

User-level Checkpointing of POSIX Threads*

William R. Dieter

James E. Lumpp, Jr.

Department of Electrical Engineering

University of Kentucky

Lexington, KY 40506, USA

{dieter,jel}@dcs.uky.edu

Abstract

Multiple threads running in a single, shared address space is a simple model for writing parallel programs for symmetric multiprocessor (SMP) machines and for overlapping I/O and computation in programs run on either SMP or single processor machines. Often a long running program's user would like the program to save its state periodically in a checkpoint from which it can recover in case of a failure. Previous user-level checkpointing libraries to checkpoint Unix processes do not support multithreaded programs. This paper describes a user-level checkpointing library to checkpoint multithreaded programs that use the POSIX threads library provided by Solaris 2.

The checkpointing library increases the amount of time required to make some thread library calls. The checkpointing library added between less than 1 % and 10 % to the execution times of tested benchmark programs. Saving the program's state to a checkpoint further increased the execution time, but the percentage of total execution time taken to save checkpoints can be adjusted by adjusting the checkpoint interval.

1. Introduction

A multithreaded program's state can be divided into private state and shared state. A thread's *private state* includes its program counter, stack pointer, and registers. Its *shared state* includes everything common to all threads in the process, such as the address space and open file state. A multithreaded checkpointing library must save and recover the program's shared state and each thread's private state.

User-level thread libraries are implemented outside the kernel using timers to preempt threads when their time slice is over. Implementing a checkpointing library for a user-level threads package is a straightforward extension of a

single-threaded checkpointing library because a user-level multithreaded process is no different from a single-threaded process from the operating system's point of view. User-level threads cannot take advantage of a symmetric multiprocessor (SMP), however, because the kernel is not aware of the threads. Thus it cannot schedule them to run concurrently on separate processors.

With *kernel-level threads*, the kernel schedules threads and keeps track of their state. Not only must the checkpointing library save and restore the address space of the process to recover the thread state, but it must also call the kernel to restart threads during recovery.

Section 2 describes how programmers and users use the checkpointing library. Section 3 presents the design and implementation of such a library for Unix programs using the POSIX threads interface on Solaris 2. Section 4 shows how the checkpointing library affects the performance of multithreaded programs. Section 5 discusses related work.

2. Usage

Adding checkpointing support to a C program is straightforward. A programmer must add one line to include the checkpoint header file, for example:

```
#include "checkpoint.h"
```

and one line to call checkpoint initialization at the beginning of the main function, for example:

```
checkpoint_init(&argc, argv);
```

Checkpoint initialization must come before any calls to thread or file library calls because it initializes data structures that thread and library functions require.

The user can control the checkpoint period by passing optional command line arguments to the checkpointing library. The checkpointing library reads all the arguments after the "--" argument. For example,

```
% prog -- -t period
```

*This research was supported by the National Science Foundation under Grant CDA-9502645 and the Advanced Research Projects Agency under Grant DAAH04-96-1-0327.

runs the *prog* program with a checkpoint period of *period* seconds. A checkpoint period of 0 disables checkpointing. Checkpoints are automatically stored in *prog.chkpt.n* where *prog* is the name of the program and *n* is the checkpoint number.

To recover from a checkpoint, the user runs the program with the recovery option and specifies a checkpoint file. For example,

```
% prog -- -r prog.chkpt.n
```

runs the *prog* program, loading the state from the checkpointing file *prog.chkpt.n*. This simple interface provides the capability to implement automatic fault tolerance [10, 8] and debugging by repeatedly restarting from a checkpoint.

3. Implementation

Our checkpointing library is implemented at the user level to improve its portability. The checkpointing library has some difficulties because it is at the user level rather than in the kernel. It must intercept some system and library calls to record information that would be directly available to the kernel. Even so, the checkpointing library does not save some parts of the program's state. In addition to limiting the amount of state saved, the library uses asynchronous signals which can cause problems that do not normally occur in multithreaded programs.

3.1. Portability

Modifying the operating system kernel to support checkpointing would tie the checkpointing library to the operating system and would require the user to install a modified operating system kernel. Many users are not willing or not able to install a custom kernel, particularly if they are using a remote machine owned by someone else. Therefore, the library described here is implemented entirely at the user level.

While complete portability is the goal, parts of the checkpointing library must be different on different platforms. We isolate and minimize the operating system dependent parts of the library. For example, the checkpointing library uses the */proc* filesystem to find information about how a process's memory segments are mapped into its address space. Different operating systems provide different interfaces to access similar information through the */proc* filesystem. The code that reads memory mappings from */proc* is called by an operating system independent interface so it can be ported to another operating system without affecting the rest of the checkpointing library.

During recovery the main thread uses the `read` system call to read memory mapping information from the checkpoint file and the `mmap` system call to remap the process's address space from the checkpoint file. Solaris uses wrapper functions in a shared library around all its system calls. The jump table for functions in shared libraries is in the data segment, but the code is in the code segment for the library. The pointer in the jump table may change when the main thread remaps the data segment from the checkpoint during recovery, but the corresponding code for `mmap` and `read` may not be mapped yet if the `mmap` and `read` system calls are in a dynamically linked library.

The main thread must use a statically linked version of `read` and `mmap` to remap the process's segments. During recovery, the system calls are made using an assembly language routine to bypass the shared library. Using the assembly language routine may also bypass some synchronization that makes the `mmap` and `read` system calls thread safe. Thread safety is not a problem, however, because only the main thread uses the assembly language versions, and it only uses them when it is remapping memory from a checkpoint. The main thread is guaranteed to be the only thread running when it is remapping segments.

3.2. Limitations

Our checkpointing library supports programs that access regular files sequentially or use signal handlers for signals. However at least one signal must be available for the checkpointing library (`SIGUSR1` by default). The library does not support programs that randomly access files or communicate with other processes, but the programmer can add recovery code to the such programs to recover the file or communication state. Eventually we plan to use the checkpointing library to provide checkpointing to the Unify distributed shared memory system which is implemented as a multithreaded library [5].

POSIX threads do not provide a way to create a thread with a particular thread identifier. The checkpointing library assumes that the thread library always assigns thread identifiers in the same order. As long as the thread library allocates thread identifiers in the same order when recovering from a checkpoint as it did when the program created the threads before the checkpoint each thread will have the same thread identifier. If a thread exits before the end of the program, the checkpointing library must create a dummy thread to compensate for the thread that exited during recovery. Currently the thread library does not create dummy threads to compensate for threads that have exited, but such support could added.

During recovery, described in detail later, the checkpointing library restarts all the threads and restores the entire process address space from the checkpoint, including

the thread library data segment. The checkpointing library assumes that the thread library will function correctly with the newly created threads and the thread data structures from before the checkpoint. This assumption could cause problems if the thread library stores information that is not restored, such as the process id.

To make this assumption true for Solaris, all threads must be created with a system wide contention scope. POSIX threads can be created with either *system wide contention scope*, meaning that they compete with every process in the system to be scheduled by the kernel, or with *process wide contention scope*, meaning that they compete within the process to be scheduled. Under the Solaris thread model each thread with system wide contention scope is assigned to a lightweight process when it is created. Threads with process contention scope are scheduled within a lightweight process by default, but the thread library can create additional lightweight processes in which to schedule the threads.

The checkpointing library has no way through the POSIX interface of telling when the thread library creates extra lightweight processes. Thus it cannot reproduce them when recovering from a checkpoint. The checkpointing library forces all threads to be created with a system wide contention scope by intercepting calls to `pthread_create` and forcing the contention scope attribute to be `PTHREAD_SCOPE_SYSTEM`. Some programs may run more slowly with a system wide contention scope than with a process contention scope because switching between threads in different lightweight processes requires more work than switching between threads in the same lightweight process.

The checkpointing library does not explicitly handle the POSIX threads functions that handle thread specific data (such as, `pthread_key_create`, `pthread_setspecific`, `pthread_getspecific`, `pthread_key_delete`). The thread specific data functions will work if they only manipulate data in the process's address space, but not if any information is stored in the kernel. They could be supported by intercepting the calls and storing the thread specific data in the process's address space so it can be restored during recovery. Support may be added in the future if there is demand.

Our checkpointing library supports thread synchronization with mutexes and condition variables, but it does not currently support POSIX semaphores (unless they are implemented using mutexes and condition variables). Semaphore support may be added later if there is demand.

Thread cancellation functions and functions that set thread scheduling priorities are not supported. Support may be added in the future if there is demand.

3.3. Handling Asynchronous Signals

Asynchronously interrupting threads with signals usually leads to problems. An interrupted thread may have locked a mutex. If the thread tries to lock that mutex in the signal handler then it will deadlock. The checkpointing library does not have to worry about this because it never locks a mutex that an application thread may have locked when it was interrupted (without first unlocking it).

How threads handle signals in different thread packages is not well defined. In Solaris 2, when a thread receives a signal while it is blocked in a `pthread_mutex_lock` call it handles the signal then returns and blocks waiting to lock the mutex. In some thread packages a `pthread_mutex_lock` call might not be interrupted by a signal, or in others the thread may return from the `pthread_mutex_lock` call without locking the mutex. In either case our checkpointing library will work because the checkpointing library prevents the checkpointing thread from sending a signal to a thread while it is blocked waiting to lock a mutex. The checkpointing library must prevent a checkpointing signal during a lock anyway because it must do some bookkeeping and lock the mutex atomically.

Condition variables also present a problem. In Solaris 2 a thread will not unblock from `pthread_cond_wait` if another thread has locked the mutex that the first thread released when it blocked on the condition variable. For example, consider the following case:

1. Thread 1 locks mutex M
2. Thread 1 blocks on a condition, unlocking mutex M
3. Thread 2 locks mutex M
4. Both threads receive the checkpointing signal
5. Thread 2 runs the signal handler and waits for thread 1

When a thread unblocks from a condition variable it locks the mutex it unlocked when it blocked. Thread 1 cannot unblock from the condition variable to run the signal handler because to unblock it must relock mutex M, which thread 2 has locked. If thread 2 unlocks mutex M when it enters the signal handler, thread 1 can run the signal handler when it receives the signal. However, thread 1 must not access the data protected by mutex M, and thread 2 must relock mutex M before any thread tries to access data protected by mutex M.

The checkpointing library handles this situation by tracking thread calls while the program runs so it can unlock and relock mutexes during a checkpoint to preserve the program's correctness. The checkpointing library intercepts thread library calls to track how many threads exist, their identifiers, which threads are blocked on condition variables, and which mutexes they have locked in the *thread*

table. The checkpointing library also intercepts file open and close calls to track the file name associated with each file descriptor. So the open file state can be restored from a checkpoint.

3.4. Normal Operation

To save a checkpoint, the checkpointing library needs to know how many threads exist, what their thread identifiers are, and which mutexes they have locked. None of this information is directly available from the thread library. Therefore, the checkpointing library must intercept thread library calls to track this information. It tracks calls by replacing thread library functions with the same name, and calling the real thread library after doing bookkeeping.

The checkpointing library stores its bookkeeping information in the thread table. The thread table contains an entry with the thread identifier, thread attributes, thread status, a place to store thread private state, and a linked list of mutex information structures for each thread. The table is designed so that a thread can access its own entry quickly without having to lock a mutex. A thread can retrieve its table entry in the time it takes to do a modulus operation and an array access. The thread corresponding to a table entry is the only thread that accesses most of the entry's fields, so it does not need mutual exclusion to access them.

The mutex information structure has a pointer to the mutex, a set of mutex status flags, and a pointer to the next mutex information structure. When a thread locks a mutex, it looks up its entry in the table and adds a mutex information structure to the head of the locked mutex list. A thread unlocking a mutex searches its locked mutex list for the mutex and removes it.

Searching the mutex information list should be quick because threads typically lock a small number of mutexes at a time. Moreover, threads typically unlock mutexes in the opposite order they were locked so the mutex to unlock will usually be at the head of the mutex information list.

When a thread blocks on a condition variable, it searches the locked mutex list for the mutex associated with the condition variable. It then sets a flag in the mutex information structure to indicate the thread is blocked on a condition wait associated with that mutex.

The checkpointing library also intercepts file open and close calls to keep track of the file name associated with each file descriptor. The checkpointing library keeps an array the same size as the file descriptor table. Whenever the program calls `open`, the checkpointing library stores the arguments passed to `open` in its table. File table entries are removed when the program calls `close`. The checkpointing library uses the file table to reopen every file with its original arguments upon recovery.

3.5. Saving a Checkpoint

The call to `checkpoint_init` initializes checkpointing variables and starts the *checkpoint thread*, the thread that initiates checkpointing. The checkpoint thread blocks during normal operation. It unblocks periodically to start a checkpoint. The checkpoint proceeds as follows:

- 1. Set the `prevent_locks` flag.** The checkpoint thread first sets the `prevent_locks` flag to prevent any thread from locking a mutex. Without the flag, a thread that receives the `SIGUSR1` later than other threads may lock a mutex that a thread in the checkpointing signal handler has unlocked. Allowing the locking thread inside the critical region violates the synchronization of the program. When all threads have finished unlocking their mutexes, they must all be running in the signal handler, so there is no danger of a thread locking a mutex it should not lock.
- 2. Send `SIGUSR1` to all threads.** The checkpointing thread then sends a `SIGUSR1` signal to every thread, including itself. When a thread receives a `SIGUSR1`, it runs the `SIGUSR1` handler unless it is blocked on a condition variable and another thread has locked the mutex associated with the condition variable (see section 3.3).
- 3. Unlock mutexes.** Each thread unlocks every mutex it has locked and sets a flag indicating the mutex has been unlocked. Unlocking mutexes allows threads that blocked because a mutex was locked to enter the signal handler. Some threads may continue executing while others enter the checkpoint signal handler because Unix does not guarantee when signals will be delivered. Threads that have not received a `SIGUSR1` are prevented from locking a mutex unlocked by a process in the signal handler because of the `prevent_locks` flag. The checkpointing thread uses thread synchronization to guarantee every thread will see the `prevent_locks` flag set before locking the mutex in the intercepted mutex lock call.
- 4. Wait for the child threads.** The main thread waits for all the child threads to signal they are ready.
- 5. Save the private thread state.** Each child thread writes its private state to its entry in the thread table using `sigsetjmp`.
- 6. Signal the main thread.** Each child thread indicates that it has saved its private state.
- 7. Wait for the main thread.** The child threads block until the main thread signals them.

8. **Save file state** The main thread records the current offset of the file pointer in each open file.
9. **Clear the `prevent_locks` flag.** The main thread clears the `prevent_locks` flag.
10. **Save the process state.** The main thread saves the thread table, the process's memory mappings, and the process's address space to a checkpoint file.
11. **Signal child threads.** The main thread wakes up the child threads.
12. **Relock mutexes.** Each thread relocks the mutexes it unlocked in step 3 in two phases as follows:

(a) **Lock held mutexes (locking phase 1).** Each thread locks all the mutexes it had locked when the checkpoint started. Each thread waits until every thread has finished phase 1 before continuing with phase 2.

No two threads will try to lock the same mutex because only one thread could have held each mutex when the checkpoint thread sent `SIGUSR1` signals to every thread. Once the thread unlocks its mutexes in step 3 above no other thread can lock them before entering the signal handler because the checkpoint thread sets the `prevent_locks` flag before signalling any thread.

(b) **Lock condition wait mutexes (locking phase 2).** Each thread attempts to lock the any mutexes it was blocked waiting to lock when the checkpoint started along with mutexes it released before the checkpoint to wait on a condition variable. In phase 2, a thread may try to lock a mutex that another thread has already locked. In that case it blocks while locking the mutex in the signal handler, but this is the desired result because it was blocked on a condition variable or trying to lock a mutex when the thread got the signal. At this point the program continues as if the checkpoint had not happened.

Thread synchronization functions are not guaranteed to be safe to call in a signal handler [1, section 6.6]. However, threads must synchronize during a checkpoint and must unlock mutexes to allow all threads to run. We assume locking a mutex in a signal handler is safe as long the mutex has not already been locked by the same thread, and unlocking a mutex in a signal handler is safe as long as it was locked by the same thread.

The functions that intercept the mutex lock and mutex unlock thread library calls use thread synchronization primitives to prevent the checkpointing thread from starting a

checkpoint. Preventing a checkpoint allows threads to do bookkeeping before and after the thread library call without being interrupted by a `SIGUSR1`. It also prevents the thread library locking functions from being called re-entrantly.

Unfortunately, calling thread synchronization functions in the signal handler is unavoidable. The threads must synchronize during a checkpoint, which includes unlocking mutexes held by some threads. The program could implement some of the synchronization using busy waiting, but busy waiting is inefficient and may interact with the memory system in unexpected ways. Moreover, the program must still unlock and lock mutexes to allow all threads to run the signal handler.

Figure 1 shows an example of a multithreaded process saving its state in a checkpoint. The program takes the following steps in the figure.

1. The checkpoint thread sets the `prevent_locks` flag to prevent the application from locking mutexes.
2. The checkpoint thread sends a Unix signal to every thread.
3. Each thread enters the signal handler and unlocks the mutexes it was using. Some thread may have been blocked on a condition variable while another thread had the associated mutex locked. The blocked thread does not handle the signal until the mutex associated with the condition variable on which it was blocked is unlocked.
4. Each thread saves its local state (PC, stack frame, registers, etc.).
5. The main thread waits until all threads have finished step 4.
6. All child threads wait until the main thread finishes step 8.
7. The main thread clears the `prevent_locks` flag.
8. The main thread saves the process's global state.
9. Each thread relocks the mutexes it held when it received the Unix signal from the checkpoint thread. None of the threads will block because only one thread locked each mutex before the checkpoint.
10. All threads wait at a barrier until step 9 is finished.
11. Each thread tries to lock mutexes it was attempting to lock and mutexes associated with a condition variable on which it was waiting when it received the Unix signal. Some threads may block, but will eventually run.
12. Each thread returns from the signal handler when it has completed step 11.

3.6. Restoring From a Saved Checkpoint

When a program recovers from a checkpoint it starts out as a single threaded program. During initialization, the checkpoint library restores the program's state using the following steps.

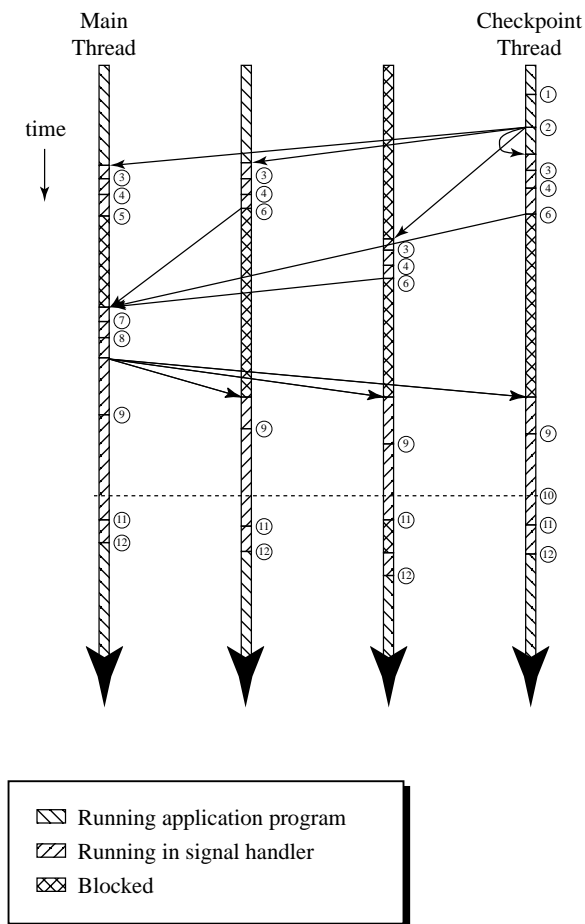


Figure 1. An example of a program saving its state. Each vertical line represents the progress of a thread. Arrows between threads show dependencies between threads.

- Restart threads.** The main thread opens the checkpoint file, reads the saved thread table, and restarts a new thread for each thread in the original program. Each new thread's stack starts at the same address as the stack of the corresponding thread in the original program.

- Restore thread stacks.** The main thread waits while the child threads restore their stack pointers. Each thread restores its stack by calling `siglongjmp` to restore its private state. Calling `siglongjmp` causes the thread to return from the `sigsetjmp` call it made when it saved its state in the checkpoint. The threads move their stack pointers before the main thread loads the address space because the act of moving them needs to use local variables, which would corrupt the stacks if they were loaded first.

- Wait for the main thread.** The child threads wait for the main thread to finish restoring the program's state.

- Restore main thread stack.** Once all of the child threads have blocked, the main thread restores its own stack pointer. The main thread stack is not necessarily as large as it was when the the program saved the checkpoint. Therefore the main thread recursively calls a function until the main thread's stack is as large as it was when it saved the checkpoint. The main thread can tell when its stack is large enough by comparing the address of a local variable to the address of a local variable when it saved its state. The child threads do not have this problem because their stack sizes are fixed when they are allocated (in Solaris 2).

- Remap the process's address space** The main thread then maps each segment in the program's address space except the main thread stack from the checkpoint file using the `mmap` system call similar to the method Condor uses [8]. The checkpointing library uses `mmap` to remap segments because `mmap` does not cause the data to be loaded immediately. The operating system demand loads the contents of the segments when the program accesses them. The checkpointing library uses `read` to reload the main thread's stack because mapping the main thread stack from a file causes problems in Solaris 2¹.

¹Normally the kernel dynamically increases the size of the stack by faulting in new pages when a program exceeds the current stack size. How-

6. **Restore main thread stack pointer** The main thread calls `siglongjmp` to return continue execution where the program was when it saved the checkpoint.
7. **Restore file state** The main thread opens all the files that were open during the checkpoint and moves the file pointer to its position at the time of the checkpoint. From this point on the threads follow the same steps they follow after the checkpoint file has been written to a file during checkpointing.
8. **Signal child threads.** The main thread wakes up the child threads.
9. **Relock mutexes.** Each thread relocks the mutexes it unlocked in step 3 in two phases as follows:
 - (a) **Lock held mutexes (locking phase 1).** Each thread locks all the mutexes it had locked when the checkpoint started. Each thread waits until every thread has finished phase 1 before continuing with phase 2. No two threads will try to lock the same mutex because only one thread could have held each mutex when the checkpoint thread sent `SIGUSR1` signals to every thread. Once the thread unlocks its mutexes in step 3 above no other thread can lock them before entering the signal handler because the checkpoint thread sets the `prevent_locks` flag before signalling any thread.
 - (b) **Lock condition wait mutexes (locking phase 2).** Each thread attempts to lock the any mutexes it was blocked waiting to lock when the checkpoint started along with mutexes it released before the checkpoint to wait on a condition variable. In phase 2, a thread may try to lock a mutex that another thread has already locked. In that case it blocks while locking the mutex in the signal handler, but this is the desired result because it was blocked on a condition variable or trying to lock a mutex when the thread got the signal. At this point the program continues as if the checkpoint had not happened.

4. Performance Results

The two main sources of checkpointing overhead are saving the checkpoint to disk and the intercepting thread. We ran several multithreaded programs with and without the checkpointing library to measure checkpointing overhead. We ran the tests on a single processor, 125 MHz

ever, the kernel does not dynamically increase the main stack size after a program has used `mmap` to map over the stack segment.

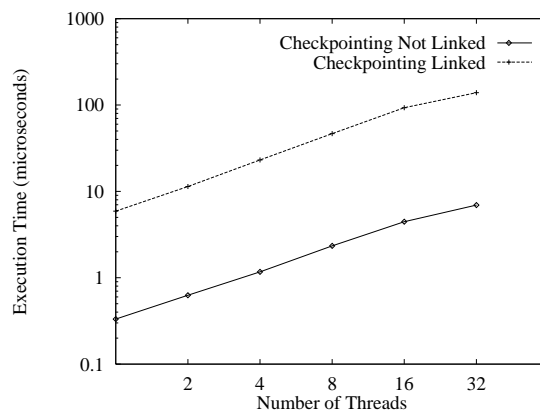


Figure 2. Average time to lock and unlock a mutex with and without checkpointing.

Sparc-20 with 64 MB of RAM running Solaris 2.6. Checkpoints were saved to an NFS mounted disk exported from an identical Sparc-20 over 100 Mbit/s Ethernet.

Two microbenchmarks show that most of the overhead comes from tracking mutex lock and mutex unlock thread library calls. In the first, called `ilock`, each thread locks and unlocks a different mutex one million times. Figure 2 shows the average amount of time to lock and unlock a mutex, calculated by dividing the total time by one million. Although the lock/unlock pair takes about 20 times longer with checkpointing, the ratio of the time with checkpointing to the time without checkpointing remains constant. Real application programs will do more work between mutex lock and unlock calls reducing the percentage of time spent locking and unlocking mutexes.

The second microbenchmark, `ring`, is a simple test program designed to measure the overhead of intercepting condition wait calls. In the benchmark threads take turns incrementing a shared counter variable. Each thread locks a mutex at the start of the program, then releases it by waiting on a condition variable. After a thread increments the counter, it signals the condition variable to wake up all the other threads. Each thread wakes up and checks the counter to see if it is its turn to increment the counter.

Each thread waits on the condition variable once per cycle. We ran `ring` with and without checkpointing for 65,536 condition wait calls. The `ring` program keeps the number of condition wait calls constant as the number of threads increases. Figure 3 shows the execution time of `ring` both with and without the checkpointing library linked. The program does not save any checkpoints in this test. `ring` and the lock test represents the worst case, in which a program only does synchronization and no work. Any real program would not have as much overhead.

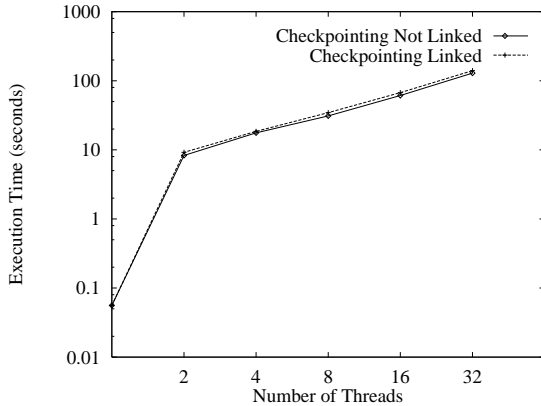


Figure 3. Execution time of the `ring` program as the number of threads increases. The execution time does not increase much without checkpointing until 16 threads. With checkpointing the execution time increases linearly with the number of threads.

Without checkpointing, execution time stays constant as the number of threads increases. With checkpointing linked, `ring`'s execution time is very close to that without checkpointing linked.

Figure 4 shows the performance of the checkpointing library with four application programs, Barnes, FMM, Radiosity, and Water-Spatial, from the SPLASH-2 benchmark suite [15]. Barnes simulates a group of particles interacting in three dimensions. FMM also simulates a group of particles, but in two dimensions. It also uses a different algorithm than Barnes. Radiosity renders a three dimensional scene. Water-Spatial simulates the potentials and forces between water particles. The “Checkpointing Disabled” curve shows the application’s execution time without any checkpointing overhead. As the number of threads increases, execution time of the program increases gradually because of thread synchronization. The “Checkpointing Enabled, No Checkpoints” curve shows the application’s execution time with the checkpointing library enabled, but without saving any checkpoints. The “Checkpointing Enabled, One Checkpoint” curve shows the program’s execution time when the checkpointing library saves one checkpoint during the program’s execution. This time includes the time to save the checkpoint to disk.

FMM and Water-Spatial took less than 1% longer on average with checkpointing enabled. Barnes with checkpointing enabled took 3% longer on average than with checkpointing disabled even though no checkpoints are saved. On average checkpointing added 10% to the execution time of Radiosity in the same case. The extra overhead with Radiosity is expected because it makes the most of the bench-

marks lock calls [15].

Figure 5 and Figure 6 show execution time and the speedup of the checkpointing library with the SPLASH-2 programs on a 50 MHz SMP Sparc-20 with 4 processors. As in the single processor case checkpointing adds little overhead to each program. Speedup of each benchmark is calculated relative to the benchmark run with one thread without checkpointing. The speedup curves have a similar shape to those without checkpointing. Radiosity has the most overhead of the benchmarks as in the single process case.

5. Related Work

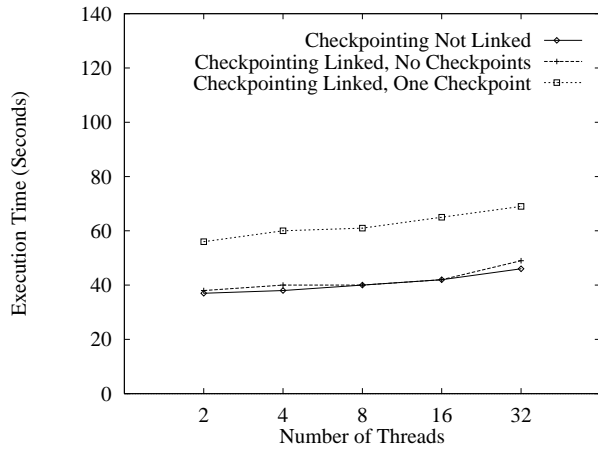
Libckpt, Condor, and libckp are checkpointing libraries developed to run on several versions of Unix. Libckpt’s features include incremental checkpointing, asynchronous checkpointing (saving a checkpoint while the program continues to run), user directed checkpointing (allowing the user to request a checkpoint), and user directed memory exclusion and inclusion [10]. Libckpt does not handle multithreaded programs or programs with multiple code and data segments. (Programs linked with shared libraries have a code and data segment for each shared library in Solaris 2.)

Condor was developed at the University of Wisconsin to provide process migration [12]. Condor migrates a process by forcing the process to save a checkpoint on one machine, then recover from the checkpoint on another machine. Condor runs on a number of operating systems including Solaris 2 and Linux. It does not support multithreaded programs or have freely available source.

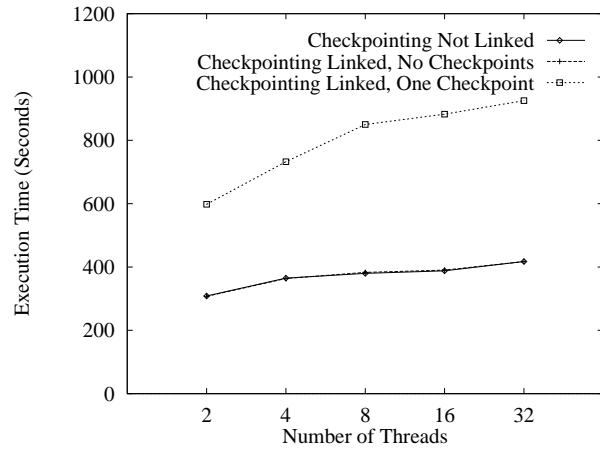
Libckp was developed at AT&T Bell Laboratories to checkpoint Unix processes [14]. Libckp saves files along with the checkpoint to guarantee they will be the same when the program recovers. It does not support multithreaded programs.

Plank and Li describe several algorithms for reducing latency when saving checkpoints [11]. Their algorithms assume the operating system provides a convenient way to stop threads and save and restore the private state of a thread. Their optimizations could be applied to our algorithm, but there would be several complications. Their algorithms assume that the threads can be stopped and their state saved from a separate process. With POSIX threads a forked child will only have one thread. Thus the state of the thread library in the new process will be different from that in the original process. The recovery would have to account for the difference.

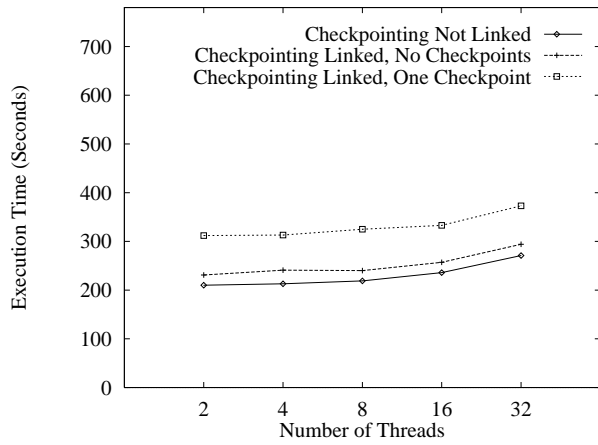
Many researchers have worked on checkpointing for distributed systems [2, 3, 4, 6, 7, 9, 13]. Distributed checkpointing algorithms are primarily concerned with forming consistent global checkpoints state either by coordinating all of the processors or by building a consistent checkpoint



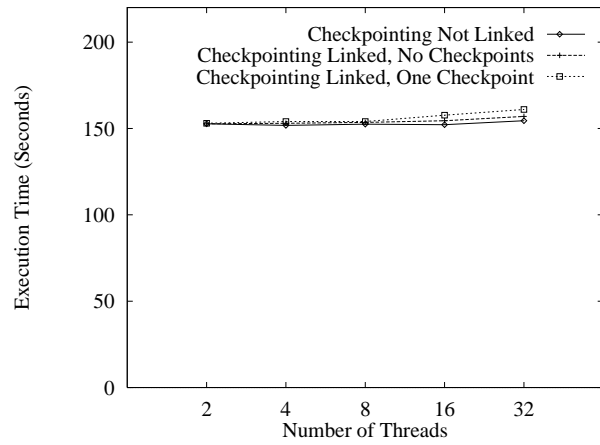
(a) Barnes



(b) FMM

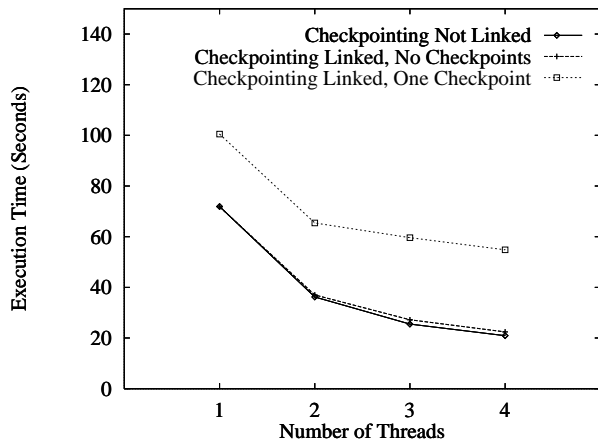


(c) Radiosity

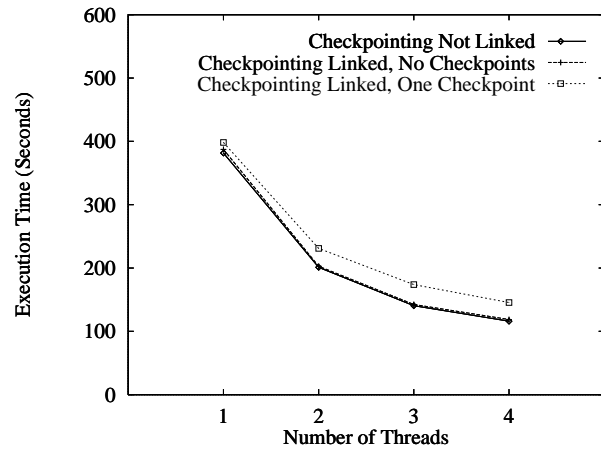


(d) Water-Spatial

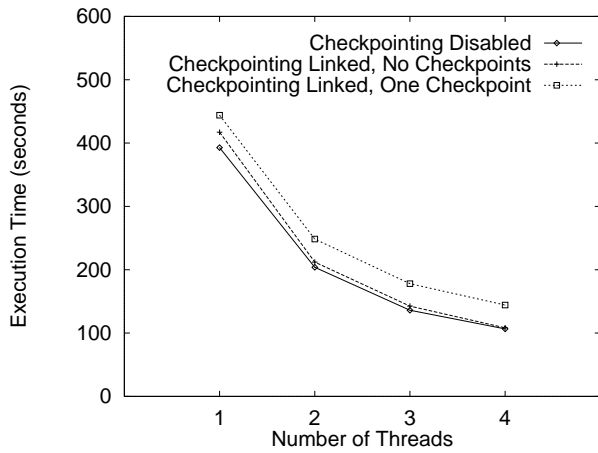
Figure 4. These graphs show how the execution time of four SPLASH-2 benchmark programs are affected by increasing the number of threads. The execution time does not decrease with more threads because the program was run on a single processor machine.



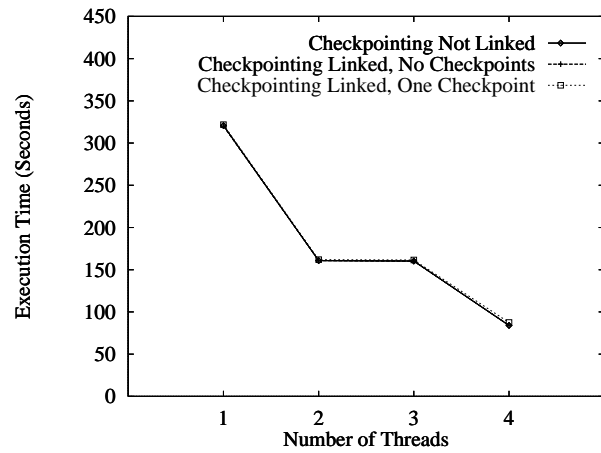
(a) Barnes



(b) FMM

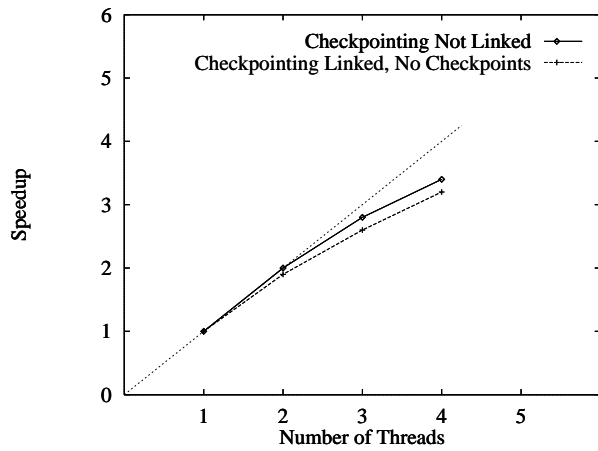


(c) Radiosity

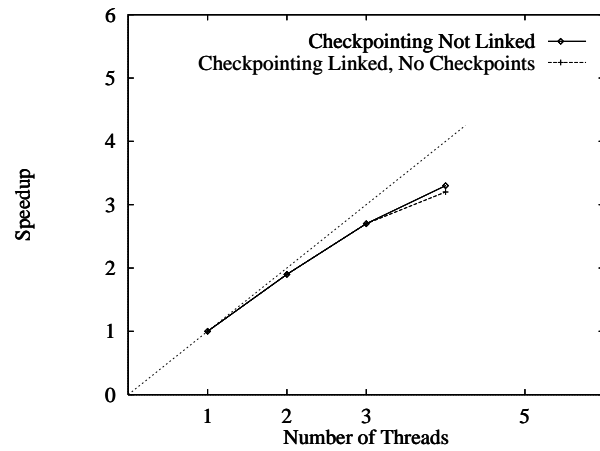


(d) Water-Spatial

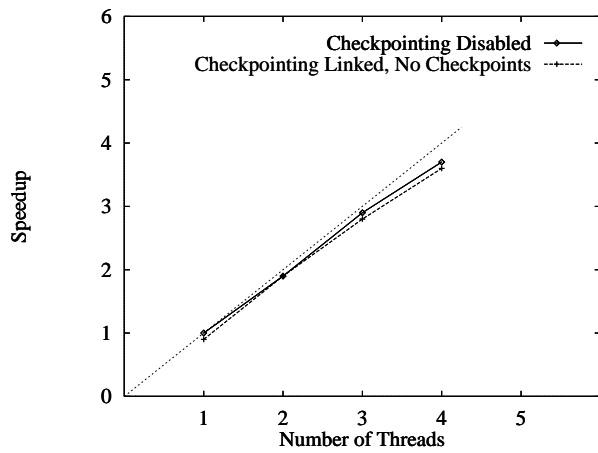
Figure 5. These graphs show how the execution time of four SPLASH-2 benchmark programs are affected by increasing the number of threads on a 4 processor Sparc-20.



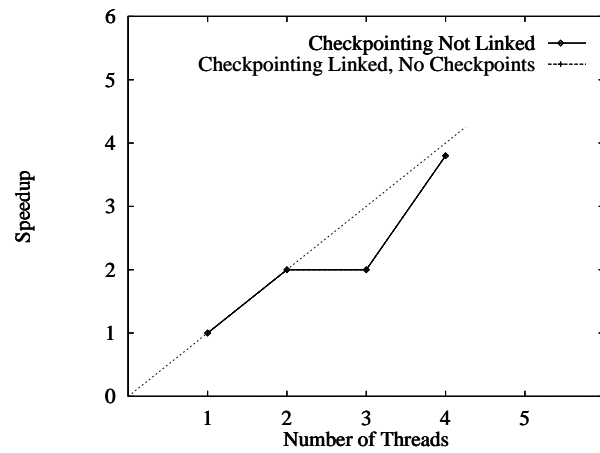
(a) Barnes



(b) FMM



(c) Radiosity



(d) Water-Spatial

Figure 6. These graphs show the speedup of four SPLASH-2 benchmark programs relative to the program run with one thread without checkpointing enabled.

from independent checkpoints. Our checkpointing library uses the thread library's memory consistency guarantees ensure checkpoints are consistent.

6. Conclusion

Our user-level checkpointing library can checkpoint multithreaded programs that use kernel-level threads. It added about 5 μ s to a mutex lock and unlock pair. For four programs from the SPLASH-2 benchmark suite checkpointing added less than 1% to 10% to execution time. Programs with similar synchronization patterns will incur similar overhead. Saving process state to disk incurs a substantial amount of overhead, and we are considering optimizations to reduce the overhead of writing a checkpoint to disk.

Currently we are working on reducing the overhead of intercepting thread library calls, integrating the checkpointing library into the Unify distributed shared memory library [5], porting the checkpointing library to other operating systems, and adding features found in other checkpointing libraries. Source for the checkpointing library is available had <http://www.dcs.uky.edu/~chkpt/>.

References

- [1] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
- [2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.
- [3] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 39–47, Oct. 1992.
- [4] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 298–307, June 1994.
- [5] J. Griffioen, R. Yavatkar, and R. Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2), 1995.
- [6] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, SE-13(1):23–31, May 1987.
- [7] P.-J. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the International Conference on Data Engineering*, pages 154–163, Feb. 1988.
- [8] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Science, Apr. 1997.
- [9] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, July 1997.
- [10] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, Jan. 1995.
- [11] J. S. Plank and K. Li. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, Aug. 1994.
- [12] T. Tannenbaum and M. Litzkow. Checkpointing and migration of unix processes in the condor distributed processing system. *Dr. Dobbs Journal*, Feb. 1995.
- [13] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, May 1995.
- [14] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 22–31, June 1995.
- [15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, June 1995.