Linux Support for Transparent Checkpointing of Multithreaded Programs

Christopher D. Carothers and Boleslaw K. Szymanski
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, U.S.A.
{chrisc,szymansk}@cs.rpi.edu

February 20, 2002

1 Introduction

The most common use of checkpointing is in fault tolerant computing where the goal is to minimize loss of CPU cycles when a long executing program crashes before completion. By checkpointing a program's state at regular intervals, the amount of lost computation is limited to the interval from the last checkpoint to the time of crash. Research in this class of checkpoint algorithms and systems has been ongoing for at least last 15 years.

Our interest here is on the fast, efficient checkpointing of threaded programs that execute on shared-memory computing platforms. We are motivated by problems that arose in our investigation of new parallel simulation and computation synchronization methodologies. There are two paradigms in which the ability to checkpoint (save the state of the computation) quickly is crucial. One is the speculative execution of a portion of code that otherwise would be suspended by synchronization. For example, consider a program reading an object mirrored on the local site. If this object changes infrequently, then instead of waiting verify the validity of the local copy, the program can checkpoint and then speculatively read the object. If the local copy is invalid, the executing copy of the program can be killed, and the copy with pre-reading state executed. The amount of time saved by not waiting to verify the validity of a local object copy defines the gain of the speculative execution.

In general, let p be a probability that the speculation is unsuccessful and would require rolling back the computation to the speculation point. Let c_r denote the cost of such a rollback and c_s the saving resulting from elimination of waiting for synchronization when speculation is successful. Finally, let s_o be the cost of speculation, (mainly the cost of checkpointing, incurred regardless of the outcome of the speculation). Under this assumptions, speculation is beneficial when:

$$c_s > c_o + c_r * p \text{ or, equivalently } p < (c_s - c_o)/c_r.$$

Hence, smaller the value of c_o, cost of checkpointing, more widely speculative execution can be applied beneficially.

The primary contribution of this paper is a new algorithm for fast, efficient checkpointing of large-scale shared memory multithreaded programs. This approach leverages the existing *copy-on-write* semantics of the virtual memory system by introducing a new `checkpoint` system call into the Linux operating system. While based on Linux, we believe that the core idea and algorithmic approach is general enough that it could be put into any operating system.

2 Linux Internals

The Linux Operating System is one of many variants of UNIX. Like most versions of UNIX, Linux supports virtual memory, process creation and control, interrupts, symmetric multiprocessing, interprocessor communications, systems management for many types of files, communications (i.e., sockets) and a host multimedia peripheral devices such as sound and video. As of this writing, the current most stable version of Linux is 2.4.10. Our algorithm is based on is 2.4.8 Linux version, however, it is a nontrivial task to port our implementation to more recent versions within the stable 2.4 source tree.

2.1 LinuxThreads

Starting at the user-level, multithreaded programs are typically realized on Linux using a threading package such as LinuxThreads or Next Generation POSIX Threading (IBM).

Developed by Xavier Leroy, LinuxThreads is an implementation of POSIX 1003.1c Pthread interface. This implementation provides the appearance of kernel level threads by realizing each thread as a separate UNIX process which shares the same address space with all other threads. Scheduling between threads is handled by the kernel scheduler, just like scheduling between UNIX processes.

One of the drawbacks of LinuxThreads is that each thread is realized as a full kernel process. This prevents what is called N:M threading models where many threads can be bound to a particular kernel-level process or thread. Currently, SGI, SUN and IBM UNIX variants support this threading model. Other problems with LinuxThreads include different process identifiers for each thread and the use of user defined signals which prevents programs that need both threading and user defined signals from operating cleanly.

LinuxThreads are created by using the `clone` system call. This Linux specific system call allows processes to be created in such a way that they can share resources at a variety of different levels. In particular, a process and its child can be configured to share (or not to share) virtual memory, file system information, file descriptors, and signal handlers.

In LinuxThreads, when the thread is created, a thread manager process is instantiated which then spawns the new thread using the `clone` system call. This manager thread then waits for other thread create requests as well as performs other thread management functions.

It should be noted that `clone` system call can be used to checkpoint threaded programs, but it would require significant modifications to the LinuxThreads library. In particular, the Pthread manager would have to be modified to clone itself where none of its previous resources are shared by threads.This newly cloned thread manager would then create new threads that share resources with the cloned thread manager. The cloned thread group and their respective parent threads would then have to coordinate the transfer of thread specific state across two address spaces, such as thread stacks. The stacks could be reproduced by having the parent threads call `setjmp` to save the stack context and the child threads would call `longjump` using the stack context pointer set by the parent threads call to `setjmp`. Because thread stacks are realized in the heap space of the thread manager, stack copying could be avoided, however there is some performance penalty associated with these additional system calls and thread synchronization.

An additional disadvantage of using `clone` for checkpointing is that implementation would be tied to LinuxThreads. As we have pointed out, there are other thread packages available under Linux. Moreover, LinuxThreads is mated to the GNU C Library `glibc`. It is well known fact in the Linux community that upgrading or modifying the local version of `glibc` is difficult and must be done with extreme care. The problem is that such modifications risk breaking every binary in the system because of the use of shared libraries (i.e., the new shared library is no longer compatible with the version your binaries were linked against). Since an OS checkpoint system call is not tied to a specific thread packages it is ultimately easier to implement and support.

2.2 System Calls and Process Creation

On the Intel (x86) port of Linux, system calls are realized by using software interrupts. In particular, interrupt 0x80 is used. Internal to the OS is a jump table of system calls which relates their numbers to the specific code address where that system call begins.

Because the invocation of a system call is architecture specific, all top-level system call handler routines are in the `asm` code directory. As a matter of convention, all system call handlers have the `sys_` prefix. For example, the `fork` system call handler is `sys_fork`. These system calls then typically invoke a more general handler routine that is not architecture specific. The prefix for those handlers is `do_`. In the case of process creation, the general handler routine for all of types of processes creation (i.e., fork, vfork or clone) is `do_fork`.

As a part of the design of a system call, the kernel always provides access to the calling process' control block or task structure by invoking a macro called `current`, as well as CPU register state which is passed as an argument. This macro returns a pointer to the task structure that invoked this system call. Beyond system calls, `current` is the process that has control of CPU. In the case of multiple CPUs, each CPU is running a different process and thus `current` will be different across CPUs.

The structural layout of the process control block includes variables to record the scheduling priority and policy, memory management, current processor, list pointers used to place a process in a wait queue or run queue, signal handler state, file system information, interprocess communication information, and process specific statistics such as cpu usage, etc. This structure is called a `task_struct` in the Linux Operating System.

Within the `task_struct`, process specific memory management data is encapsulated into its own structure, called `mm_struct`. This data structure contains a mapped address space of the process. Thus, by switching a process from one `mm_struct` to another, its execution address space is changed. We use this feature to cleanly implement our new system call. For the interested reader, these structures are defined in `/usr/src/linux/include/linux/sched.h`.

3 Checkpoint Algorithm

3.1 Overview

As previously indicated, our algorithmic approach leverages the existing *copy-on-write* capability of virtual memory by introducing a new `checkpoint` system call. This new system call is very similar to the `fork` and `clone` system calls. The primary difference is that `checkpoint` considers all processes that are part of a multithreaded program.

The algorithm works by creating a *rendezvous* of all threads inside the kernel. By using a rendezvous approach, the system call guarantees that the checkpoint is made consistent. That is no copy of the address is made until all threads have entered the system call and ceased all user-level execution.

Once all threads of a program are inside the system call, the thread with the smallest process identifier is made the "parent" or master thread. The parent thread then creates a new duplicate `mm_struct` which is a copy of its own memory management structure. The parent thread then makes this new structure active by setting the `task_struct` memory management pointers to the new `mm_struct`. Meanwhile, the other threads are in a barrier waiting for the parent thread to complete creation and swap of memory management structures. Once the copy is complete, each thread then swaps it `task_struct` memory management pointers for the ones in the parent thread. Now, all threads are actively using the new management structure.

It is at this point that, our algorithm behaves like the `clone` system call. After swapping the old memory management structure for the new one, each thread concurrently invokes the `do_fork` routine. As previously indicated, it is this routine that does the work of process creation. However, each thread invokes the `do_fork` routine in such a way that it will share the current memory address space. So, each new thread created will use the new memory structure that was just allocated and made active.

Once all threads complete the `do_fork` routine, each thread then swaps the new memory management structure back for its old one. Thus, the new set of threads (children) are running in *copy-on-write* shared address space of their original parent threads.

On returning from the `checkpoint` system call, the children threads have a return value of zero and each parent thread has a return value of the child thread that it created. At this point, each parent thread could sleep itself or decide to lower its priority and slowly write its state to stable storage from which the program could be restarted in the event of a hardware failure.

To revert back to a previous checkpointed state in the multithreaded program (i.e., rollback), the children threads would signal the parent threads to wake-up and then kill themselves. Thus, the rollback operation is completely accomplished at the user-level. The parent threads could then decide to redo the checkpoint or progress forward depending on the needs of the application.

Below we discuss the specifics of our algorithm implementation starting with the new global data structures that where introduced into the Linux operating system.

3.2 Global Data Structures

In Algorithm 3.1, the new global data elements are presented. The design philosophy is that because this is operating system level software, correctness and robustness must be guaranteed to the greatest possible extent. In keeping with that design philosophy, we employ a multi-phase approach in which a barrier synchronization among all the threads is used between each phase.

Algorithm 3.1: GLOBAL DATA()

$int\ \ checkpoint\_waits := 0, 0, 0, 0$
$pid\_t\ \ checkpoint\_min\_pid := 0x7fffffff$
$spinlock\_t\ \ checkpoint\_mm\_lock :=$
$\quad SPIN\_LOCK\_UNLOCKED$
$struct\ \ mm\_struct * checkpoint\_mm := NULL$
$spinlock\_t\ \ checkpoint\_task\_lock :=$
$\quad SPIN\_LOCK\_UNLOCKED$
$struct\ \ task\_struct*$
$\quad checkpoint\_parent\_task :=$
$\quad NULL$

The first variable is `checkpoint_waits`. This array of four integers is used to implement the various barriers between phases. The `checkpoint_mm_lock` is a lock for the `checkpoint_mm` variable, which is a pointer to the current memory management structure that is being checkpointed among a group of threads. Since only one set of threads can be checkpointed at a time, `checkpoint_mm_lock` is used to prevent another set of threads from initiating a checkpoint operation until the current set is complete. The `checkpoint_task_lock` provides internal synchronization and coordination between phases. Finally, the `checkpoint_parent_task` is the pointer to the thread which is *master* (i.e., possesses the smallest process identifier) among all the threads involved in the checkpoint operation.

3.3 Core Algorithm

Algorithm 3.2: SYS_CHECKPOINT(*regs*)

```
return (DO_CHECKPOINT(regs))
procedure DO_CHECKPOINT(regs)
  ADMISSION()
  CREATE_MM(regs)
  CLONE_THREADS(regs)
  RESTORE_MM()
  LEAVE()
```

Algorithm 3.3: DO_CHECKPOINT(*regs*)

```
procedure ADMISSION()
  old_mm := current → mm
  if current → mm → mm_users = 1
    then { clone_flags := SIGCHLD
         { return (DO_FORK(clone_flags, regs))
  SPIN_LOCK(&checkpoint_mm_lock)
  while checkpoint_mm ≠ NULL
   and checkpoint_mm ≠ old_mm
    do { SPIN_UNLOCK(&checkpoint_mm_lock)
       { SCHEDULE_TIMEOUT(1)
       { SPIN_LOCK(&checkpoint_mm_lock)
  if checkpoint_mm = NULL
    then checkpoint_mm := old_mm
  SPIN_UNLOCK(&checkpoint_mm_lock)
  if current → checkpoint_counter = 0
    then mm_users :=
     current → mm → mm_users − 1
    else mm_users :=
     current → mm → mm_users
  SPIN_LOCK(&checkpoint_task_lock)
  if current → pid < checkpoint_min_pid
    then checkpoint_min_pid := current → pid
  checkpoint_waits_0 + +
  while checkpoint_waits_0 < mm_users
    do { SPIN_UNLOCK(&checkpoint_task_lock)
       { SCHEDULE_TIMEOUT(1)
       { SPIN_LOCK(&checkpoint_task_lock)
  SPIN_UNLOCK(&checkpoint_task_lock)
```

In keeping with Linux system call convention, `sys_checkpoint` is the top-level handler of the system as shown Algorithm 3.2. This handler routine invokes the architecture independent routine, `do_checkpoint`. This routine is divided into the following four phases: `admission`, `create_mm`, `clone_threads`, `restore_mm`, and `leave`. These phases correspond to the different parts of the algorithm as previous described at the start of this section.

The `admission` phase shown in Algorithm 3.3, determines which threads are allowed into the core parts of the checkpoint system call. The first part determines if there are no other threads sharing the current process' memory management structure (i.e., a single threaded/uniprocessor program). If so, the `checkpoint` system call behaves just like a `fork` system call by directly invoking the `do_fork` general handler routine. This is possible because the `do_fork` routine can handle the concurrent processing of `fork` system calls since shared variables are placed inside of critical sections.

Algorithm 3.4: DO_CHECKPOINT(*regs*)

```
procedure CREATE_MM(regs)
 if current → pid = checkpoint_min_pid
   then { checkpoint_parent_task := current
        { parent := current
        { if new_mm := ALLOCATE_MM() = NULL
        {   then { notify_other_threads_of_error
        {        { reset_global_variables
        {        { return (error)
        { MEMCPY(new_mm, parent → mm)
        { DUP_MMAP(new_mm)
        { COPY_SEGMENTS(new_mm)
        { old_mm := parrent → mm
        { parent → mm := new_mm
        { parent → active_mm = new_mm
        { ACTIVATE_MM(old_mm, new_mm)
        { SPIN_LOCK(&checkpoint_task_lock)
        { checkpoint_waits_1 := 1
        { SPIN_UNLOCK(&checkpoint_task_lock)
   else { SPIN_LOCK(&checkpoint_task_lock)
        { while checkpoint_waits_1 = 0
        {   do { SPIN_UNLOCK(&checkpoint_task_lock)
        {      { if parent_detects_error
        {      {   then return (error);
        {      { SCHEDULE_TIMEOUT(1)
        {      { SPIN_LOCK(&checkpoint_task_lock)
        { SPIN_UNLOCK(&checkpoint_task_lock)
        { parent := checkpoint_parent_task
        { old_mm := current → mm
        { current → mm := parent → mm
        { current → active_mm := parent → mm
        { current → mm → mm_users + +
        { ACTIVATE_MM(old_mm, current → mm)
```

Now, if `checkpoint_mm` is set, then that signals there is an existing group of threads in the process of checkpointing, so it must determined if the current process is with the current checkpoint group or not by comparing the `checkpoint_mm` variable to the process' memory management structure pointer, `mm`. If the process is with the checkpoint thread group, then it is allowed to pass through the barrier. Otherwise, it will wait using the `schedule_timeout` internal routine for a *jiffy* (i.e., 10 milliseconds). During this time, the Linux scheduler executes other runnable processes. This kind of barrier enables many threads bound to a single processor to be involved in a checkpoint operation.

Next, if `checkpoint_mm` is not set, then this process *atomically* sets the variable to the address of its memory management structure. Once a thread is admitted (i.e., moves past the first barrier), it determines the number of other threads in this thread group using the memory management structure's `mm_users` variable.

Algorithm 3.5: DO_CHECKPOINT($regs$)

```
procedure CLONE_THREADS(regs)
  current → checkpoint_counter + +
  clone_flags := (CLONE_VM|SIGCHLD)
  retval := DO_FORK(clone_flags, regs)
  current → checkpoint_counter − −
  SPIN_LOCK(&checkpoint_task_lock)
  while checkpoint_waits_2 < mm_users
       ⎧ SPIN_UNLOCK(&checkpoint_task_lock)
    do ⎨ SCHEDULE_TIMEOUT(1)
       ⎩ SPIN_LOCK(&checkpoint_task_lock)
  SPIN_UNLOCK(&checkpoint_task_lock)
```

Algorithm 3.6: DO_CHECKPOINT($regs$)

```
procedure RESTORE_MM(regs)
  new_mm := current → mm
  current → mm = old_mm
  current → active_mm = old_mm
  new_mm rightarrowmm_users − −
  ACTIVATE_MM(new_mm, old_mm)
```

The current process' `checkpoint_counter` variable records the number of times this process has been checkpointed. Currently, we are special casing the first checkpoint for LinuxThread programs. Recall, that LinuxThreads create a thread manager. Thus, the `mm_users` variable is one greater than the number of checkpointing threads. Consequently, we need to reduce the number of `mm_users` by one for the purposes of keeping an accurate count of the number of threads that will be involved in the checkpoint operation. This is crucial since the subsequent barriers block until every process has move into the barrier.

The last part of the `admission` phase is the election of the "parent" thread followed by a "task" barrier. The task barrier uses independent `wait` variables. This is done because with a large numbers of threads, it cannot be guaranteed that the last thread has left the previous barrier before the first one enters the next barrier. Last, these barriers only allow an atomic evaluation of the barrier condition. This conservative approach was taken to insure robustness. It may be possible to relax this condition, however, a more comprehensive analysis and testing on other processors would be required before any conclusions can be made about the efficacy of this synchronization approach.

Shown in Algorithm 3.4, the `create` phase allows the parent process to allocate a new memory management structure and then swap this new one for its original. During this allocation the other threads wait in the `checkpoint_wait`$_1$ barrier which releases them only when the parent has completed the allocation and swap of memory management structures. Once complete, each process will then swap the original memory management structure for the new one as well.

A closer look at the memory management structure allocation and swap process reveals a number of interesting details. To create a new memory management structure, the space is not only allocated, but also copied. After the copy, a Linux specific initialization routine is invoked, which is not shown in the algorithm. After that the virtual memory page tables are duplicated in the `dup_mmap` routine. We note here that in Linux this operation is encapsulated in a semaphore. Last, descriptor tables which are used by the processor to perform address translation are copied in the `copy_segments` routine.

The swapping of memory management structures requires that the old structure be deactivated and the new one must take its place. This is accomplished by the `activate_mm` routine.

With the new memory management structure created, the threads enter the `clone` phase, as shown in Algorithm 3.5. In it, each thread creates a child thread using the `do_fork` handler routine that will take their place and utilize the newly allocated address which is a *copy-on-write* instant of the original address space. Once complete, all threads synchronize in the third barrier.

Next, the original memory management structure needs to be restored (see Algorithm 3.6). The `restore_mm` completes this task by reverting back to the original memory management structure and then re-activating it.

Algorithm 3.7: DO_CHECKPOINT($regs$)

```
procedure LEAVE(regs)
  if parent = current
         ⎧ SPIN_LOCK(&checkpoint_task_lock)
         ⎪ while checkpoint_waits_3 ≠ mm_users − 1
         ⎪      ⎧ SPIN_UNLOCK(&checkpoint_task_lock)
         ⎪   do ⎨ SCHEDULE_TIMEOUT(1)
         ⎪      ⎩ SPIN_LOCK(&checkpoint_task_lock)
         ⎪ checkpoint_min_pid := 0x7fffffff
    then ⎨ checkpoint_waits := 0, 0, 0, 0
         ⎪ checkpoint_parent_task := NULL;
         ⎪ SPIN_LOCK(&checkpoint_mm_lock)
         ⎪ checkpoint_mm := NULL
         ⎪ SPIN_UNLOCK(&checkpoint_mm_lock)
         ⎩ SPIN_UNLOCK(&checkpoint_task_lock)
         ⎧ SPIN_LOCK(&checkpoint_task_lock)
    else ⎨ checkpoint_waits_3 + +
         ⎩ SPIN_UNLOCK(&checkpoint_task_lock)
  return (retval)
```

The last phase of the `checkpoint` system call is `leave`. As shown in Algorithm 3.7, the parent waits in the fourth barrier while all other processes exit. Once all threads have left, the parent resets all global variables, which allows the next set of threads to enter the system call and thus restarting the algorithm.

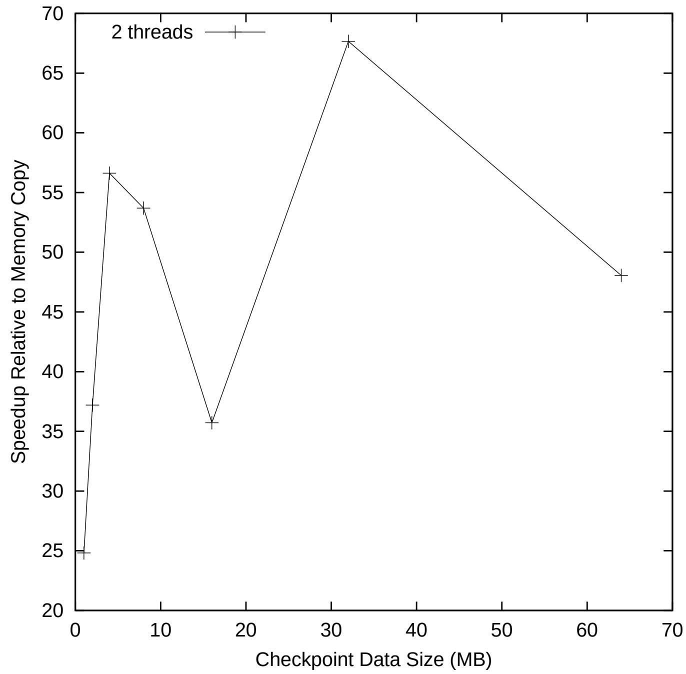

Figure 1: Speedup of 2 thread checkpoint relative to user-level memory copy.

4 Performance Study

4.1 Checkpoint vs. Memory Copy

The computing platform used in this study is a dual processor system running Linux 2.4.8. Each processor is a 400 MHz Pentium II. The total amount of physical RAM is 256 MB. We note here that the RAM is shared.

In this first series of experiments we compare the execution time of the `checkpoint` system call to a user-level memory copy method of checkpointing as a function of the number of threads and the amount of data being checkpointed. We measure performance in terms of *speedup* relative to memory copy (i.e., memory copy execution time divided by system call execution time).

In Figure 1, we observe that the speedup for the 2 thread case varies from 25 up to 67. These speedup results are attributed to the efficiency of *copy-on-write* semantics of the underlying virtual memory system. Interestingly, non-linear speedup behavior is observed. For instance, there is a large drop off in speedup when the data size changes from 8 MB to 16 MB, then a sharp increase at 32 MB followed by a sharp decrease at 64 MB. The cause of this non-linear behavior is not completely understood. We hypothesize that it is due to differences in the amount of data copied between memory copy and checkpoint at the various data points. However, a more thorough performance analysis of the Linux virtual memory subsystem is required before any definitive conclusions can be drawn.

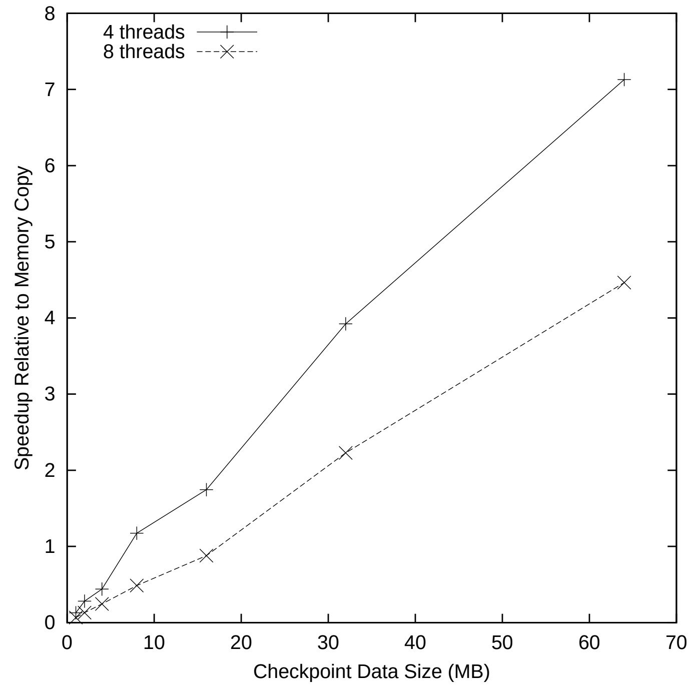

Figure 2: Speedup of 4 and 8 thread checkpoints relative to user-level memory copy.

Finally, the speedup results for the 4 and 8 thread cases are reported in Figure 2. There are between 2 to 4 times more threads than processors. Thus, each thread will context switch several times during the processing of the system call and generate much greater overheads. Because of this aspect, we observe a significant drop in speedup, particularly for small checkpoint data sizes. However, what is surprising is that at 32 and 64 MB data sizes the speedup results are above 4 for the 4 thread case and above 2 for the 8 thread case.

4.2 Start-to-Finish Results

As indicated in the first series of performance results, the high speedups are attributed to the *copy-on-write* semantics of the underlying virtual memory system. To better understand how these raw system call performance statistics would translate into overall start-to-finish program performance, we conducted a full program performance test were the start-to-finish execution time of a synthetic workload program was measured. The workload program consists of two threads and 64 MB of data. The synthetic threaded program performs ten checkpoint operations of system using

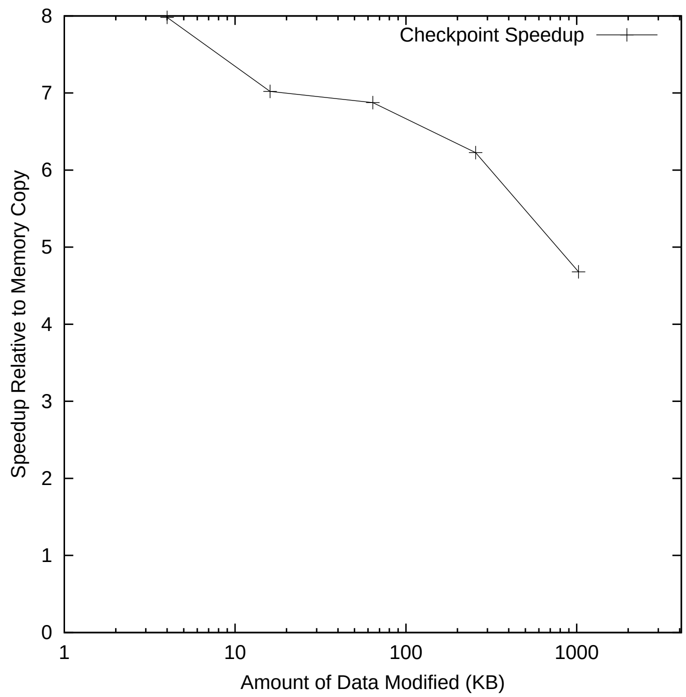

Figure 3: Speedup of 2 thread benchmark program using checkpoint relative to full memory copy.

either memory copy or the `checkpoint` system call. In between the checkpoints, the threads would modify a certain amount of the data. The amount of modified data is varied from 4KB to 1MB. The speedup results are report in Figure 3

It is observed that total execution of the program using the `checkpoint` system call is around one second with a small increase in total execution time as the amount of modified data is increased. However, we see that the memory copy execution time remains unchanged regardless of how much data is modified. When execution times are translated into speedup results, we see that overall program performance is increased by a factor of eight for small data sizes and a factor of five for the 1 MB data size when the new system call is used. We have observed that when the amount of modified data approaches the total amount of data in the program, the execution time is the same for both memory copy and the `checkpoint` system call.

5 Conclusions

The `checkpoint` system call is a new approach that leverages the copy-on-write semantics of virtual memory to enable a transparent, fast, reliable, consistent state copy of a large-scale, multithreaded program. In this paper we present our algorithm and its implementation in the Linux Operating System. Our performance results demonstrate that for many cases, the system call out-performs a user-level copy, particularly when the number of threads out-number the processors by a factor of 2 to 4 times. However, if the number of threads become significantly larger relative to the number of processors, then context switching overheads dominate the cost of the `checkpoint` system call and user-level checkpointing is faster. We point out, though, that this case is pathological in nature since the threaded program itself would fail to realize much performance benefit when run in such a configuration.

6 Acknowledgements

This research is supported by the DARPA's Network Modeling and Simulation program, contract #F30602-00-2-0537.