# Experimental Evaluation of 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 computation's state is saved periodically to disk. Upon failure the computation is restarted from the last saved state. The common CPR mechanism, called System-level Checkpointing (SLC), requires modifying the Operating System and the communication libraries to enable them to save the state of the entire parallel application. This approach is not portable since a checkpointer for one system rarely works on another. Application-level Checkpointing (ALC) is a portable alternative where the programmer manually modifies their program to enable CPR, a very labor-intensive task.

We are investigating the use of compiler technology to instrument codes to embed the ability to tolerate faults into applications themselves, making them self-checkpointing and self-restarting on any platform. In [9] we described a general approach for checkpointing shared memory APIs at the application level. Since [9] applied to only a toy feature set common to most shared memory APIs, this paper shows the practicality of this approach by extending it to a specific popular shared memory API: OpenMP. We describe the challenges involved in providing automated ALC for OpenMP applications and experimentally validate this approach by showing detailed performance results for our implementation of this technique. Our experiments with the NAS OpenMP benchmarks [1] and the EPCC microbenchmarks [21] show generally low overhead on three different architectures: Linux/IA64, Tru64/Alpha and Solaris/Sparc and highlight important lessons about the performance characteristics of this aproach.

## Categories and Subject Descriptors

D.4.5 [**Reliability**]: Checkpoint/Restart
; C.1.4 [**Parallel Architectures**]
; D.1.3 [**Concurrent Programming**]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'06 June 28-30, Cairns, Queensland, Australia.
Copyright ©2006 ACM 1-59593-282-8/06/0006 ...$5.00.

## General Terms

Algorithms Reliability Performance Experimentation

## Keywords

fault tolerance, checkpointing, shared memory, parallel, 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 as machines are becoming bigger and more complex, their mean time between failures (MTBF) 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 high-performance computing(HPC) is checkpoint and restart (CPR). During a program's execution its state is saved periodically to stable storage; when a fault is detected the program is restarted from the last checkpoint. Most existing systems for checkpointing such as Condor [16] take System-Level Checkpoints (SLC), which are essentially core-dump-style snapshots of machine state. 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." [11]. Indeed, a brief survey of the major US supercomputing centers and government labs shows that most machines do not have automated checkpointing and even when they do, its use is generally not recommended by the system administrators. Furthermore, system-level checkpoints by definition cannot be restarted on a platform different from the one on which they were created.

In contrast to SLC, in Application-level Checkpointing (ALC) application source code is modified to save and restore its own state. Because the application become self-checkpointing and self-restarting, ALC avoids SLC's extreme dependence on particular machines and OS's. Further, an appropriately created checkpoint 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 IBM's BlueGene/L, an application-level checkpoint is a few MB's in size while a full system-level checkpoint is many TB's. On most large

platforms hand-implemented ALC is the default.

This paper describes a semi-automatic ALC system for OpenMP applications. Programmers need only add to their program calls to function `ccc_potential_checkpoint` at program locations where a checkpoint may be desirable (ex: due to smaller live state). Our Cornell Checkpointing Compiler ($C^3$) tool then automatically instruments the code so that it can save and restore its own state using any implementation of OpenMP. The system described here builds on our previous work on ALC for generic shared memory programs [9]. While in [9] our basic approach was shown to work with a severely limited shared memory API, this paper experimentally proves the viability of the approach by extending the framework from [9] to support the 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 Linux/IA64 (Intel compiler), Tru64/Alpha (Compaq/HP compiler) and Solaris/Sparc (SunPro compiler), showing low overheads. For each platform we present performance results for the NAS Parallel Benchmarks [1] and a detailed feature-by-feature performance breakdown via the EPCC microbenchmarks [21].

By combining this OpenMP checkpointer with our previous work on checkpointing MPI programs [22] [6, 7], it is possible obtain fault tolerance for 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. We discuss the categories of state that can be present when an OpenMP application is checkpointed, with hidden state in Section 4 and synchronization state in Section 5. In Section 6, we present experimental results. Finally, we give our conclusions and discuss ongoing work in Section 7.

## 2. PRIOR WORK

Alvisi et al. [14] is an excellent survey of of rollback-restart techniques for message-passing applications. The bulk of the work on CPR has focused on such applications and almost all of this work uses SLC. *Blocking* techniques [24] bring all processes to a stop before taking a global checkpoint. In *non-blocking* checkpointing, processes may checkpoint themselves at different times but coordinate the timing of these local checkpoints to ensure a consistent global state. The Chandy-Lamport protocol is perhaps the most well-known such protocol [10].

In the HPC community hand-coded application-level checkpointing at global barriers is the norm. The Dome project explored hand-coded ALC in an object-oriented language for computational science applications [4]. Recently, our research group has pioneered preprocessor-based approaches for implementing ALC (semi-)automatically [22] [6, 7, 9]. 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 have been restricted to SLC and are bound to particular shared memory implementations. Both hardware and software approaches have been proposed.

SafetyNet [23] inserts buffers near processor caches and

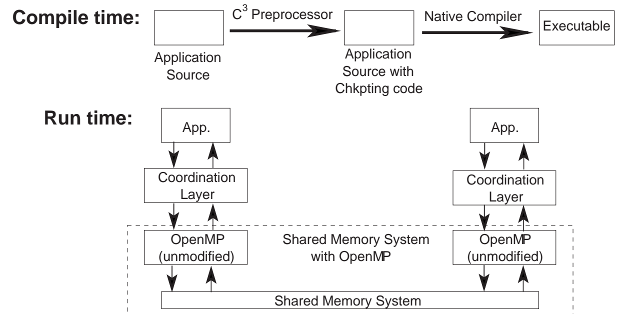


**Figure 1: Overview of the $C^3$ System**

memories to log changes in local processor memories as well as messages between processors. ReVive [18] 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.[12] and BLCR [13] 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. The latter uses dynamically loadable kernel modules to directly control the thread scheduler and force consistency among all threads. In contrast to our solution, both approaches are bound to a particular thread library and kernel version, and are non-portable. Checkpointing for software distributed shared memory (SW-DSM) has been explored in [17, 19]. All are implemented inside the SW-DSM system and are bound to a particular shared memory implementation.

While prior work has focused on modifying shared memory libraries on specific systems, it is also possible to enhance a shared memory library that works on multiple platforms to enable it to provide checkpointing functionality. Although this would achieve the same goal as $C^3$, a significant amount of labor would be required to port the library to many different platforms and make it as efficient as the native implementation on each platform. The goal of $C^3$ is to create a single checkpointing solution for all implementations of OpenMP, making porting to new platforms trivial.

## 3. OVERVIEW OF APPROACH

### 3.1 Architecture

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 private state. This makes checkpointing a property of the application rather than of the underlying system, making it available on any platform on which the application is executed. The only modification programmers make to source files is to insert calls to function `ccc_potential_checkpoint` at program points where checkpoints may be taken. These points should ideally have minimal live state and need to be executed at least as often as the desired checkpoint interval. In practice we've found that placing potential checkpoint locations at the top of the application's main computation loop satisfies these conditions. Note that checkpoints need not be taken at every `ccc_potential_checkpoint` call; instead, the choice can be based on a timer or an adaptive mechanism such as [3]. Checkpoints taken by individual threads are coordinated by the protocol described below.

The pre-compiler's output is compiled with the given platform's native compiler and linked with a library that implements a coordination layer for generating consistent snapshots of the computation state. This layer sits between the application and the OpenMP runtime and intercepts all calls from the instrumented application program to OpenMP. This design permits us to implement checkpointing without access to the underlying OpenMP implementation.

In this paper we focus on a blocking protocol to coordinate the state saving tasks of different threads into a consistent global snapshot of application state. Our basic protocol is shown below:

```
Global_Barrier
Save State
Global_Barrier
```

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. This makes this protocol applicable to relaxed consistency models, of which OpenMP is an example.

### 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 the above protocol. We consider four categories of state that are present in a running OpenMP process,

- **Application state**: Includes application variables (both global and local), heap objects and the stack activation frames that exist at checkpoint-time.
- **Hidden state**: Any state inside the OpenMP runtime system that must be recreated on restart. Includes the locations of privatized and reduction variables, the binding between stack regions and thread numbers, and the scheduling of worksharing constructs.
- **Synchronization state**: The state of any synchronization objects that a thread held or was blocked on at checkpoint-time. These include barriers, locks, critical sections, and ordered regions in worksharing 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 and can manipulate it via loops and gotos. The same is not true for heap objects or thread IDs. While the application knows the locations of its heap objects and the IDs of its threads, it has no power to choose where `malloc()` places an object or what ID is given 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 access the application has to different types of state, we use four strategies for checkpointing and restarting these types of state:

- **Restore**: State that can be directly manipulated by the application, can be directly saved at the checkpoint-time and restored on restart. Example: Global and local variables.
- **Replay**: State that cannot be directly manipulated by the application but can be recreated using a sequence of deterministic operations, can on restart be regenerated by replaying these operations. Example: OpenMP locks since their state is inaccessible to the application but can be recreated using lock routines.
- **Reimplement**: If the state cannot be recreated by replaying operations, then the operations need to be reimplemented so that it can be. Example: heap objects in C since on restart it is not possible to force `malloc()` to place these objects at their original locations. We have developed our own self-checkpointing implementation of the heap functions.
- **Restrict**: If the state cannot be recreated by reimplementing the operations, it cannot be supported. The application must be restricted from using this kind of state or to only using it in a restricted manner.

Next to Restricting the application from using OpenMP, the simplest way to checkpoint an OpenMP application is to apply Reimplement to provide a cross-platform self-checkpointing implementation of OpenMP. However, to ensure maximal performance on all platforms we have attempted to use the least invasive type of checkpointing mechanism possible. (i.e. Replay before Reimplement, etc.) While this has the downside of a more complex solution, our experiments (Section 6) validate our approach.

### 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 4.1.3. More complete descriptions of these techniques can be found in [6, 9]. The remaining categories of Hidden state and Synchronization state are described in detail in Sections 4 and 5, respectively.

Due to lack of space, 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) as these details are trivial. 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, OpenMP guarantees atomicity for only the update operation of the atomic construct. Any computations on the atomic update's right hand side (including the potential checkpoint location) have no atomicity guarantees. As such, we hoist the right hand sides of atomic updates out of OpenMP atomic directives, legally ensuring that potential checkpoint locations never occur in atomic directives.

## 4. HIDDEN STATE

### 4.1 Threads

The parallel directive creates new threads and assigns them memory regions for their stacks. Each thread can determine its own thread number and the number of threads currently running and to changes these values. A checkpoint must contain enough information to recreate the threads on restart and reset the OpenMP runtime system into an equivalent state from the application's point of view.

#### *4.1.1 Recreate threads and their IDs*

On restart the application must recreate the same number of threads that were present at checkpoint-time. In OpenMP new threads are created when some parent thread passes through a `#pragma omp parallel` directive. To recreate

the same threads on restart we have the application pass through the `parallel` directives that had created the threads it had at checkpoint-time. (`omp_set_num_threads` is used to recreate the same number of threads) Thread IDs are generated deterministically.

#### 4.1.2 Recreate thread to stack mapping

The application may have pointers to stack objects. Since our system does not use a mechanism for retargetting these pointers on restart[1], they will become invalid if the objects move. As such, it is necessary that each thread be reassigned the same stack region that it had in the original execution. Although OpenMP does not provide guarantees about thread stack locations nor allows applications to choose them, in practice OpenMP implementations operate under some reasonable assumptions.

Suppose that we assume that when a `parallel` construct is re-executed on restart, the set of stacks allocated to the threads will be the same as during the original execution, even if the thread to stack mapping is different. (i.e. thread 0 may be assigned the stack that was originally assigned to thread 1). By reimplementing `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 (a few KB). In this case the start of each stack can be padded with a buffer, using `alloca()` on restart to pad it with the right number of bytes to make the new stack's starting point line up to the old stack from the application's point of view.

If neither assumption holds, we can reimplement the thread to stack assignment mechanism. In many versions of UNIX and Linux, this can be done using the `mmap` and `setcontext` system calls. Windows has equivalent system calls.

To summarize, depending upon the assumptions that can be made about how a particular implementation assigns stacks to threads, it is 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 of this feature.

#### 4.1.3 Recreate the stack contents

Once we recreate the threads and re-associate each thread with its correct stack and thread ID, the contents of each stack must be recreated. This consists of

- 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.

*Recreating the activation frames.* Since the developer may place checkpoint locations at arbitrary points in the source code, checkpoints can occur within nested function calls, including regions encapsulated by OpenMP directives (e.g. within parallel regions). While the application can restore the stack regions that it created, it cannot restore the stack regions created by entering OpenMP regions since their size and contents are determined by the OpenMP implementation and their presence may correspond to additional state inside of OpenMP. As such, we recreate the stack via Replay by calling the function calls and reentering the OpenMP directives that were on the stack at checkpoint-time. This is done by adding statement labels to function calls and OpenMP directive that can reach a checkpoint location and using these labels on restart to invoke the same functions whose frames were present when the checkpoint was taken, as described in [7, 6].

*Ensuring the location of stack allocated variables.* To preserve the validity of application pointers, on restart all stack allocated variables must be assigned to the same addresses. In Section 4.1.2 we discuss how threads are assigned their original stack regions. Thus, all that remains is to ensure that all the function activation frames and OpenMP directives are placed at the addresses they held at checkpoint-time. This is trivial for function calls but more complex for OpenMP directives since there is no guarantee that when a given OpenMP directive is reentered on restart it will take up the same amount of stack space as during the original execution. The solution is to use `alloca` to pad the stack during the original execution, using the mechanism above to align restart stack starting addresses to their original values.

*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 values from the checkpoint.

### 4.2 Data

In OpenMP there exist variables whose locations cannot be deterministically assigned. These are the explicitly privatized variables and the 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, which is performed by the precompiler. The details of the transformation can be found in [8] 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 because assignment of work units to threads is non-deterministic. Consider the `single` directive. Although all threads must reach the same `single` directive, only one of them may execute its code. Suppose during the application's original run this work unit is assigned to thread 3, which took a checkpoint inside of it. On restart, there is no way to re-execute the `single` construct and ensure that it would again be assigned to thread 3.

Because Replay is not possible, one possibility would be to Reimplement the worksharing constructs. In this case, checkpointing the reimplemented worksharing constructs would become simple: our implementation would directly save and restore 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 Reim-

[1]For portable checkpointing, we do use such a mechanism. See [15] for details.

plement. Consider the example shown in Figure 2. Suppose that every thread takes its checkpoint within this construct. On restart, we must 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 unexecuted work units will be scheduled to run on some threads. The restart code is shown in Figure 3. To deal with the first issue our compiler extracts the loop's body and directly jumps to the location inside where the checkpoint was taken (Reimplement). Once these work units have been completed, the native OpenMP `for` construct is used to schedule unexecuted work units (Replay). Since OpenMP requires that the loop inside a `for` construct have a very simple form, the remaining sparse iteration space is mapped into a dense form (into array `remaining_iters[]`). Note that since this solution does not depend on which work units were executed, in-progress or remaining, it works with all loop scheduling policies.

```
#pragma omp for
for(i=0; i<1000; i++)
{ ... loop body ... }
```

**Figure 2: For Construct Example**

```
i = iter_at_checkpoint(thread_id);
{ ... loop body ... }
// wait for the other threads to finish.
ccc_global_barrier();
remaining_iters[*] =
    condense_remaining_iters();
#pragma omp for
for (ii=0; ii<num_remaining_iters; ii++) {
    i = remaining_iters[ii];
    { ... loop body ... }
}
```

**Figure 3: Restarting a simple `for` construct**

The above mechanism works for worksharing constructs without a `nowait` clause (i.e. terminated by an implicit barrier). However, in situations where the `nowait` clause is present it is possible for different threads to checkpoint inside of different worksharing constructs. Since OpenMP requires each thread to execute the same sequence of worksharing constructs, on restart we will not be able to use native OpenMP `for` loops (as we did in Figure 3) to schedule the unexecuted iterations of application worksharing constructs that were already passed by some threads at the time of the checkpoint. As such, in this situation we use the following scheme. Let $W$ be the latest worksharing construct that any thread took a checkpoint inside of. For all worksharing constructs that precede $W$ we schedule their unexecuted work units using our own implementation (Reimplement). $W$'s unexecuted work units are schedule by using the mechanism in Figure 3 (Replay).

# 5. SYNCHRONIZATION STATE

## 5.1 Synchronization and the Protocol

While it is easy to see how the basic protocol can successfully save application state, the fact that it is blocking presents difficulties if the application contains synchronization constructs. The problem arises when one thread tries to checkpoint while holding a resource that another thread is waiting to acquire. This can result in deadlock, as shown in Figure 4 where thread 0 will block at the first global

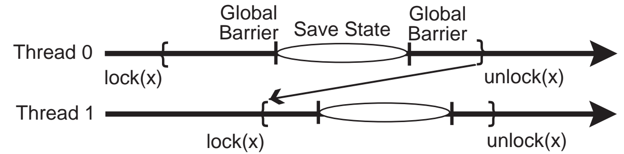


**Figure 4: Deadlock example**

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

OpenMP has 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 wake up all blocked threads so that they could participate in taking the checkpoint. Unfortunately, OpenMP does not provide applications with such functionality and the only way to unblock a blocked thread is to give it the resource on which it is blocked.

Thus, our approach is: 1. at checkpoint-time ensure that all threads are awake by releasing all resources and 2. before the checkpoint is complete, put all the threads that were forcefully awoken back into a blocked state. For locks this means that the thread holding the lock must release it. For 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 a barrier. At the end of checkpoint, threads must reacquire the resources they held at the checkpoint's start and must re-execute any operation that they might have been blocked on when the protocol began. Figure 5 shows the complete protocol framework[2]. Function `time_to_checkpoint()` decides whether it is time to take a checkpoint by using a timer, user guidance, an adaptive mechanism [3] or some other technique.

```
/* 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() {
    /* Any barrier implementation that */
    /* doesn't use OpenMP's barrier.   */
}
```

**Figure 5: Checkpointing Protocol Framework**

When performing a checkpoint, a thread first releases

[2]For clarity, the code executed on restart is not shown.

all the resources it holds, which may include locks, critical regions and ordered regions. Then releases any threads blocked at a barrier. After performing the first checkpoint barrier, all the threads save the application and hidden state. Before performing the second checkpoint barrier, each thread reacquires all of the resources 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` 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. This is because the OpenMP specification precludes the use of the barrier directive in certain constructs (e.g., work-sharing, critical, ordered or master). Since developers may place checkpoints within these constructs, we have reimplemented barriers in the routine `ccc_global_barrier` to avoid such violations.

The following sections expand this high level idea into a complete protocol for OpenMP, describing how locks, critical regions and ordered regions are released and reacquired and how barriers are released.

## 5.3 Incorporating Locks

The $C^3$ precompiler replaces all calls to OpenMP locking routines (e.g., `omp_set_lock`, `omp_unset_lock`) 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 6.

```
release_lock(omp_lock_t *l) {
    omp_unset_lock(l);
}
reacquire_lock(omp_lock_t *l) {
    omp_set_lock(l);
}
void ccc_set_lock(omp_lock *l) {
    omp_set_lock(*l);
    while (checkpoint_initiated) {
        omp_unset_lock(*l);
        ccc_perform_checkpoint(FALSE);
        omp_set_lock(*l);
    }
    record_resource_held(l);
}
void ccc_unset_lock(omp_lock_t *l) {
    remove_resource_held(l);
    omp_unset_lock(l);
}
```

**Figure 6: Checkpointing locks**

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 the `checkpoint_initiated` flag is read as false and and the lock acquisition is complete. To release a lock we record the fact that the thread no longer owns the lock and call `omp_unset_lock` to release the OpenMP lock itself.

```
while(1) {
    ccc_set_lock(&l);
    ccc_potential_checkpoint();
    ccc_unset_lock(&l);
}
```

**Figure 7: Locks example**

For a more intuitive understanding of the protocol as it applies to locks, consider the code in Figure 7 being executed by two threads. Suppose that at some point in time Thread 0 acquires lock $l$ and decides to take a checkpoint (i.e. calls `ccc_perform_checkpoint(FALSE)`) while still holding the lock. Meanwhile, thread 1 calls `ccc_set_lock` and blocks on lock $l$. In order to wake tread 1 up, thread 0 sets `checkpoint_initiated` to true and releases the lock. This causes thread 1 to acquire $l$, wake up and check the `checkpoint_initiated` flag. Since the flag is equal to true, thread 1 knows that it was woken up in order to be checkpointed. As such, it lets go of lock $l$ and takes a checkpoint. Meanwhile, thread 0 finishes its checkpoint, making sure to reacquire lock $l$ before the second checkpoint barrier, and resumes its execution. After thread 1 has finished its checkpoint (and passed the second checkpoint barrier), it attempts to acquire the lock again.

Since thread 1 may not acquire lock $l$ again before thread 0 decides to take another checkpoint, it may need to loop inside of *ccc_set_lock* multiple times before it can actually acquire the lock without being forced to checkpoint. Note that although thread 0 released lock $l$ in the middle of the lock-protected region of code, application semantics were not violated because the protocol guarantees that by the end of each checkpoint resources are always restored to their pre-checkpoint owners. (i.e. since thread 0 held $l$ when it called `ccc_perform_checkpoint`, it will hold $l$ when `ccc_perform_checkpoint` returns)

## 5.4 Incorporating Barriers

The $C^3$ precompiler replaces OpenMP barrier directives 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 8.

```
/*shared*/ int threads_awoken[2]
                = { FALSE, FALSE };
/*private*/ int barrier_id = 0;
void release_barrier() {
    threads_awoken[barrier_id] = TRUE;
    ccc_global_barrier();
    barrier_id = !barrier_id;
}
void ccc_barrier() {
    threads_awoken[!barrier_id] = FALSE;
    ccc_global_barrier();
    while (threads_awoken[barrier_id]) {
        barrier_id = !barrier_id;
        ccc_perform_checkpoint(TRUE);
        threads_awoken[!barrier_id] = FALSE;
        ccc_global_barrier();
    }
    barrier_id = !barrier_id;
}
```

**Figure 8: Checkpointing barriers**

The flag `threads_awoken` is used to determine whether a barrier completed because all threads arrived at the bar-

rier (=FALSE) or because a thread called release_barrier to awaken any thread that was blocked at the barrier (=TRUE).

The purpose of the at_barrier parameter to ccc_perform_checkpoint can now be explained. When a thread is released from a barrier to take a checkpoint, it does not need to call release_barrier since the fact that is was released in this way implies that it and all the other threads that were blocked on this global barrier have been awoken. Thus, all threads previously blocked on barrier calls are now awake and ready to take a checkpoint. As such, in the call to ccc_perform_checkpoint, release_barrier is skipped since at_barrier is true.

## 5.5 Incorporating Critical and Ordered

Since critical and ordered sections are conceptually similar to locks, they are incorporated into the framework in a similar fashion (Figure 6). There are several key differences.

First, OpenMP has no routines for explicitly acquiring or releasing critical or ordered sections. Instead, they are blocks of code tagged by a #pragma that identifies them as critical or ordered. As such, entering and exiting a critical or ordered section is done implicitly by entering and exiting the corresponding block of code. This means that in order to release and reacquire these resources the routines release_criticial/release_ordered and reacquire_criticial/ reacquire_ordered must execute code to exit and enter these code blocks.

For example, in the program 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. This is done by first jumping from the checkpoint location to the end of critical section B, exiting B, jumping to the end of critical section A, exiting A and finally taking a checkpoint. In reacquiring the critical sections the same things are done in reverse: the thread reenters critical section A, jumps to the start of critical section B, reenters B, jumps to the original checkpoint location and finishes the checkpoint.

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

**Figure 9: 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 [8].

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 using alloca in a manner similar to the approach in Section 4.1.3 for ensuring that stack regions are correctly reassigned on restart.

The case of ordered sections is complicated by their special semantics: if an ordered section appears inside a parallel for, its code must be executed in iteration order (i.e. if $i < j$ then the ordered section in iteration $i$ must be executed before the ordered section in iteration $j$). Suppose that thread $t_1$ has decided to take a checkpoint inside an ordered section in iteration $i$ while another thread $t_2$ is blocked on entry into an ordered section in a later iteration $j$. In order to wake up $t_2$ and force it to take a checkpoint, 1. $t_1$ must exit its ordered section and 2. we must make sure that all iterations upto $j$ are passed so that $t_2$ can wake up, enter its ordered section and realize that a checkpoint has begun. After the checkpoint has completed, the skipped iterations must be reallocated to some threads in a manner similar to how we restart parallel for loops and executed normally.

## 5.6 Implicit Synchronization

It is possible for an application to implement its own form of synchronization. (e.g. use the atomic directive together with a polling loop to implement barriers) Our checkpointing framework may deadlock on applications such such implicit synchronization. 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. Indeed, software-implemented protocols would have significant difficulty efficiently detecting such synchronization as this would require monitoring individual reads and writes to shared variables. Therefore, we use the Restrict methodology and prohibit the application from using its own form of synchronization.

# 6. EXPERIMENTAL EVALUATION

In this section we report on our implementation of the mechanisms described above. The $C^3$ implementation currently supports most of the OpenMP specification. The only important feature that has not yet been implemented are ordered sections. $C^3$ provides trivial support for nested parallelism in that only a single thread may be created inside a nested parallel region.

Application-level checkpointing increases applications running times in two different ways. Even if no checkpoints are taken, the instrumented code executes more instructions than the original application to perform book-keeping 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, we measure these two overheads separately.

To measure the overheads introduced by $C^3$ to a real application, we evaluate it using the OpenMP versions [1] of the NAS Parallel Benchmarks [2]. While NAS benchmarks give us general information, they provide little insight about how $C^3$'s transformations affect the overhead of OpenMP constructs. The EPCC microbenchmark suite [21] was used for this purpose.

To demonstrate the portability of our approach we ran experiments on three different platforms:

- 4-way 1.5Ghz Itanium with 2GB RAM and RedHat Enterprise Linux 4.0 with a version 2.6.9 kernel. Intel C++ Compiler V8.1 using the -O3 optimization level.
- 4-way Compaq Alphaserver ES45 (1Ghz EV68 processors, 4GB RAM) running Tru64 V5.1A. Compaq C Compiler V6.5, with -O3 -fast optimization. (Lemieux cluster at the Pittsburgh Supercomputing Center).

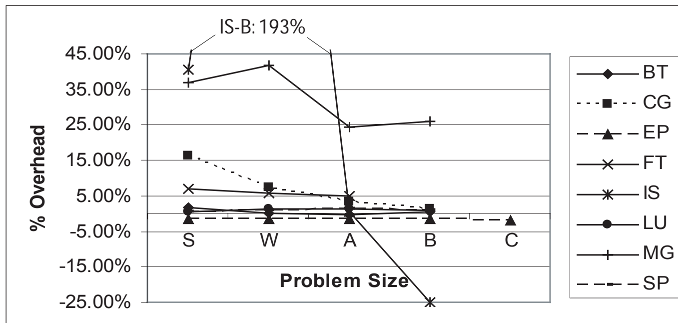


**Figure 10: NAS Linux/IA64 No-Checkpoint Overheads (40 runs each)**

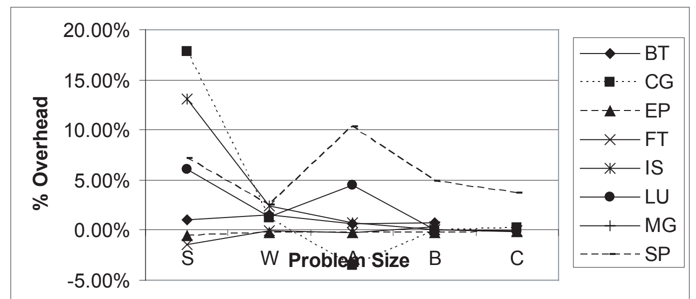


**Figure 11: NAS Tru64/Alpha No-Checkpoint Overheads (40 runs each)**

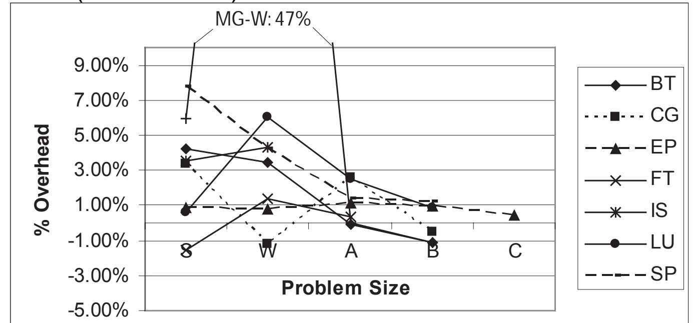


**Figure 12: NAS Solaris/Sparc No-Checkpoint Overheads (40 runs each)**

- 4-way Sun 420R (v9 SPARCII processors, 4GB RAM) running Solaris 9. SunPro 9 Compiler with `-fast` optimization level.

For all platforms, the experiments were run using 2 or 4 processors on the nodes. In this paper we only report the results of the 4-way experiments due to space constraints. Each experiment is the average of multiple runs with the upper and lower $10^{th}$ percentiles discarded. The number of runs in each experiment is listed with its table. To minimize error from accessing a networked disk, checkpoints were recorded to the local disk of each node. (efficient and reliable checkpoint storage is an orthogonal research topic of that is not addressed in this paper)

## 6.1 Application Overheads

We used the NAS benchmarks to evaluate $C^3$'s effect on the performance of realistic OpenMP applications.

### *6.1.1 Checkpoint-free Overheads*

Our first experiment measures the overhead of $C^3$'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 running time of the section of code performing the computational behavior represented by that benchmark. Results for the Linux/IA64, Tru64/Alpha and Solaris/Sparc runs are shown in Figures 10, 11 and 12, respectively. We show results for all benchmark/problem size combinations that we could run on each platform and for each combination show the % overhead from transforming the application to make it self-checkpointing, without taking any checkpoints. Results for MG on Alpha/Tru64 are missing because this code fails to run on this platform.

The tables show that for most codes, the overhead introduced by $C^3$ was generally small. There are a few important things that become apparent.

- In general, larger input sizes lead to smaller overheads. This happens because $C^3$'s transformations and bookkeeping may make OpenMP constructs more expensive. When running on larger inputs applications typically spend less time on OpenMP constructs and more time doing useful work, reducing $C^3$'s overhead. We believe this to be the typical real world behavior since HPC applications typically feature large inputs. Except for MG on Linux/IA64, the overhead for the largest input sizes is $< 4\%$ for all benchmarks on all platforms and $< 1\%$ in almost every case.
- In some cases overheads are negative. In almost every case they are tiny relative to the running time of the application and can be regarded as an expected variance due to modifications to the program's source code, which can be either positive or negative. The only exception is IS on Linux/IA64 with problem size B, where the no-checkpoint overhead is -25%. This happens because our transformations may take addresses of stack variables. In particular, by taking the address of array `prv_buff1` in function `rank` we cause the Intel compiler to perform some more aggressive optimizations that are otherwise not performed.
- MG on Linux/IA64 is the only case where the 0-checkpoint overhead is greater than a few % on all problem sizes. This is because $C^3$ creates aliases by taking addresses of variables that it wishes to checkpoint. While this can be avoided for most variables, it is not always possible and in MG on Linux/IA64 it resulted in high overheads. In order to avoid this overhead in the near future $C^3$ will transition to a different stack-saving mechanism that uses the `ucontext` family of functions to record and restore portions of the stack without taking the addresses of stack variables or using gotos.

### *6.1.2 Cost of Checkpointing*

To evaluate the cost of taking checkpoints, Tables 1, and 2 show the amount of time it takes to record a single checkpoint on Linux/IA64 and Tru64/Alpha, respectively. (Solaris/Sparc omitted due to space constraints) For each application/platform we show for the largest problem size that we could run, the absolute time to take a checkpoint (in seconds), the relative time as a percentage of the original application's running time and the size of the checkpoint. The checkpoint time was computed using the formula: $ChkptCost = 1ChkptTime - 0ChkptTime$, where $1ChkptTime$ = running time of the transformed applica-

tion where it takes a single checkpoint and $0ChkptTime =$ running time of the same transformed application without taking any checkpoints. For all the applications it does not matter when during the application's execution the checkpoint is taken.

The cost of checkpointing is generally low, on the order of a few seconds, rising beyond a few seconds only for checkpoints larger than 1GB. In cases where the original application runs for a very short time relative to the amount of state it uses, even these few seconds are large relative to the total program runtime. This emphasizes the need for compiler analyses to reduce checkpoint sizes.

In some cases the time to record a checkpoint is negative. This is generally a small fraction of the running time of the application itself and is an expected runtime variance due the non-trivial perturbation of the system caused by a checkpoint.

### *6.1.3 Cost of Restarting*

Computing the cost of restarting uses the formula:
$RestartCost = (BefChkpt{+}AftRestarting){-}0ChkptTime$.
$BefChkpt =$ time from the start of the application's main computation region (the region of code timed by the benchmark itself) until it began to take a checkpoint.
$AftRestarting =$ time from the beginning of the restart execution until the end of the application's main computation region.
$0ChkptTime =$ time to execute the transformed application without taking any checkpoints. Tables 1 and 2 show the cost of restarting on Linux/IA64 and Tru64/Alpha, respectively. (Solaris/Sparc omitted due to space constraints) In each case we show the absolute restart cost in seconds and the relative restart time as a percentage of the original application's running time.

For Tru64/Alpha, Linux/IA64 and Solaris/SPARC the restart overheads are consistently low. The only anomaly is the very negative restart times for CG and LU Tru64/Alpha, apparently caused by the restart process touching key arrays and causing the cache hit rates to improve. This appears to be a common effect on this platform, affecting both checkpoint and restart results of different NAS applications as the system underwent updates during the course of our experiments.

| Bench | Input Size | 1-Chkpt Time | 1-Chkpt Relative % | 1-Restart Time | 1-Restart Relative % | Chkpt Size |
|---|---|---|---|---|---|---|
| BT | B | 17s | 2.02% | 7.3s | 0.87% | 1.2GB |
| CG | B | 1.6s | 2.07% | 0.45s | 0.59% | 428MB |
| EP | C | -0.003s | 0.00% | 1.2s | 0.50% | 4.3MB |
| FT | A | 1.1s | 18.87% | 0.68s | 11.30% | 419MB |
| IS | B | 1.3s | 14.02% | 0.72s | 7.83% | 384MB |
| LU | B | -1.3s | -0.51% | -11s | -4.56% | 174MB |
| MG | B | 2s | 12.66% | 0.046s | 0.28% | 438MB |
| SP | B | 1.1s | 0.21% | 0.15s | 0.03% | 317MB |

**Table 1: NAS Linux/IA64 1-Checkpoint and Restart Times in Seconds and % of Runtime(35 runs each)**

## 6.2 Detailed Examination of Overheads

In this section we use the EPCC microbenchmarks (divided into Synchronization, Scheduling and Array) to examine in detail the overheads of $C^3$'s transformations on individual OpenMP constructs. We report the times recorded for three configurations:

- *Original*: the original microbenchmarks.

| Bench | Input Size | 1-Chkpt Time | 1-Chkpt Relative % | 1-Restart Time | 1-Restart Relative % | Chkpt Size |
|---|---|---|---|---|---|---|
| BT | B | 0.77s | 0.8% | -23s | -2.4% | 1.2GB |
| CG | C | -18.5s | -2.0% | -191s | -21% | 1.1GB |
| EP | C | 0.2s | 0.06% | -0.3s | -0.1% | 4.3MB |
| FT | B | 10.3s | 10.8% | 7.3s | 7.6% | 1.7GB |
| IS | C | 9.7s | 13.9% | 8.6s | 12.33% | 1.5GB |
| LU | C | 1.3s | 0.08% | -184s | -10.81% | 678MB |
| SP | C | 11.4s | 0.5% | -14.9s | -0.6% | 1.3GB |

**Table 2: NAS Tru64/Alpha 1-Checkpoint and Restart Times in Seconds and % of Runtime(40 runs each)**

- $C^3NoPChkpt$: the microbenchmarks after transformation by $C^3$ but with no potential checkpoint locations inside the code. In realistic applications most OpenMP directives will not contain a potential checkpoint location.
- $C^3PChkpt$: the microbenchmarks that have been modified to contain a potential checkpoint location inside each OpenMP construct being tested.

All times are in microseconds.

### *6.2.1 Synchronization*

The results of the Synchronization benchmarks for Linux/IA64, Tru64/Alpha and Solaris/Sparc are shown in Tables 3, 4 and 5, respectively. Below are the description of each microbenchmark and the lessons learned from it:

- `Parallel`: the cost of entering and exiting a parallel region. Since $C^3$ does not transform parallel regions with no potential checkpoint locations, they have the same cost under *Original* and $C^3NoPChkpt$. A parallel region that contains a potential checkpoint is significantly more expensive than one that does not because $C^3$ uses a number of global `threadprivate` variables with `copyin` to maintain accounting state (the overhead comes primarily from the `copyin`). While parallel regions are much more expensive under $C^3PChkpt$, they still only take up tens to a few hundred microseconds. Since most applications open few parallel regions, this overhead is unlikely to have any effect on realistic applications.
- `For`: the cost of a parallel for loop with a single iteration per thread. Because the for loop in this benchmark was followed by an implicit barrier, in $C^3NoPChkpt$ and $C^3PChkpt$ the implicit barrier is removed and it is followed by an explicit call to *ccc_barrier*. Since on all platforms *Original* and $C^3NoPChkpt$ have almost identical cost, it appears that this barrier transformation has little effect. Under $C^3PChkpt$ the cost of `for` is significantly larger because in addition to the barrier transformation 1. it has to record that a given for loop iteration has been allocated to a given thread and 2. as part of our Reimplementation of private variables, $C^3$ extracts the bodies of parallel for loops and turns them into functions. Any local variables used inside the body of the for are passed into the function as arguments.
- `Parallel For`: the cost of a `parallel for` directive, which is a `parallel` directive that contains a single parallel `for` loop with a single iteration per thread. $C^3$ breaks up `parallel for` directives into a `parallel` directive that contains a `for` directive. On Linux/IA64 the $C^3NoPChkpt$ version's `parallel for` costs twice as much as the native version, while on Solaris/Sparc it is 32% more and 0% on Tru64/Alpha. The cost for $C^3PChkpt$ is approximately the sum of the cost of `parallel` and `for` directives.
- `Barrier`: the cost of a single barrier. *Original* uses the native compiler's implementation of barrier while

$C^3 NoPChkpt$ and $C^3 PChkpt$ use the $C^3$ implementation. The cost of both implementations is similar on all platforms, with a maximum overhead of 37% on Solaris/Sparc. This is a minor overhead since the primary cost of a barrier is the time lost to threads waiting for each other to reach the barrier rather than the cost of the barrier itself. If this proves to be an important overhead for some applications, it would be trivial to create multiple implementations of barrier and use the fastest one for each given platform.

- `Single`: the cost of a `single` directive, which allocates a piece of code for execution by some thread, followed by an implicit barrier. This directive has the same cost in *Original* and $C^3 NoPChkpt$ (despite the same barrier transformation as with `for`) but is higher in $C^3 PChkpt$ (by 3 $\mu s$ - 14 $\mu s$) because of the additional accounting and privatization code inserted by $C^3$ in this version.
- `Critical`: the cost of a `critical` section. Again, *Original* and $C^3 NoPChkpt$ have critical sections of the same cost. On Linux/x86 and Tru64/Alpha critical sections are more expensive in $C^3 PChkpt$ because of the additional accounting code required to deal with potential checkpoint locations inside them. However, on Solaris the cost of a critical section under $C^3 PChkpt$ is strongly negative. The reason is the use of `setjmp` to exit critical sections at checkpoint-time. On this platform a call to `setjmp` in a critical section speeds up the execution of the delay loop the EPCC microbenchmarks put inside OpenMP constructs. Since `setjmp` is such a low-level function it is difficult to determine the source of this effect.
- `Lock/Unlock`: the cost of acquiring and releasing a lock. It is higher in $C^3 NoPChkpt$ and $C^3 PChkpt$ than in *Original* because of the additional testing performed by $C^3$ lock functions to determine whether a checkpoint has begun.
- `Atomic`: the cost of `atomic` directives. It is the same in all configurations because $C^3$ pulls the right hand side of the atomic update expression outside the `atomic` directive, as discussed in Section 3.4.
- `Reduction`: The additional cost of a parallel region with a reduction variable over that of a regular parallel region. The cost of reduction is very similar for *Original* and $C^3 NoPChkpt$. For $C^3 PChkpt$ we Reimplement OpenMP's reduction functionality, resulting in a few $\mu s$ overhead on Tru64/Alpha and Solaris/Sparc and a few $\mu s$ reduction in cost on Linux/IA64.

### 6.2.2 Scheduling

The Scheduling benchmarks measure the cost of using different parallel for iteration scheduling policies (i.e. which thread will get which loop iteration). The benchmark runs 128 iterations of the for loop on each thread and we report the cost on a per-iteration basis (i.e. total cost / 128). Since this benchmark's for loops are followed by an implicit barrier, $C^3$ performs the barrier transform described above.

The results for the Scheduling benchmarks for Tru64/Alpha are shown in Table 6, respectively. (Linux/IA64 and Solaris/Sparc omitted due to space constraints) Table columns correspond to different chunk sizes. For static scheduling it is possible to not specify a chunk size.

- `Static`: for a given chunk size `cksz`, iterations are allocated in deterministic round-robin order in units of `cksz` iterations. $C^3$'s barrier transformation causes little difference in the per-iteration cost under *Original* and $C^3 NoPChkpt$ on Linux/IA64 and Tru64/ Alpha and a noticeable but small (in absolute terms) difference on Solaris/Sparc.
  $C^3 PChkpt$ has a larger cost because $C^3$ performs additional accounting work to record the which iterations have been allocated to which threads. The current implementation is based on a generic linked list library and appears to have overheads that vary with the the number of iterations allocated in a contiguous block to each thread. While this has little effect on the NAS benchmarks, applications with many small loops may be affected by this overhead. It can be significantly reduced via simple compiler and runtime optimizations.
- `Dynamic`: same as static except that the thread to iteration mapping is done dynamically. The performance characteristics are the same as for Static.
- `Guided`: the first half of the iterations are allocated evenly among threads and subsequent allocations shrink exponentially in size until a minimum `cksz` is reached. The overheads for *Original* and $C^3 NoPChkpt$ are very similar. The overhead for $C^3 PChkpt$ is higher but stable for all chunk sizes.

### 6.2.3 Array

The Array benchmarks measure the cost of privatizing otherwise shared variables upon entry into a parallel region (i.e. creating a copy of the variable for each thread). The results of the Array benchmarks for Tru64/Alpha are shown in Table 7. (Linux/IA64 and Solaris/Sparc omitted due to space constraints) Table columns correspond to different sizes of the variable being privatized, growing in powers of 3. We report the difference in the cost of a parallel region with privatization and a regular parallel region with no privatization.

- `Private`: measures the cost using the `private` clause, which privatizes shared local variables. *Original* and $C^3 NoPChkpt$ have essentially the same cost, since $C^3$ does not transform parallel regions without checkpoints. $C^3 PChkpt$, shows significantly larger costs because of $C^3$'s reimplementation of private variables (as discussed in Section 4.2), which appears to be less efficient than the native implementation. $C^3$ converts the body of a parallel region into a function and passes the privatized variables by value into it, thus creating a private copy of the variable on each thread's stack. Optimizing this is a useful direction in which to focus our future efforts. On Tru64/Alpha it appears that `private` has a negative cost under *Original* and $C^3 NoPChkpt$ but this is very close to 0 $\mu s$ and is almost certainly due to noise.
- `FirstPrivate`: like `Private`, but the `firstprivate` clause initializes each privatized copy of a variable with its pre-parallel region value. Again, *Original* and $C^3 NoPChkpt$ have the same cost. $C^3 PChkpt$'s cost is higher but as the same as for the `Private` benchmark since the above implementation ensures `firstprivate` semantics. Note that the cost of `firstprivate` is constant for small variable sizes and increases linearly for large sizes. This is the cost of variable initialization copies.
- `Copyin`: much like `firstprivate`, but applies to global variables (marked `threadprivate`). $C^3$ Reimplements `threadprivate` variables by `malloc`-ing a copy for each thread and Reimplements `copyin` via `memcpy`. This apears to be slower than using the call stack or whatever the native OpenMP implementations are doing.

| Type | Parallel | For | Parallel For | Barrier | Single | Critical | Lock/Unlock | Atomic | Reduction |
|---|---|---|---|---|---|---|---|---|---|
| Original | 3.32 | 1.9 | 3.4 | 1.88 | 2.08 | 0.47 | 0.47 | 2.47 | 2.57 |
| $C^3NoPChkpt$ | 3.36 | 2.1 | 6.1 | 1.97 | 1.9 | 0.47 | 0.69 | 2.41 | 2.87 |
| $C^3PChkpt$ | 33 | 28 | 65 | 1.98 | 5.4 | 0.99 | 0.69 | 2.33 | 1.44 |

**Table 3: Linux/IA64 Synchronization Microbenchmarks (times in $\mu$sec) (60 runs each)**

| Type | Parallel | For | Parallel For | Barrier | Single | Critical | Lock/Unlock | Atomic | Reduction |
|---|---|---|---|---|---|---|---|---|---|
| Original | 4.5 | 1.69 | 6.46 | 1.72 | 1.2 | 0.98 | 2.95 | 0.17 | 1.47 |
| $C^3NoPChkpt$ | 4.53 | 2.28 | 6.33 | 2.05 | 3.13 | 0.99 | 3.91 | 0.17 | 1.45 |
| $C^3PChkpt$ | 150 | 111 | 211 | 2.06 | 6.39 | 4.2 | 3.87 | 0.17 | 3.82 |

**Table 4: Tru64/Alpha Synchronization Microbenchmarks (times in $\mu$sec) (90 runs each)**

# 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 [9] to support the full OpenMP specification. While [9] only deals with barriers and locks, this work addresses the issues that arise when trying to checkpoint all 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 have evaluated it's performance using both full benchmarks and detailed microbenchmarks. We have shown that code inserted by our system has a minimal impact on performance, and that checkpoint and restart costs are reasonable.

Finally, by showing experimental results from three different platforms, with different Operating Systems, Instruction Sets and OpenMP implementations, we have shown that ALC in general and the $C^3$ approach in particular is inherently portable. Indeed, in our experience it takes no more than a few days to port $C^3$ to a few platform. In practice, the variations in the dialects of C we need to parse are more difficult to deal with than any system issues. If $C^3$ were to move to an industrial strength compiler such as ROSE[20] even these difficulties would go away.

## 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 checkpointer 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 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 Replay cannot be used and it is necessary to Reimplement this piece of state and checkpoint this implementation directly. Finally, if Reimplementation proves 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 [10] is a way for system-level checkpointers to use Replay to checkpoint the state of the network. Finally, this framework extends even to hardware-level checkpointers 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

With the rise of grid computing, checkpointing in a heterogeneous computing environment (e.g., checkpoint on IA-32/Windows, restart on Alpha/Tru64) is growing in importance. In [15] we present a system that uses compile-time type analysis to determine a unique type for each piece of application state, which is used to translate between different machine representations on restart. In [15] this analysis is combined with our work on checkpointing MPI applications. We plan to investigate an extension of this to OpenMP.

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.

# 8. REFERENCES


[1] http://phase.hpcc.jp/Omni/benchmarks/NPB.

[2] http://www.nas.nasa.gov/Software/NPB.

[3] R. K. Sahoo A. J. Oliner, L. Rudolph. Cooperative checkpointing: A robust approach to large-scale systems reliability. International Conference on Supercomputing (ICS) 2006.

[4] 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.

[5] OpenMP Architecture Review Board. OpenMP application program interface, version 2.5.

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level

| Type | Parallel | For | Parallel For | Barrier | Single | Critical | Lock/Unlock | Atomic | Reduction |
|---|---|---|---|---|---|---|---|---|---|
| Original | 8.1 | 3.5 | 9.8 | 1.9 | 1.3 | 0.55 | 0.49 | 0.89 | 2.1 |
| $C^3NoPChkpt$ | 8.3 | 4.2 | 13 | 2.6 | 2.1 | 0.56 | 1.0 | 0.89 | 2.2 |
| $C^3PChkpt$ | 89 | 72 | 161 | 2.8 | 16 | -2.2 | 1.0 | 0.92 | 9.5 |

**Table 5: Solaris/Sparc Synchronization Microbenchmarks (times in $\mu$sec) (60 runs each)**

| Static | No Chunk Size | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|---|
| Original | 0.013 | 0.020 | 0.017 | 0.021 | 0.019 | 0.017 | 0.017 | 0.018 | 0.019 |
| $C^3NoPChkpt$ | 0.018 | 0.027 | 0.023 | 0.024 | 0.022 | 0.021 | 0.021 | 0.021 | 0.021 |
| $C^3PChkpt$ | 1.2 | 19 | 11 | 6.74 | 4.5 | 2.9 | 2.0 | 1.5 | 1.3 |
| Dynamic | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
| Original | 0.72 | 0.47 | 0.35 | 0.27 | 0.22 | 0.11 | 0.038 | 0.030 | |
| $C^3NoPChkpt$ | 0.73 | 0.54 | 0.40 | 0.34 | 0.24 | 0.11 | 0.056 | 0.051 | |
| $C^3PChkpt$ | 7.1 | 6.3 | 5.3 | 4.2 | 2.95 | 1.9 | 1.4 | 1.2 | |
| Guided | 1 | 2 | 4 | 8 | 16 | 32 | | | |
| Original | 0.070 | 0.067 | 0.063 | 0.061 | 0.057 | 0.043 | | | |
| $C^3NoPChkpt$ | 0.086 | 0.083 | 0.079 | 0.078 | 0.072 | 0.059 | | | |
| $C^3PChkpt$ | 1.8 | 1.7 | 1.6 | 1.5 | 1.4 | 1.4 | | | |

**Table 6: Tru64/Alpha Scheduling Microbenchmarks (times in $\mu$sec) (90 runs each)**

| | | 1 | 3 | 9 | 27 | 81 | 243 | 729 | 2187 | 6561 | 19683 | 59049 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Private | Original | -0.066 | -0.078 | -0.074 | -0.065 | -0.058 | -0.065 | -0.076 | -0.10 | -0.072 | -0.0073 | -0.056 |
| | $C^3NoPChkpt$ | -0.055 | -0.039 | -0.061 | -0.069 | -0.071 | -0.053 | -0.027 | -0.018 | -0.018 | -0.016 | 0.067 |
| | $C^3PChkpt$ | 4.1 | 4.4 | 6.7 | 2.01 | 0.68 | 7.9 | 9.0 | 11.9 | 13.0 | 283 | 286 |
| Firstprivate | Original | -0.065 | -0.027 | 0.035 | 0.09 | 0.17 | 0.45 | 1.27 | 3.8 | 40 | 73 | 219 |
| | $C^3NoPChkpt$ | -0.041 | -0.061 | -0.02 | 0.066 | 0.16 | 0.43 | 1.21 | 3.5 | 23 | 73 | 222 |
| | $C^3PChkpt$ | 2.46 | 2.1 | 3.9 | -0.04 | -0.63 | 5.7 | 5.9 | 8.3 | 8.7 | 283 | 287 |
| Copyin | Original | 3.0 | 2.5 | 2.6 | 2.5 | 2.7 | 2.8 | 3.6 | 6.9 | 25 | 74 | 224 |
| | $C^3NoPChkpt$ | 2.5 | 2.5 | 2.6 | 2.6 | 2.7 | 2.9 | 3.6 | 7.3 | 25 | 75 | 225 |
| | $C^3PChkpt$ | 17 | 16 | 13.2 | 10.1 | 10.8 | 21 | 25 | 44 | 80 | 1346 | 2925 |

**Table 7: Tru64/Alpha Array Microbenchmarks (times in $\mu$sec) (90 runs each)**


checkpointing of MPI programs. In *Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[7] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing*, 2003.

[8] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. A protocol for application-level checkpointing of OpenMP programs. Technical report, Cornell Computer Science, 2005.

[9] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level checkpointing for shared memory programs. In *Conference on Application Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[10] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *IEEE Transactions on Computing Systems*, 1985.

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

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

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

[14] 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, October 1996.

[15] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile MPI programs in computational grids. Principles and Practice of Parallel Programming (PPoPP), 2006.

[16] 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.

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

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

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

[20] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. Conference on Parallel Compilers (CPC2000), 2000.

[21] Fiona J. L. Reid and J. Mark Bull. OpenMP microbenchmarks version 2.0. In *European Workshop*

*on OpenMP*, 2004.

[22] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *Supercomputing 2005*, 2004.

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

[24] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *International Parallel Processing Symposium (IPPS)*, Honolulu, Hawaii, 1996.