Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux

Michael Rieker, Jason Ansel, Gene Cooperman
College of Computer and Information Science
Northeastern University
Boston, MA 02115
Email: {mrieker,jansel,gene}@ccs.neu.edu

***Abstract*— Checkpointing of single-threaded applications has been long studied [3], [6], [8], [12], [15]. Much less research has been done for user-level checkpointing of multithreaded applications. Dieter and Lumpp studied the issue for LinuxThreads in Linux 2.2. However, that solution does not work on later versions of Linux. We present an updated solution for Linux 2.6, which uses the more recent NPTL (Native POSIX Thread Library). Unlike the earlier solution, we do not need to patch glibc. Additionally, the new implementation can take advantage of the ELF architecture to eliminate the earlier requirement to patch the user's `main` routine. This fills in the missing link for full transparency. As one demonstration of the robustness, we checkpoint the Kaffe Java Virtual Machine including any of several multithreaded Java programs running on top of it.**

***Index Terms*— checkpointing, user-level checkpointing, multithreaded, Linux, NPTL, NAS Parallel Benchmark**

I. Introduction

SMP computers are becoming more prevalent through commodity dual-core chips, and the roadmap of at least one prominent vendor calls for quad-core commodity chips. This makes imperative the need for checkpointing of multithreaded programs. Kernel-level checkpointing packages such as [7] require modifications to the kernel (or in some cases, kernel modules that must be updated along with new kernel versions). Application-level checkpointing packages such as [2] require the end user to add code for checkpointing.

We present a user-level checkpointing method that provides transparent library-based checkpointing of multithreaded processes. Dieter and Lumpp demonstrated an initial implementation for LinuxThreads for Linux 2.2 [5]. Unfortunately, that version does not work beyond the Linux kernel 2.2. The user-level checkpointing of multithreaded programs we present runs with NPTL (Native POSIX Thread Library) on Linux 2.6. We describe the issues encountered in this implementation.

The package checkpoints user address space (including libraries, data, stack, heap, etc.), threads, open file descriptors, signal state, mutexes, and process environment. It does not support checkpointing of shared memory, process ids, and sockets, since those are related to a collection of processes. As a demonstration, we checkpoint and restart an entire Java Virtual Machine including a running Java program.

a) Portability: This implementation is known to run on Linux 2.6 systems. We wrap the `clone` system call in order to maintain a list of threads in the process being checkpointed. Note that in Linux, `pthread_create` calls `clone`. Writing a wrapper for `clone` instead of `pthread_create` allows us to capture the location of the stack (an argument of clone), and therefore we can specify the location of the stack on restart. We also use the proc filesystem to retrieve the memory layout and open file descriptor list at checkpoint time.

b) Full transparency: Currently we require the user to link against a library and add one initialization call. In future even this will not be needed. It is possible to add an additional section to an ELF binary containing a wrapper for main and then modify the entry point of the binary to refer to that new code.

c) Layout of paper: Section II describes the implementation. At the time of checkpoint, all memory segments (as read from `/proc`) are saved to disk, along with their original address in memory.

Section II-A describes the checkpointing process. The system checkpoints a process by first, at program startup, spawning a checkpointing control thread. This thread will checkpoint the program on a user defined interval. To checkpoint, this checkpointing thread gains control of all other threads by using signals and then writes out all state to disk, after which it resumes the suspended threads then hibernates until time for another checkpoint.

Section II-B describes the restart process. Restart is done by calling an independent application `mtcp_restart`. This first `munmaps` all of its own memory segments outside of program code. It then rebuilds the original process by first remapping the checkpoint library `mtcp.so` from the checkpoint file. Control is transferred to that routine, which then continues reading the checkpoint file into memory to restore remaining addresses that were in use at the time the checkpoint was taken. Finally, all threads are restarted.

By checkpointing all memory segments (instead of just read/write segments), we avoid the difficulty of the randomized address space feature of Linux 2.6, where it will load libraries at randomly selected addresses each time a particular program is loaded. The kernel developers do this to slow down the progress of remote, automated vulnerability attacks. See Section III for details of how this is handled.

We demonstrate scalability by checkpointing NAS Parallel

Benchmarks, both the C version that uses OpenMP and the Java version under Kaffe. This has the potential to checkpoint all Java programs simply by checkpointing the Java Virtual Machine. For details on this see Section IV.

A. Related Work

Transparent checkpointing refers to checkpointing techniques that can be applied directly to application binaries. This can be a convenience (no source code transformations needed) or a necessity (vendor-provided software in binary-only format). There are two forms of such checkpointing: kernel-level (modifications to the kernel or use of kernel modules); and user-level (no kernel modifications). (A third checkpointing style, application-level, requires end users to make possibly non-trivial modifications to the source code.)

User-level checkpointing was implemented by Libckpt [12], followed by variations [3], [6], [8]. Variations have been introduced for more recent architectures, such as the Intel IA-64 under Linux [15].

Much less work has been done on transparent, user-level checkpointing. Dieter and Lumpp were the first to produce a user-level checkpointing program for multi-threaded processes in Linux [5]. Unfortunately, that work is based on LinuxThreads and runs only on Linux 2.2, which is now obsolete. Abdel-Shafi et al. present user-level thread migration and checkpointing on Windows NT by taking advantage of the Brazos run-time parallel system [1]. The current work describes the issues for Linux 2.6, which uses the NPTL (Native POSIX Thread Library), instead of LinuxThreads.

Other work on transparent checkpointing of multithreaded programs includes the application-level checkpointing of OpenMP applications [2], and a kernel-level checkpointing package with the additional advantage of checkpointing multiple processes on a network [7]. SafetyNet [14] provides a hardware-assisted solution for shared memory multiprocessors. See [11] for a web page listing other checkpointing packages. The work of this paper grew out of earlier work [4] on single-threaded checkpointing to efficiently support master-worker parallelism.

II. Implementation

The acronym MTCP stands for *MultiThreaded CheckPointing*. The checkpointing implementation relies on two binaries: `mtcp.so` and `mtcp_restart`. The main idea is that `mtcp.so` periodically saves the state of all threads, memory and list of open files to a checkpoint file. The `mtcp_restart` utility can reconstruct the process on demand from that checkpoint file. `mtcp.so` is the runtime library added to the users application. `mtcp_restart` is the command line utility used to restart a checkpoint.

Currently, to add checkpointing to an existing application, the user needs only to link against our `mtcp.so` and add a call to our initialization routine, `mtcp_init`, at the beginning of their `main` routine. In the next version (in progress), we eliminate even this minor lack of transparency. This is done by providing a utility to modify the user binary by adding an additional ELF section containing our initialization routine. The new section is made the entry point. It uses `dlopen` and `dlsym` to load and call `mtcp_init`, after which it calls the original entry function (`main`) of the user's binary.

Section II-A describes how `mtcp.so` saves the state of the user's program to disk. Section II-B describes how the restart utility, `mtcp_restart`, reconstructs and restarts the user's process. Section II-D discusses the more technical issue of how to make system calls during restart, when even `libc.so` is not yet resident.

A. Initialization and Checkpointing

The initialization and checkpointing is summarized in Figure 1.

The initial routine registers a signal handler for some signal (configurable) that is not used by the application. A wrapper function around `clone` can modify the input and output of `clone`, while using `dlopen/dlsym` to call the actual `clone` library code. It then sets up internal data structures so that these wrappers can track creation and deletion of threads. Finally, it spawns a *checkpointing control thread*, which arranges for periodic checkpoints.

Periodically, when it is time for a checkpoint, the checkpoint control thread signals all other application threads. The signal handlers cause those target threads to save their integer registers to their stack through setjmp/longjmp. In addition to setjmp/longjmp, an assembly routine writes the floating point registers to the stack. The target threads then wait on a futex (Linux native condition variable). The checkpoint thread then writes the state to disk. The state consists of:

1) the contents of all memory regions, including stacks, code, heap, any dynamically loaded libraries (via dlopen, etc), mmap'ed regions, etc. This also includes the register sets for each thread, since they were saved on their respective stacks.
2) a list of open file descriptors and the position within the file they are pointing to. Only "regular" files and ttys are saved.
3) and per-thread storage area descriptors and per-thread signal masks.

The details of saved thread state can be found in section II-C.

When the checkpoint file is written, the threads are released from waiting, and the checkpoint thread sleeps until it is time to write a new checkpoint.

The checkpoint control thread executes this loop:

```
Loop:
    Wait for a few seconds
    Signal all known threads to suspend
       and check for exited threads
    Wait for those threads to all suspend
       also check for exited threads
    If all known threads have exited,
       then this thread exits, too
    Write the checkpoint file
```

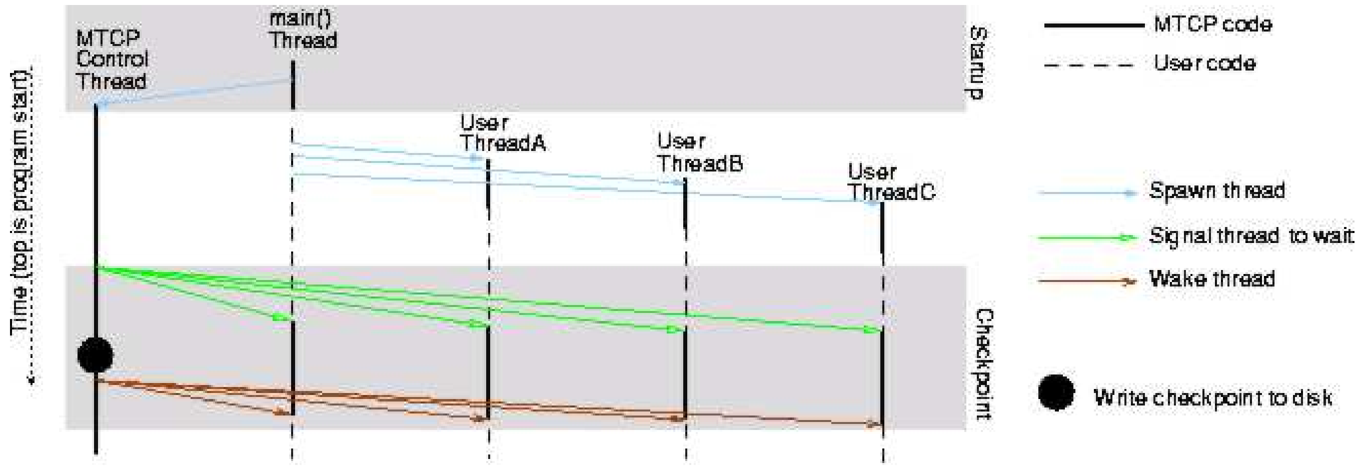

Fig. 1. Internals of MTCP (Multi-Threaded CheckPointing)

```
Resume the suspended threads
```

The wait operation consists of a simple `nanosleep` call. `Nanosleep` was preferred to `sleep` because it does not depend on `SIGALRM`, which may be in use by the application.

We loop through our internal list of known threads and use the Linux-specific system call, `tkill`, to send a user-configurable signal (e.g. `SIGUSR1` or other signal not conflicting with the program logic). The Linux-specific `clone` wrapper sets up a signal handler for the threads that, when called, will:

1) save the thread's state (registers, tls parameters);
2) sets a flag in its state structure indicating that its state is saved;
3) wake the checkpoint control thread;
4) and then suspend by waiting on a futex cell.

To prevent trouble in case of a crash during the writing of a checkpoint file, the checkpoint is first written to a temporary file. Then, when the temporary file is complete, the temporary file is renamed to the permanent name, thus replacing the previous checkpoint only with a complete checkpoint. If desired, it is easy to extend this scheme to save the last "n" checkpoints for added safety.

B. Restarting

To restart a process from a checkpoint file, the `mtcp_restart` command is invoked with the name of the checkpoint file as its argument. The contents of the address range that was mmap'ed from `mtcp.so` at the time of the checkpoint are written to the beginning of the checkpoint file. The `mtcp_restart` routine directly reads this into memory at the same address where it was found in the original application. Thus the `mtcp.so` portion of memory is restored. The `mtcp_restart` routine then calls a restart routine within `mtcp.so`.

The above assumes that there is no clash between the addresses used by `mtcp_restart` and those used by `mtcp.so`. Due to address space randomization in future versions of the Linux kernel, this may not always be true. If there were a clash, it would be possible for `mtcp_restart` to be linked at two widely separated addresses. Then if a conflict occurs with one `mtcp_restart` image against the `mtcp.so` being restored, the alternative `mtcp_restart` can be automatically loaded and the restart can proceed.

The restart routine then performs the following steps.

1) Switch to an internal stack area contained within the `mtcp.so` image's address range.
2) Unmap everything in user address space except the restored `mtcp.so` image. All of `mtcp_restart`, `libc.so`, etc. are unmapped. Specifically, we call `munmap` on two address ranges: from address 0 to the beginning of `mtcp.so`; and from the end of `mtcp.so` to the beginning of kernel space.
3) Read the rest of the memory contents from the checkpoint file into exactly the same addresses from which they were written. There can be no address clash, since `mtcp.so` was read into memory from that same checkpoint file.
4) Restore file descriptors and the offsets within files. Restore signal handlers.
5) Create a new thread for each thread that existed at the time of the checkpoint and have each thread again wait on the original futex. (The thread is created using the Linux-specific `clone` call, and the original stack is passed to `clone`. The thread calls a `restart_thread` routine, which calls longjmp to re-enter the original signal handler routine and again wait on the original futex.)
6) Resume all the threads. We call `futex` once for each thread to unlock the thread. Each thread will then restore its registers and signal masks from its own stack and resume execution where the checkpoint was taken.
7) The checkpoint thread then sleeps until it is time to write another checkpoint.

The restoration of memory associates the original mapped files as much as possible, by saving the state indicated by the filesystem information from /proc/self/maps. This information consists of the memory address range, the access permissions and a filename (if any). Thus, subsequent checkpoints will have this information available. Also, by using /proc/self/maps to determine the current memory configuration, a program is free to *mmap* and *munmap* segments at will, and the checkpoint is able to save an accurate memory image. Finally, by using /proc/self/maps, it is not necessary to place wrappers around mmap and munmap.

We found it necessary to wrap the clone system call to track the creation of threads (see section II-A). By intercepting the child_tidptr parameter and forcing the CLONE_CHILD_CLEARTID flag, we are able to detect when a thread has exited and thus we know we do not have to signal it to suspend when it is time for a checkpoint.

There are some points about item 2, above, that are worth noting. First, by unmapping all other addresses outside of mtcp.so, it has also unmapped libc.so. Hence, it cannot make calls, even system calls in libc.so, that are not part of the mtcp.so image itself. The solution to this is described in Section II-D, below.

Second, during the restart process, mtcp.so cannot reference any memory outside the mtcp.so image address range, such as would be provided by malloc, as it might interfere with restoring the restarted process' memory areas. This implies that some system calls, such as fopen, cannot be used for restarting, because they call malloc.

C. Saved Thread State

We also save information about each thread in memory, so that it will be checkpointed. This information includes:

1) tid — the thread-id as returned by gettid(). It is used to determine which is the 'current' thread. It remains valid even after the thread has exited.
2) mtcp state — the state mtcp considers the thread to be in. This is an int who's possible values are RUNDISABLED (running with checkpoint disabled), RUNENABLED (running with checkpoint enabled), SIGDISABLED (has been signaled to suspend, but checkpoints are disabled and so it continues to run), SIGENABLED (has been signaled to suspend with checkpoints enabled), SUSPINPROG (suspend in progress), and SUSPENDED (state is saved, waiting to resume). The state may be updated only via atomic updates (in practice, by the thread itself or by the checkpoint thread). The Linux futex system call is used by threads to wait for a change of the mtcp state.
3) parent, children, siblings — these links are used to keep track of the thread hierarchy. This is necessary so that, upon restore, the threads are re-created with the same hierarchy, in case the application depends on that hierarchy.
4) clone_flags, parent_tidptr — these are the values the application originally passed to the clone through pthread_create. Mtcp detects these values via a wrapper around clone. It saves the values in order to recreate the clone with the original flags during restart. See given_tidptr and actual_tidptr for exceptions.
5) given_tidptr, actual_tidptr — these are normally copies of the child_tidptr parameter passed to the wrapper for clone as its sixth argument. However, mtcp uses the child_tidptr functionality of the clone system call to detect when a thread has exited. The Linux system will clear (zero out) the location pointed to by child_tidptr when the thread exits (since Linux 2.5.49). Since the child_tidptr is an optional parameter to the clone call, if the original call from user code did not call this value, then the the mtcp clone wrapper must fill in this parameter with a location of its own before invoking the actual clone system call.
6) Sigblockmask, sigaction, jmpbuf, fs, gs, gdtentrytls — these fields are used to save the thread's state while it is suspended. They are saved and restored just like any other memory location in the restored image. When a restart is performed, our restartthread function will load these values into the corresponding thread's kernel context and the thread will resume processing as before. Restartthread does its work via the system calls set_thread_area (for gdtentrytls), sigaction, sigprocmask (for sigblockmask) and an assembler mov instruction to load the fs and gs registers. Finally, it calls longjmp on jmpbuf to restore the stack pointer and other registers.

D. Making System Calls during Restart

We need to have all references by the mtcp.so restart routine resolved within the mtcp.so image itself, since, when mtcp.so restart is running, it can only depend on addresses within its own image mapped. Everything else, including libc.so, has been unmapped. (See Section II-B.)

Simply performing a static link of mtcp.so with libc.a presents a problem. Suppose a restart routine within mtcp.so references a system call such as read. This leads to the following problematic scenario.

1) Linking mtcp.so statically dutifully includes read.o from libc.a in mtcp.so.
2) When the application, linked to mtcp.so, is run, however, the dynamic loader links the restart routine's read call to the external libc.so rather than to the internal copy of read.o.
3) When the restart is attempted, the mtcp.so restart routine points to the read within the application's libc.so. But it no longer has access to this dynamically linked read, as it hasn't been restored yet. The internal, statically linked copy of read, though still present, is not referenced.

To work around this problem, for each system call used by the restart program, we:

1) extract the object file from libc.a;

2) disassemble to a source file; and
3) include that disassembled source as part of the assembly of the restart module.

Since all sources for the restart module are assembled as one source file, as far as the dynamic loader is concerned, it has no external symbol references as they are all resolved at assembly time. Thus the dynamic loader will not be able to redirect any references to libc calls, such as read, to the external `libc.so`. If the routines were simply extracted in object form from `libc.a` and linked in the `mtcp.so` image, the dynamic loader would see the 'external' references and direct them to an external `libc.so` image at runtime, thus breaking the requirement that `mtcp.so` have all references be internal to itself.

Currently, the system calls that are disassembled and statically linked into `mtcp.so` are: `close`, `dup`, `dup2`, `getppid`, `lseek`, `mmap`, `mprotect`, `munmap`, `open`, `read`, and `write`.

The disassembly is done automatically by a utility we provide that is invoked by the makefile, so any updates to those system services in glibc will be taken into account by re-making the `mtcp.so` image. Should a future change to glibc make this method unworkable, another possibility would be to link `mtcp.so` with a static `libc.a`, then post-process the `mtcp.so` image with a utility that would resolve the linkages internally to the `mtcp.so` image.

In fact, we use "nocancel" versions of the above system calls, such as `__close_nocancel`. This is because the standard version `close` refers to the `tls` (thread local storage), which has not been restored yet. We don't need to refer to thread local storage, since we have not yet restored any threads at the time that we make these system calls. We are considering replacing the calls to `__close_nocancel` by a direct system call, for example `syscall(SYS_close,` *fd*`)`. This avoids the unwanted glue code in `libc.so`. This has the further advantage of removing the requirement to disassemble routines from `libc.a`.

III. Address space randomization

Address space randomization (since Linux 2.6.9) and the exec-shield patch (commonly used in some versions of Red Hat and Fedora Linux) are additions to the Linux kernel that randomize the initial location of the stack and of the base load address of dynamically loaded libraries such as `libc.so`. This is an attempt to thwart attacks on a system by exploiting bugs in coding, where the program will allow erroneous input to corrupt the program stack. Such an exploit will cause a return address on the stack to be overwritten thus causing the program to execute code that will compromise the system, allowing unauthorized access. These exploits rely on code and data being at known positions. By randomizing them to some degree, an attacker will need to try several possibilities before succeeding, thus discouraging or delaying the attack. See [13] for further analysis.

By using address space randomization, there will be no fixed address for the address of a buffer on the stack (for the buffer overrun) and no fixed address for the exploit code (in some data segment). This makes the job of the attacker more difficult. However, it also makes the job of the checkpointer more difficult if the image loader will not guarantee to load a code or data segment into the requested memory address range. For this reason, we must query the kernel via the proc filesystem to determine where libraries are actually loaded. The details were discussed in Section II-B.

IV. Experimental Results

In our first test (Figure 2) we attempted to see how scaling memory usage would effect the time to checkpoint a program. Here we checkpoint a single threaded program using a variable amount of memory set to random values. The memory is allocated in 1 MB chunks using `malloc`. The results shows that, with our hardware, that time to checkpoint scales linearly with memory usage at the rate of approximately 0.3 seconds per 100 MB of memory usage. There is also an additional cost of checkpointing because the operating system writes data to disk asynchronously. This leaves two timings to report (a) interruption in execution of the users program, (b) time for checkpoint data to asynchronously reach disk. We approximately measure this "time to reach disk", by timing the command "sync".

In our second test (Figure 3), we attempt to see how scaling the number of threads would affect the time to checkpoint a program. Here our test program creates a predetermined number of threads using `pthread_create`. This test was run on a single processor machine, but all threads wait on a mutex so that the checkpointing routine will not be starved for CPU cycles. These tests also show a linear increase in time with the number of threads, but take longer than expected.

The extra time here is because we do not (yet) optimize for zero-pages, which are not mapped to physical memory. When a pthread is created approximately 8 MB of memory addresses are reserved for that thread, but these pages are not actually allocated until they are first written to. Our current code will write all 8 MB of address space to disk. So the 100 thread test, produced a checkpoint of approximately 800 MB. Not handling zero-pages specially also has the disadvantage of forcing the physical allocation of all pages of memory on restore, thus increasing the memory usage of the restored program. Optimizing the handling of zero-pages is planned for a future version and will greatly increase the speed of checkpointing programs with many threads.

To test our checkpointing solution with a real world application we ran NAS Parallel Benchmark (NPB) [10], with our checkpointing package. NPB has been ported to several different programming languages, we chose the OpenMP version because it provides an example of a multithreaded process, for us to checkpoint.

As a more robust test, we checkpointed the entire Java Virtual Machine (JVM) including the Java program on which that virtual machine was running. Specifically, we checkpointed the Kaffe implementation of the JVM. We successfully checkpointed Kaffe running many small Java test programs

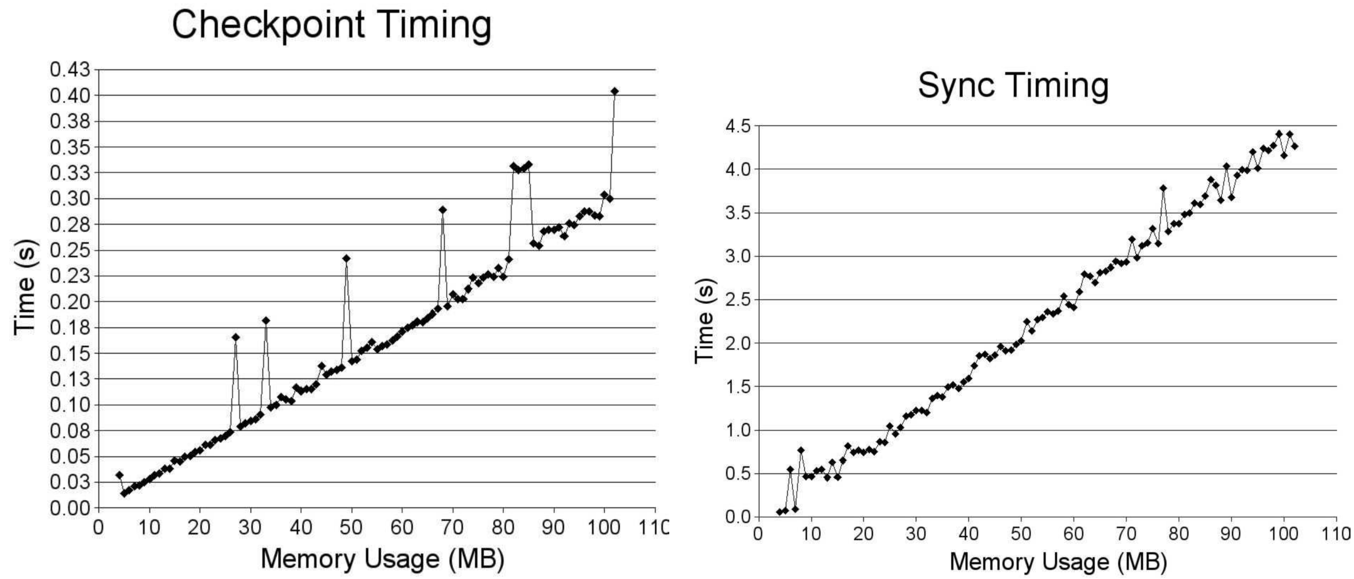

Fig. 2. **Left:** Delay in execution of user program to create checkpoint as the size of memory increases. **Right:** Time for checkpoint data to asynchronously reach disk as the size of memory increases.

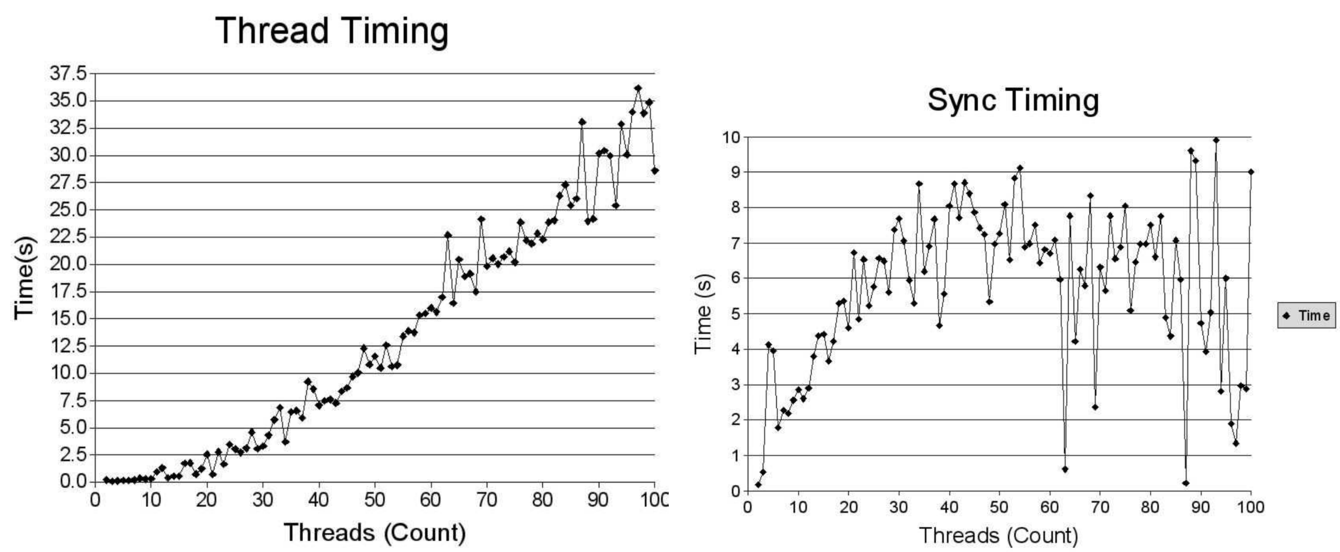

Fig. 3. **Left:** Delay in execution of users program as number of threads increases. **Right:** Time for checkpoint data to asynchronously reach disk as number of threads increases.

NAS CLASS	Size	Checkpoint	Restart
W	33x33x33	0.07 s	0.07 s
A	64x64x64	0.19 s	0.17 s
B	102x102x102	0.95 s	0.51 s

Fig. 4. Timings running the benchmark LU from NPB2.3-omp-C [9] (in C / OpenMP, derived from NAS Parallel Benchmark NPB-2.3-serial)

NAS CLASS	Size	Checkpoint	Restart
W	175 MB	0.7 s	0.6 s
A	632 MB	12.0 s	19.7 s

Fig. 5. Timings running the benchmark MG from NAS Parallel Benchmark NPB-3.0-JAV (in Java, on top of the Kaffe Java Virtual Machine)

and the Java port of the classic NAS Parallel Benchmarks (NPB) [10].

All timings, except for the last two were run on a Mobile AMD Athlon 64 processor 3000+ with 512 MB RAM, running Linux 2.6.12. The last two timings, concerning Kaffe and OpenMP, were run on a Pentium 4 2.4 GHz processor with 1 GB of RAM, running Linux 2.6.12.

Figure 6 demonstrates that the time for restarting a process grows linearly with memory usage at the rate of approximately

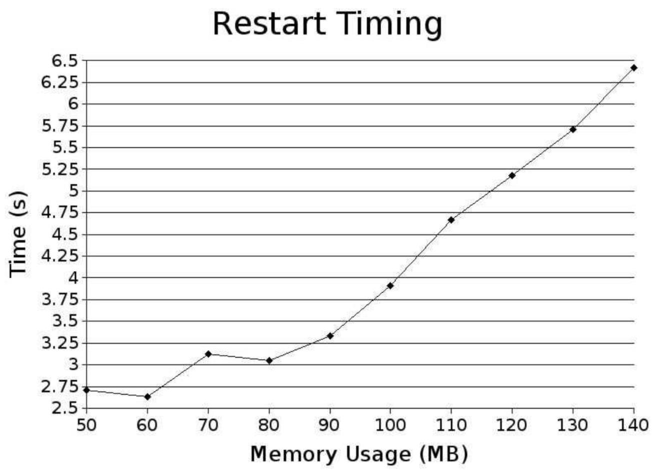

Fig. 6. Time to restart a process with 4 threads as the size of memory increases.

25 MB/second. Since this is a mobile hard disk (2-1/2" ATA disk), we assume that the restart process is limited only by the bandwidth of disk.

Acknowledgement

This research was supported in part by the National Science Foundation under grant number 0342555.

References

[1] Hazim Abdel-Shafi, Evan Speight, and John K. Bennett. Efficient user-level thread migration and checkpointing on Windows NT clusters. In *Usenix 1999 (3rd Windows NT Symposium)*, pages 1–10, 1999.

[2] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 235–247, New York, NY, USA, 2004. ACM Press.

[3] P. Emerald Chung, Woei-Jyh Lee, Yennun Huang, Deron Liang, and Chung-Yih Wang. Winckp: A transparent checkpointing and rollback recovery tool for Windows NT applications. In *Proc. of 29th Annual International Symposium on Fault-Tolerant Computing*, pages 220–223, 1999.

[4] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, Singapore, 2006. IEEE Press. to appear.

[5] William R. Dieter and James E. Lumpp Jr. User-level checkpointing for LinuxThreads programs. In *USENIX Annual Technical Conference (FREENIX Track)*, pages 81–92, 2001.

[6] K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada, and P.M.A. Sloot. The implementation of dynamite — an environment for migrating PVM tasks. *Operating Systems Review*, 34:40–55, July 2000.

[7] Oren Laadan, Dan Phung, and Jason Nieh. Transparent networked checkpoint-restart for commodity clusters. In *2005 IEEE International Conference on Cluster Computing*. IEEE Press, 2005.

[8] 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, Wisconsin, April 1997.

[9] NAS parallel benchmarks in OpenMP.

[10] The NAS parallel benchmarks, 1994.

[11] The Home of Checkpointing Packages. `http://www.checkpointing.org/`.

[12] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proc. of the USENIX Winter 1995 Technical Conference*, pages 213–323, 1995.

[13] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[14] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Weimin Zheng, Wenguang Chen, Youhui Zhang, and Ruini Xue. Thckpt: Transparent checkpointing of Linux processes under IA-64. In *Proc. of PDPTA-05*, pages 325–331, 2005.