9.8</td>
<td>10</td>
</tr>
<tr>
<td>IS</td>
<td>905</td>
<td>3.8</td>
<td>3.06</td>
<td>5.6</td>
<td>32</td>
</tr>
<tr>
<td>LU</td>
<td>175</td>
<td>3.2</td>
<td>1</td>
<td>2.7</td>
<td>.4</td>
</tr>
<tr>
<td>SP</td>
<td>317</td>
<td>5.3</td>
<td>.3</td>
<td>2.7</td>
<td>.4</td>
</tr>
</tbody>
</table>

Table 4: Overhead of Checkpoint and Recovery on Alpha/Tru64. (restart overheads for CG and LU could not be measured in time for submission)


