USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

User-level Checkpointing for LinuxThreads Programs*

William R. Dieter and James E. Lumpp, Jr.
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, KY 40506, USA
{dieter,jel}@dcs.uky.edu, http://www.dcs.uky.edu/~chkpt

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. This paper introduces the first system to provide checkpointing support for multithreaded programs that use LinuxThreads, the POSIX based threads library for Linux.

The checkpointing library is simple to use, automatically takes checkpoint, is flexible, and efficient. Virtually all of the overhead of the checkpointing system comes from saving the checkpoint to disk. The checkpointing library added no measurable overhead to tested application programs when they took no checkpoints. Checkpoint file size is approximately the same size as the checkpointed process's address space. On the current implementation WATER-SPATIAL from the SPLASH2 benchmark suite saved a 2.8 MB checkpoint in about 0.18 seconds for local disk or about 21.55 seconds for an NFS mounted disk. The overhead of saving state to disk can be minimized through various techniques including varying the checkpoint interval and excluding regions of the address space from checkpoints.

1 Introduction

Computer systems are prone to hardware and software failures and the probability that a machine will crash before a process finishes running grows in proportion to the process's run time. A process can save its state in a checkpoint to help tolerate system downtime. A multithreaded process has both 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 process'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*, like LinuxThreads in Linux or lightweight processes in Solaris, 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.

Hybrid thread libraries like the one found in Solaris, use both kernel-level and user-level threads. User-level threads are scheduled to run inside several kernel-level threads, called light-weight processes (LWP) in Solaris. The library usually starts with one LWP per processor. If a user-level thread makes a blocking call and there are more runnable threads the thread library starts a new LWP so the whole process does not need to block. A hybrid thread library cannot be checkpointed like a user-threads library because it uses kernel-level threads.

*This research was supported by the National Science Foundation under Grant CDA-9502645, NFS/EPSCoR under Grant EPS-9874764, the Advanced Research Projects Agency under Grant DAAH04-96-1-0327, and the Kentucky Opportunity Fellowship.

We have tested our checkpointing library on several programs in the SPLASH-2 benchmark suite in addition to some simple test programs. The WATER-SPATIAL application ran with no noticeable overhead other than the time to save a checkpoint. It saved a 2.8 MB checkpoint to local disk in about 0.18 seconds or to an NFS mounted disk in about 21.55 seconds. The time to save a checkpoint to disk is about the same as the time required to copy a file of the same size as the checkpoint with the `cp` command.

Section 2 discusses related work and section 3 describes how programmers and users use the checkpointing library. Section 4 presents the design and implementation of such a library. Section 5 describes restrictions on programs using the checkpointing library. Finally, section 6 shows experimental results and performance.

2 Related work

Checkpointing is a popular way of providing fault-tolerance for computer systems. Both user-level and kernel-level systems have been developed for single threaded processes, however, ours is the first to provide support for multithreaded programs. In addition our system provides this functionality in the form of a user-level library which makes it easier to use and the design is still efficient.

Several other user-level checkpointing libraries for single processes run on multiple versions of Unix [12, 14, 16]. `libckpt` has many features including asynchronous (forked) checkpointing, incremental checkpointing, memory exclusion and user-directed checkpointing [12]. It has been ported to many different versions of Unix. However, `libckpt` does not handle multithreaded processes or dynamically linked executables.

Condor is a process migration system designed to use idle cycles in the network [14]. When the system decides to migrate a process it checkpoints the process on one machine then restarts it on another. Condor runs on a number of operating systems including Solaris and Linux. It neither supports multithreaded programs nor does it have freely available source code.

`libckp` was developed at AT&T Bell Laboratories to checkpoint Unix processes [16]. In contrast with `libckpt`, Condor and our own checkpointing library, `libckp` saves files along with the checkpoint to guarantee they will be the same when the program recovers from a checkpoint. Saving copies of all open files guarantees all the files will exist during recovery and allows `libckp` to handle arbitrary file I/O access patterns, but it can make the checkpoint much bigger. Many scientific programs do not need the extra guarantees if the user is willing to retain the input and output files and the application only writes to files in sequential order. `libckp` also does not support multithreaded programs.

Process hijacking uses dynamic executable rewriting to add checkpointing to programs that were not compiled with checkpointing support [19]. Process hijacking does not support multithreaded processes.

MOSIX and `epckpt` provide kernel-level checkpointing solutions. MOSIX is a set of kernel extensions which have been ported to BSD and Linux [4, 3]. MOSIX uses a kernel module to provide transparent load balancing and process migration. `epckpt` is a Linux kernel patch that adds support for processes and process groups [1]. It is in an early stage of development and requires patching, recompiling, and installing a new kernel. Neither MOSIX nor `epckpt` work for multithreaded programs.

Process migration in general [11, 18] is related to checkpointing. Several process migration facilities, like Condor and MOSIX, use checkpointing to provide process migration. In the case of process migration a process is transported through space to another machine. In checkpointing the process is transported to a later time on the same or a different machine. The difference is that a process may recover from a checkpoint at a later time when the environment has changed. Resources the original process was using may be unavailable when it recovers.

Checkpointing for distributed message passing systems has been heavily studied [8]. Most message passing algorithms work to reduce synchronization overhead and handle in transit messages, which are not an issue for multithreaded processes.

The LinuxHA project [2] is bringing support for high availability to Linux. LinuxHA's failure detection mechanisms could be used with our library to automatically restart programs. Most of the LinuxHA work is focused on replicating processes on different machines for fault-tolerance. Replication can offer better guaranteed bounds on recovery time, but usually requires a duplicate machines to take over for each replicated process when a machine fails. Checkpointing only needs extra machines when a machine fails, and then only enough to replace the failed machines. The program can wait until

the the failed machines are repaired if no machines are available and the application can tolerate the delay.

The IEEE Portable Application Services Committee (PASC) 1003.1m Checkpointing Restart working group has been developing a standard API for checkpointing [5].

3 Features

The checkpointing library we introduce here allows, for the first time, LinuxThreads programs to automatically be checkpointed. In addition to checkpointing multithreaded programs our checkpointing library provides features that help meet our goals of being simple to use, flexible, and efficient.

Adding checkpointing support to a C program is straightforward with our checkpointing library. The application programmer only needs to add one line to include the checkpoint header file:

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

and one line to call checkpoint initialization in `main`.

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

`checkpoint_init` initializes data structures the checkpointing library uses to track thread and file state. Passing `argc` and `argv` to `checkpoint_init` allows the checkpointing library to read options from the command line. 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
```

runs the *prog* program with a checkpoint period of *period* seconds. A checkpoint period of 0 disables checkpointing. The user can also pass options to the checkpointing library by putting the options in the `CHKPTOPTS` environment variable. The programmer can set checkpointing options directly using third argument to `checkpoint_init`.

Checkpoints are automatically stored in *prog*`.chkpt.`*n* where *prog* is the name of the program and *n* is the checkpoint number. The user can change the default checkpoint base name with the `-b` option.

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*.

An application program can install callback functions to save any state not saved by the checkpointing library. For example, we have used callback functions to help add checkpointing to the Unify distributed shared memory system [9]. Unify processes communicate through UDP sockets, but the checkpointing library does not save their state. To make checkpointing work Unify makes sure checkpoints are consistent and uses a recovery callback function to reopen the UDP sockets when it recovers from a failure.

A process can install callback functions that will be called before a checkpoint, after a checkpoint, and after recovering from a checkpoint. `chkpt_callback_push` installs three functions: a *pre-checkpoint* callback called before each checkpoint, but after all application threads have been stopped, a *post-checkpoint* callback called after the checkpoint, but before any application thread has been restarted, and a *post-recovery* callback called after recovering from a checkpoint.

Pushing a new set of callback functions does not remove any of the old ones. Instead they are pushed onto a stack. The most recently pushed pre-checkpoint callback function is called last. The most recently pushed post-checkpoint and post-recovery callback functions are called first. The program can remove callback functions in any order using the ID returned by `chkpt_callback_push`. The pushing and popping mechanism simplifies installing and removing callbacks to handle different kinds of state as a program enters different phases.

Our checkpointing library provides memory exclusion similar to that provided by `libckpt` [12]. Memory exclusion allows the application to specify regions of memory that need not be saved in the checkpoint. Excluding large areas of memory that the application does not need reduces the size of the checkpoint.

4 Implementation

The difficulty of checkpointing multithreaded programs comes from making sure that the thread library is in a useful state after recovering from a checkpoint. Threads must be carefully restarted in the correct order to match the way they were originally created.

The basic idea behind our checkpointing library is simple. During initialization the *main thread*, the only thread that exists when the program starts, starts the *checkpoint thread*. After initializing itself, the checkpoint thread blocks with a timed wait on a condition variable. When the timer expires or when another thread calls `checkpoint_now` the checkpoint thread starts a checkpoint. The checkpoint thread is also responsible for running application callback functions.

To take a checkpoint, the checkpointing library blocks all threads, except the main thread, to prevent any threads from changing the process's state while it is being saved. The main thread then saves the process's state and unblocks all the remaining threads.

To recover from a checkpoint, the checkpointing library restarts the threads that were running at the time of the checkpoint. The restarted threads block while the main thread loads the process's state from a checkpoint. Then the main thread unblocks the other threads and they continue running from the checkpoint. Section 4.1 and Section 4.2 describe the algorithm in more detail.

The difficulty comes from doing everything in the correct order, making sure threads do not try to change the address space while it is being saved, and making sure the process's idea of its state matches the operating system's idea of the state. For example the thread library keeps track of the process IDs of all the threads. The checkpointing library must be able to update the thread library's copies of the thread process IDs.

The process state saved in a checkpoint includes the address space, thread registers, thread library state, signal handlers, and open file descriptors. The checkpointing library cannot save every part of the program's state. The unsaved parts lead to the restrictions described in Section 5.

A process's address space is made up of several segments. These segments include the code segment, data segment, heap, stack segment, code and data segments for each of the shared libraries linked with the program, and thread stacks. The checkpointing library uses the `/proc`(4) file system interface to find the segments that are mapped into memory.

4.1 Saving a Checkpoint

Figure 1 shows how the checkpointing library takes a checkpoint.

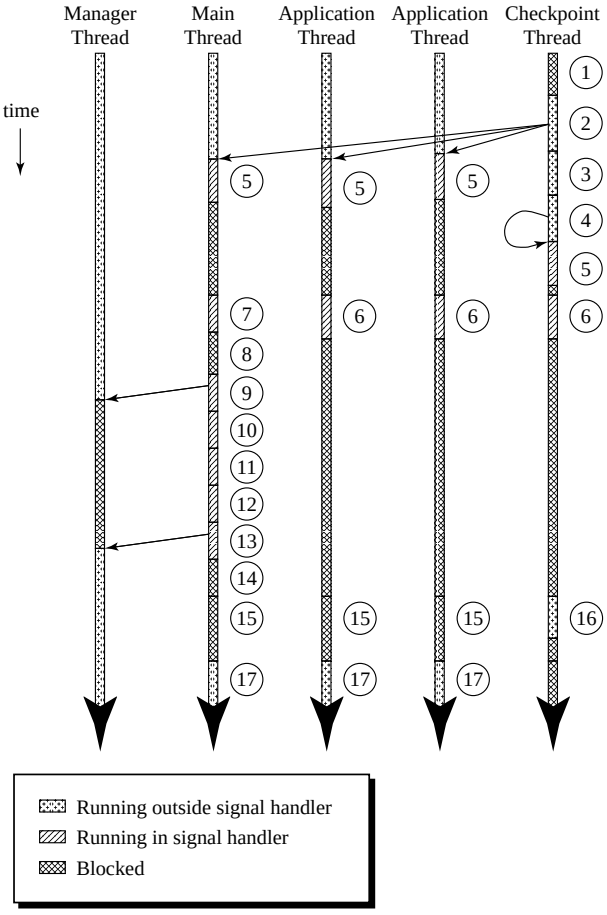

Figure 1: Each thread coordinates with the others to save the process's state. This figure shows how the threads interact.

1. **Checkpoint thread unblocks.** The checkpoint starts when either the checkpoint thread's timed wait expires or an application thread calls `checkpoint_now`.

2. **Send a signal to application threads.** To start a checkpoint, the checkpoint thread sends a signal to each of the application threads. Unlike Solaris, when a thread in Linux receives a signal it enters the signal handler for the signal regardless of

the state of the mutex associated with the condition variable [7].

3. **Call pre-checkpoint callbacks.** The checkpoint thread calls the pre-checkpoint callbacks.

4. **Send a signal to the checkpoint thread.** For symmetry the checkpoint thread sends a signal to itself to force itself into its signal handler like all the other threads.

5. **Block signals and wait.** Once in the signal handler every thread blocks all signals and waits at a barrier for the rest of the threads to enter the signal handler.

6. **Save private thread state.** When all threads have entered the signal handler, each thread, except the main thread (and the manager thread), saves its context to memory by calling `sigsetjmp`(3). Each thread, except the main thread, then blocks at another barrier.

7. **Save signal handlers.** The main thread saves the process's signal handlers using `sigaction`(2).

8. **Wait for other threads.** The main thread waits until all the other threads have called `sigsetjmp`(3) and reached the barrier.

9. **Stop the manger thread.** The checkpoint thread cannot send a signal to the manager thread when it is signalling all the other threads in step 2 because the manager thread has no thread ID. Instead the main thread sends a message to the pipe the manager thread normally uses to communicate with other threads. When the manager thread receives the message it blocks until the main thread unblocks it.

10. **Save main thread stack environment.**The main thread calls `sigsetjmp`(3) to save its stack environment.

11. **Save file state** Once the other threads have reached the barrier the main thread saves the current file pointer for all open regular files.

12. **Save address space.** The main thread saves the entire address space to the checkpoint file.

13. **Unblock Manager Thread.** The main thread unblocks the manager thread.

14. **Wait at barrier.** The main thread waits at the same barrier as the other threads causing all threads to continue.

15. **Wait at barrier.** After leaving the barrier all threads except the checkpoint thread and manager thread wait at another barrier.

16. **Run post-checkpoint callbacks.** The checkpoint thread runs all registered post-checkpoint callback functions while the rest of the threads wait at the barrier.

17. **Resume execution.** Finally, the checkpoint thread joins the barrier and all the threads leave the barrier, restore their signal mask, and return from the signal handler.

4.2 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 as shown in Figure 2.

1. **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. The thread library automatically creates the manager thread when the checkpointing library creates the first thread.

2. **Restore thread stacks.** The main thread waits while the child threads restore their stack pointers. Each thread restores its stack by calling `siglongjmp`(3) which causes the thread to return from the `sigsetjmp`(3) 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.

3. **Wait for the main thread.** The child threads wait at a barrier for the main thread to finish restoring the program's state.

4. **Get thread ID to process ID mapping.** After starting all the threads, the main thread calls `pthread_chkpt_restart` to get the new thread ID to process ID mapping. The main thread copies the mapping into an area of memory that will not be overwritten when the main thread restores the address space. `pthread_chkpt_restart` also sends a message to the manager thread telling it to call `siglongjmp`(3) and block so it will be prepared for its stack to be restored.

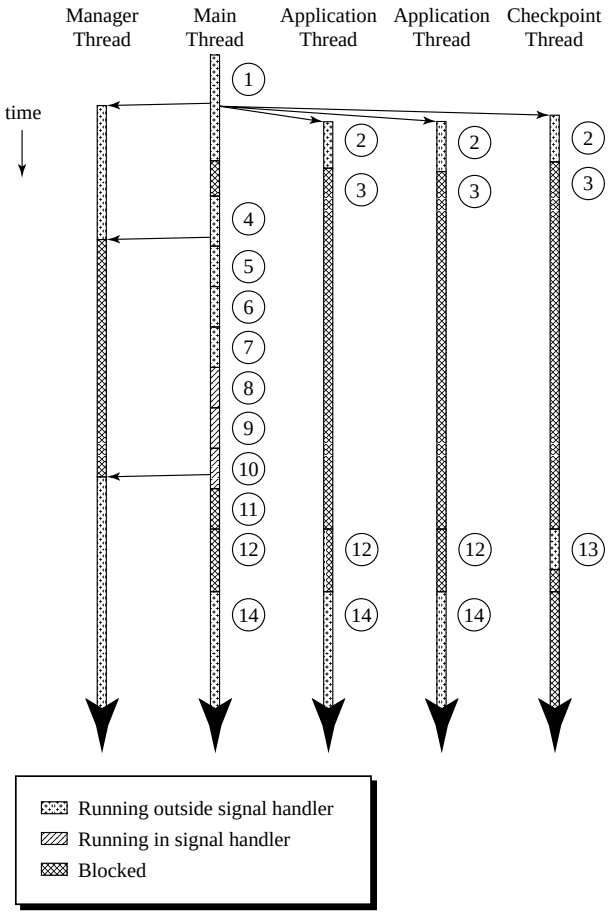

Figure 2: Each thread coordinates with the others to save the process's state. This figure shows how the threads interact.

5. **Load main thread stack.** Once all of the child threads and the manager thread 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 its 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.

6. **Remap the process's address space** The main thread then maps every segment except the main thread stack into the program's address space from the checkpoint file using `mmap`(2) similar to the method Condor uses [10]. The checkpointing library uses `mmap`(2) to remap segments because `mmap`(2) does not cause the data to be loaded immediately. The operating system demand loads the contents of the segments when the program accesses them.

7. **Restore signal handlers.** The main thread restores the signal handlers with `sigaction`(2).

8. **Restore main thread stack pointer** The main thread calls `siglongjmp`(3) to continue execution where the program was when it saved the checkpoint.

9. **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.

10. **Restore thread ID to process ID mapping.** Next the main thread restores the thread ID to process ID mapping and unblocks the manager thread by calling `pthread_chkpt_postrestart`.

11. **Wait at barrier.** The main thread waits at the same barrier as the other threads causing all threads to continue.

12. **Wait at barrier.** After leaving the barrier all threads except the checkpoint thread wait at another barrier.

13. **Run post-checkpoint callbacks.** The checkpoint thread runs all registered post-checkpoint callback functions while the rest of the threads wait at the barrier.

14. **Resume execution.** Finally, the checkpoint thread joins the barrier and all the threads leave the barrier, restore their signal mask, and return from the signal handler.

4.3 Intercepting Library Functions

The checkpointing library takes advantage of dynamic linking to intercept some library function calls and system calls so it can track the program's state. The checkpointing library intercepts library functions by providing an *intercepting function* with the same name as the library function. The checkpointing library calls `dlsym`(3) with the `RTLD_NEXT` option during initialization to get the addresses of all the intercepted library functions so the intercepting function can call the system version of the function. This method works for system calls as well as library functions because the code to setup and make system calls is part of the C library.

For example, when the application calls `pthread_create`(3), it gets the checkpointing library's version. The checkpointing library records the parameters passed to `pthread_create`(3) so it can use them during recovery. Then it calls the system `pthread_create`(3) using the address it got from `dlsym`(3) during initialization. If the `pthread_create`(3) call is successful, the checkpointing library updates the number of running threads and returns. Otherwise, it cleans up its thread table and passes the error on to the application program.

4.4 Handling Open File Descriptors

The checkpointing library uses an array that mirrors the kernel's file descriptor table to save the state of open files in each checkpoint. To re-open the files during recovery the checkpointing library needs the filename, mode, and current offset into each open file. When the process opens a file with `open`(2) the checkpointing library adds an entry in its table for that file descriptor with the filename and mode. If the process calls `dup`(2) or `dup2`(2) the checkpointing library links the new file descriptor information to the old file descriptor information. When the process closes the file descriptor its entry is removed from the checkpointing library's file descriptor table. The `read`(2) and `write`(2) system calls are not intercepted.

When the process takes a checkpoint the checkpointing library saves the current file pointer of every regular file. When the process recovers from a checkpoint it uses the information in the checkpointing library's file descriptor table to re-open files and seek the file pointer to the position at the time of the checkpoint.

The checkpointing library intercepts the `popen`(2) call to keep track of pipes that are open. During recovery the checkpointing library reopens pipes to replace those that existed at the time of the checkpoint. However, the checkpointing library does not keep track of data read from or written to the pipe, so data buffered in the kernel may be lost. It also does not handle processes outside the main process. The pipe support is only useful if two threads in the same process share a pipe.

4.5 Linux Specific Issues

Implementing checkpointing for LinuxThreads programs is simpler than for Solaris because LinuxThreads is simpler than the Solaris pthread library. The Solaris kernel treats threads, lightweight processes (LWPs), and processes as different entities. Handling the interactions between threads and LWPs in Solaris is complex. In addition Solaris adds some rules about when a process can handle a signal that complicate the checkpointing library [6].

The Linux kernel is less complex than Solaris because the Linux kernel does not distinguish between threads and processes. LinuxThreads creates threads with `clone`(2) a generalized version of `fork`(2). Like `fork`(2), `clone`(2) creates a new process, but `clone`(2) allows the caller to specify which resources the new process shares with its parent and which resources the new process copies from its parent. Thus threads in a LinuxThreads program are separate processes that happen to share an address space and file descriptors with all other threads.

We had to modify the thread LinuxThreads library to handle two different problems. First, the thread library stores a mapping from thread IDs to process IDs in its data segment. When the checkpointing library reloads the process's address space from a checkpoint, the thread ID to process ID mapping is restored to the mapping at the time of the checkpoint, which is out of date for the restored process. To handle this problem, the checkpointing library saves the thread ID to process ID mapping in memory that will not be reloaded from the checkpoint before it restores the address space, but after it restarts threads. The checkpointing library corrects the thread ID to process ID mapping after it restores the process's address space.

Second, the thread library uses a *manager thread* to create processes. When a thread creates a new thread it sends a message to the manager thread through a

pipe and the manager thread creates the new thread. The checkpointing library coordinates with the manager thread during checkpoints to save the manager thread's private state.

The checkpointing library adds four functions to the thread library to handle its interactions with the thread library. The checkpointing library calls `pthread_chkpt_precreate` before it saves the process's address space. `pthread_chkpt_precreate` sends the manager thread a message telling it a checkpoint is beginning. The manager thread saves its environment by calling `sigsetjmp`(3) and blocks. The checkpointing library unblocks the manager thread by calling `pthread_chkpt_postcreate` after saving the checkpoint.

When restoring a process from a checkpoint, the checkpointing library calls `pthread_chkpt_prerestart` to get a copy of the thread ID to process ID mapping and to send a message to the manager thread telling it to call `siglongjmp`(3) and block. The thread library saves the thread ID to process ID mapping in memory that will not be overwritten when the address space is restored. After restoring the address space, the checkpointing library calls `pthread_chkpt_postrestart` to restore the thread ID to process ID mapping and unblocks the manager thread. The `pthread_chkpt_` calls are the only added entry points to the thread library.

5 Restrictions

Our checkpointing library supports programs that access regular files sequentially or use signal handlers for signals. At least one signal must be available for the checkpointing library. The checkpointing library cannot restore process IDs and it does not support programs that randomly access files or communicate with other processes. In most cases, however, the application programmer can add recovery code in callback functions to recover the file or communication state. For example, we are using the checkpointing library to add checkpointing to the Unify distributed shared memory system [9].

Random access reads do not present a problem as long as the program never writes to the file. General random access files are difficult to handle because the checkpointing library must be able to roll the file back during recovery to the state it was in during the checkpoint. One simple way to do this is to save the entire file with the checkpoint [16]. Saving could increase the checkpoint size a lot if the program uses a lot of large files. The checkpointing library could avoid some of the overhead by not saving files opened with mode `O_RDONLY`. Assuming the files do not change between when they are opened and when the program finishes using them.

The other alternative for handling random access files would be to log each change made to the file. During recovery the checkpointing library could undo all changes made since the last checkpoint. The disadvantage of logging changes is that it adds overhead to log every write operation and the log grows with each write between checkpoints. We did not want to add this overhead when our applications want to use sequential access files. We could reduce overhead by only logging files that are open with mode `O_RDWR`. In our current implementation, an application writer that wants to save random access files with a checkpoint could write a callback routine to manually save the desired files during a checkpoint.

POSIX threads (and LinuxThreads) do not provide a way to create a thread with a particular thread ID. The checkpointing library assumes that the thread library always assigns thread IDs in the same order. As long as that assumption is true, the checkpointing library can guarantee each thread has the same thread ID after recovering by creating threads in the order in which they were originally created. Currently our checkpointing library does not handle programs with threads that exit before the end of the program. If a thread exits early, the thread created immediately after the thread that exited early will get the exited thread's ID during recovery. This problem could be fixed during recovery by creating a thread that exits immediately in place of the exited thread to use up the exited thread's ID. Alternatively we could modify the thread library to allow programs to request particular thread IDs, but we wanted to minimize the changes to the thread library to make it easier to work with different versions of the thread library. The applications we work with create all the threads they need at the start and the threads keep running until the program exits so it was not a problem for our applications.

During recovery, described in section 4.2, the checkpointing library restarts all the threads and restores the process's entire address space from a 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 is not entirely true in Linux and thus the thread library must be modified as described in section 4.5.

The checkpointing library interrupts each thread with a

signal to start a checkpoint. When the checkpointing library installs its signal handler, it passes SA_RESTART flag to `sigaction`(2) to tell the Linux kernel to restart interrupted system calls if possible. The application code must restart system calls that the Linux kernel cannot restart.

We added nothing extra to support thread cancellation functions. The problem with thread cancellation is recreating the threads with the correct thread IDs as described above. Otherwise the checkpointing library would just need minor adjustments to cleanup the canceled thread after its last cancellation handler is called.

We also did nothing special to support thread scheduling priorities. Handling thread scheduling priorities would be a matter of logging the calls to the thread scheduling priority calls and reissuing them to restore scheduling priorities during recovery. Support may be added if there is demand.

The checkpointing library does not handle interprocess communication, but it does reopen pipes open at the time of a checkpoint. It cannot make another process use either end of the pipe it opens, and it does not save any data buffered in the kernel. Thus pipes will only be restored if they are used between threads in the same process and no data is buffered in the pipe at the time of the checkpoint.

Handling IPC, either between processes on the same machine or on different machines, is difficult in general. Both processes must agree on when they take checkpoints or make assumptions about how deterministic they are to avoid inconsistent checkpoints. Otherwise one process could fail and recover from a checkpoint that rolls it back to a state in which it has not sent a message on which another process depends. At that point the program is in a state which it could not have reached without a failure. The second process is in a state the causally depends on a state that never happened, as far as the first process is concerned. In that case the two processes are said to be *inconsistent*. Issues of consistency have been well studied and are beyond the scope of this paper [8]. Extending our checkpointing library to work for a general case with multiple processes communicating with pipes or TCP sockets would be non-trivial.

The checkpointing library does not intercept or modify time related system calls in any way. A program that uses absolute time values may behave strangely after recovering from a checkpoint. For example, assume a thread blocks using `pthread_cond_timedwait`(3) to wait for several seconds and the process checkpoints while the thread is blocked. The process is killed and restarts from the checkpoint several minutes later. After recovery, the thread will immediately unblock because the timer has expired.

Strictly speaking this behavior is correct because the current time is later than the time for which the thread was waiting. Applications that have time based events, however, might get a flood of expired timers when recovering from a checkpoint. We could intercept all calls that have anything to do with absolute time and adjust the time they see after recovering from a checkpoint, but some programs need to know what the absolute time really is. Instead we leave it up to the application programmer to write a callback function to adjust any time values that need to be adjusted after recovering from a checkpoint.

6 Results

The checkpointing library adds overhead due to intercepting calls, overhead due to the checkpointing thread, and overhead due to saving checkpoints. The checkpointing library only intercepts thread creation, file open, and file close calls. Unless the program opens and closes files often or creates and destroys threads often that overhead will be low. The checkpointing thread does not add much overhead because it is blocked except during checkpointing.

6.1 Applications

To verify that checkpointing adds little overhead when it is not writing a checkpoint, we ran several applications from the SPLASH2 [17] benchmark suite. SPLASH2 is a set of benchmarks designed to test the performance parallel shared memory machines. The benchmarks are based on applications and kernels commonly used in scientific computing. We present the results of running the BARNES and WATER-SPATIAL benchmarks below. The other SPLASH2 benchmarks we ran gave similar results.

BARNES simulates the gravitational effects of a number of bodies in space using the Barnes-Hut algorithm. It uses a tree to represent the locations of the bodies in space. WATER-SPATIAL simulates the the interaction of water particles using a 3 dimensional grid. We increased the problem sizes of both applications from the size used in the original SPLASH2 paper [17] to in-

crease the running time of the benchmarks. For WATER-SPATIAL we increased the number of particles to 8000 instead of 512. For BARNES we used 65536 particles instead of 16384.

The test machine was a two processor 500 MHz Pentium III SMP machine with 512 KB L2 cache and 128 MB of main memory running Linux kernel version 2.2.10 with NFS v2 and glibc2. The file systems mounted for the NFS tests were served by an UltraSPARC-10 running Solaris 5.7. The network over which the NFS disk was mounted was a moderately used 100 Mb/s switched Ethernet connected by a Cisco Catalyst 2924 XL auto-sensing 10/100 Mb/s switch. None of the machines used in the test shared a port on the switch with any other machine.

Table 1 compares running each of the benchmarks with and without checkpointing linked for one and two processors. With checkpointing linked, the program called the checkpoint initialization code, but did not take any checkpoints. This test shows how much overhead checkpointing adds not including the time to save the checkpoint to disk. Table 1 shows that the checkpointing library does not add much overhead when the program is not checkpointing. This overhead is important because the program will spend most of its time running, not checkpointing.

In some cases, code with checkpointing linked but not used runs faster than without checkpointing. Changes in the program's memory layout cause this speedup. Linking the benchmark with unused code gave the same effect as linking with the checkpointing library.

Table 2 shows the amount of time BARNES and WATER-SPATIAL spend taking a checkpoint using a local disk or an NFS mounted disk. Most of the time is spent saving the checkpoint to a file. These numbers are intended to give a feel for how long it takes for application programs to checkpoint and how the checkpointing time is spent. In both the local disk and the NFS disk cases the amount of time spent synchronizing threads before and after the checkpoint is several orders of magnitude smaller than the amount of time to save the checkpoint to disk. Saving the checkpoint to disk is the largest overhead of checkpointing.

6.2 Checkpoint Size

The size of the checkpoint file is directly proportional to the size of the process's address space. Most of the overhead of taking a checkpoint comes from writing the address space to disk. Figure 3 gives an idea how long a program will take to save a checkpoint depending the size of the checkpoint and whether it is saving to a local disk or an NFS mounted disk.

The amount of time required to save the checkpoint depends on the file system to which it is saved. The figure shows times for local disk and NFS. Writing a checkpoint to an NFS mounted disk[1] takes much longer than writing to a local disk. The figure also shows that the amount of time required to save a checkpoint is directly proportional to the size of the checkpoint.

In this case, the time required to save a checkpoint to the NFS mounted disk can be approximated by the equation $t_{cp} = 2.37\text{sec}/\text{MB} \times s - 0.60\text{sec}$ for checkpoints larger than about 1.3 MB, where s is the size of checkpoint. The user can use t_{cp} to decide which checkpoint interval to use. A simple method is to calculate the maximum percentage of execution time taken by checkpointing given t_{cp} and the maximum address space size (checkpoint size) of the program. More sophisticated methods can determine the checkpoint interval that will minimize the program's expected run time given t_{cp} and a particular failure rate [13, 15].

7 Conclusion

Our checkpointing library provides checkpointing multithreaded Linux programs. It adds little overhead except when taking a checkpoint. The overhead that it does add is directly proportional to the size of the address space. Saving the checkpoint to local disk is much faster than saving it to an NFS mounted disk.

Our checkpointing library combines simplicity for many programs but flexibility for programs willing to use it. Callback function provide flexibility for applications that have special needs. Features like memory exclusion can and user directed checkpointing can reduce overhead when taking a checkpoint. We are considering adding incremental and asynchronous checkpointing to further reduce the overhead of saving a checkpoint.

The latest version of the checkpointing library is available through our web site at:

http://www.dcs.uky.edu/~chkpt

[1] The disk with mounted with rsize=8192,wsize=8192.

	1 Processor			*2 Processors*		
Benchmark	Not Linked (seconds)	Linked (seconds)	Overhead (percent)	Not Linked (seconds)	Linked (seconds)	Overhead (percent)
BARNES	53	53	0	32	31	-3.1
WATER-SPATIAL	666	678	1.8	335	340	1.5

Table 1: This table gives execution times of several SPLASH2 applications. For the "Not Linked" case, the application was run without the checkpointing library linked to it. In the "Linked" case, the checkpointing library was linked to the application, but it did not take any checkpoints. This case measures checkpointing overhead outside not directly related to taking checkpoints. All times are in seconds.

	Local Disk			*NFS*		
Benchmark	Total Time (seconds)	Save Time (seconds)	Sync Time (seconds)	Total Time (seconds)	Save Time (seconds)	Sync Time (seconds)
BARNES	9.8351	9.8330	0.0021	337.1552	337.1536	0.0017
WATER-SPATIAL	0.0736	0.0716	0.0020	16.3965	16.2361	0.1604

Table 2: This table lists the time required to save checkpoints for several SPLASH2 applications. "Total Time" is the total amount of time to take a checkpoint. "Save Time" is the amount of time required to save the checkpoint to a file. "Sync Time" is the amount of time required for processes to synchronize before and after the checkpoint. All times are in seconds with 2 Processors.

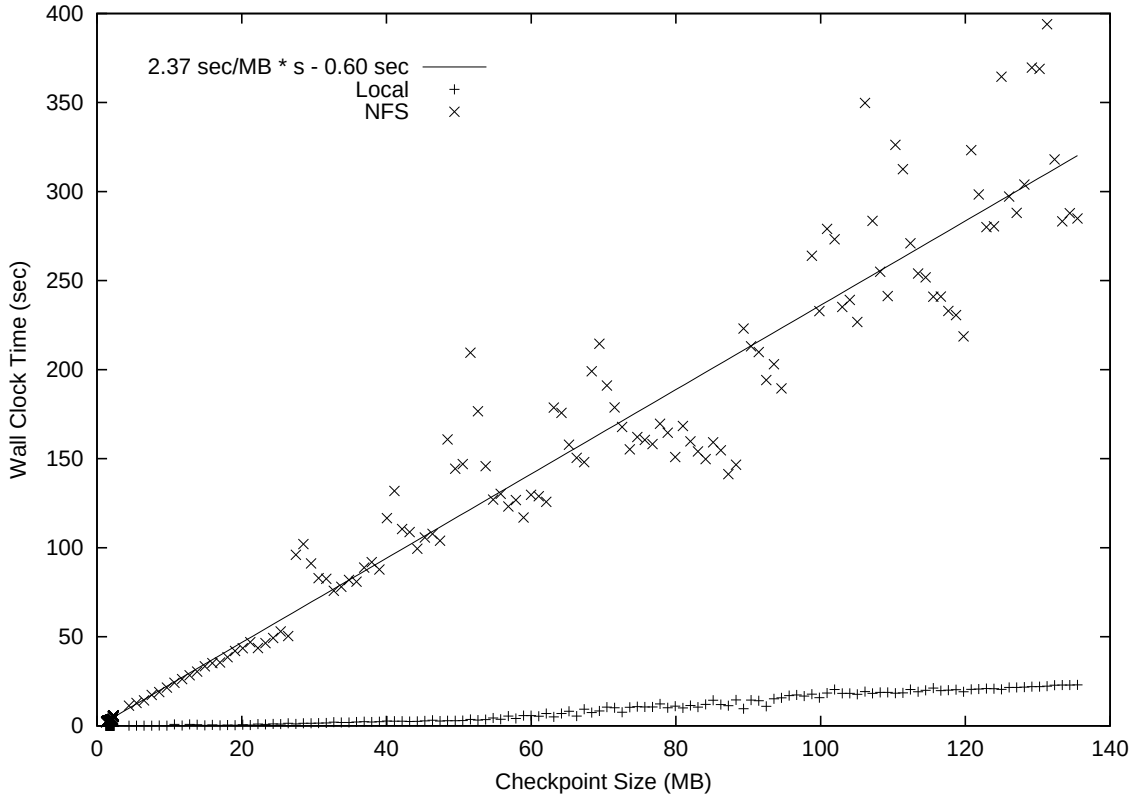

Figure 3: This graph shows a plot of the time to save a checkpoint as a function of checkpoint size. The line through the NFS data points was fitted using the method of least squares.

References

[1] EPCKPT: Eduardo Pinheiro Checkpoint Project Website. Technical report, http://www.cs.rochester.edu/u/edpin/epckpt/.

[2] High-Availability Linux Project Website. Technical report, http://www.linux-ha.org/.

[3] Amnon Barak, Oren La-adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for LINUX. In *Proceedings of Linux Expo '99*, pages 95–100, May 1999.

[4] Amnon Barak and Aren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[5] Portable Application Standards Committee. *Draft Standard for Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) – Amendment m: Checkpoint/Restart Interface*. IEEE Computer Society, p1003.1m/d2 edition.

[6] William R. Dieter and James E. Lumpp, Jr. A user-level checkpointing library for POSIX threads programs. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 224–227, June 1999.

[7] William R. Dieter and James E. Lumpp, Jr. User-level checkpointing of posix threads. Technical Report CEG-99-004, University of Kentucky, Department of Electrical Engineering, Lexington, KY 40506–0046, http://www.dcs.uky.edu/˜chkpt, 1999.

[8] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and Davaid B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, http://www.cs.utexas.edu/users/lorenzo/papers/Pap6.ps, June 1999.

[9] James Griffioen, Rajendra Yavatkar, and Raphael 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.

[10] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Science, April 1997.

[11] Dejan S. Milojičič, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration survey. *ACM Computing Surveys*, pages 241–299, September 2000.

[12] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, January 1995.

[13] James S. Plank and Wael R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 48–57, June 1998.

[14] Todd Tannenbaum and Michael Litzkow. Checkpointing and migration of unix processes in the condor distributed processing system. *Dr. Dobbs Journal*, pages 40–48,102, February 1995.

[15] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, pages 942–947, August 1997.

[16] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 22–31, June 1995.

[17] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, June 1995.

[18] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 Unix for massively parallel multicomputers. In *USENIX Winter 1993 Technical Conference*, pages 449–468, January 1993.

[19] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, pages 177–184, August 1999.