Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

Title

The design and implementation of Berkeley Lab's linux checkpoint/restart

Permalink

https://escholarship.org/uc/item/40v987j0

Author

Duell, Jason

Publication Date

2005-04-30

eScholarship.org Powered by the California Digital Library
University of California

The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart

Jason Duell
Lawrence Berkeley National Laboratory
<jcduell@lbl.gov>

1. Introduction

Clusters of commodity computers running Linux are becoming an increasingly popular platform for high-performance computing, as they provide the best price/performance ratio in the marketplace. But while the size and raw power of Linux clusters continues to increase, many aspects of their software environments continue to lag behind those provided by proprietary supercomputing systems. One feature missing from Linux clusters is a robust, kernel-level checkpoint/restart implementation that can support a wide variety of parallel scientific codes. This paper describes Berkeley Lab's Linux Checkpoint/Restart project (BLCR), which seeks to provide a foundation for this capability. BLCR can be used either as a standalone system for checkpointing applications on a single machine, or as a component by a scheduling system or parallel communication library for checkpointing and restoring parallel jobs running on multiple machines. BLCR has established a collaboration with the LAM MPI developers, who have modified LAM to work with BLCR's user level hooks. As a result, it is already possible to checkpoint and restart simple parallel scientific codes written in MPI (such as the NAS Parallel Benchmarks [NPB02]) with BLCR. While more work needs to be done to get BLCR ready for use in a production environment, this work is in progress, and the modular design of BLCR should make it the logical choice for a wide range of cluster management and communication projects that are underway for Linux clusters.

2. The Uses of Checkpoint/Restart

The ability to checkpoint and restore applications (i.e. save the entire state of a job to disk, then later restore it) provides many useful features in a cluster environment:

2.1 Gang scheduling

Being able to checkpoint and restart a set of parallel processes that are part of a single application (gang scheduling) allows for both more flexible scheduling, and higher total system utilization. System administrators can allow different types of jobs to run at different times of day (favoring large, long jobs at night, for instance, and short interactive ones by day), simply by checkpointing any jobs that are still running when the scheduling policy changes, then restarting them when the policy changes back. A subset of nodes or even the entire cluster may be brought down for system maintenance without interfering with user job completion. Without checkpoint/restart, these types of scheduling actions must be implemented either by simply killing user applications (thus wasting the resources they have consumed so far), or by 'queue draining', in which no new jobs are accepted after a certain time, and the system waits for users jobs to finish (during which, by definition, the system runs at less than full capacity). Checkpoint/restart also allows extremely large jobs that may consume the entire system for a long duration (these are often the motivating applications for building a large cluster) to be intermittently scheduled, avoiding locking out all other users for what would be unacceptably long periods of time. Finally, checkpoint/restart can provide higher total system utilization by allowing the running of whatever configuration of available processes will most fully saturate the number of CPUs (or other resources) provided by the cluster.

2.2 Process Migration

A well-designed checkpoint/restart can allow checkpointed processes to be restarted not only on their original node, but also on other nodes in the system (or possibly even nodes on a different system that are part of a computing grid or other distributed system). Such migration can allow a job to continue if the imminent failure of one of its nodes

is detected (such as a failing CPU fan or local disk). Certain partial failures can also be worked around. For instance, most Linux clusters that use custom high-performance networks also provide a standard Ethernet interface for administrative traffic. If a node's high-performance network interface becomes unusable, it may still be possible to migrate its processes onto a fully performing node via the Ethernet interface, allowing a job that would otherwise be terminated to continue.

Process migration has also proved extremely valuable for systems whose network topology constrains the placement of processes in order to achieve optimal performance. The Cray T3E's interconnect, for instance, uses a three-dimensional torus that requires processes that are part of the same parallel application to be placed in contiguous locations on the torus. This naturally results in fragmentation as jobs of different sizes enter and exit the system. With process migration, jobs can be packed together to eliminate fragmentation, resulting in significantly higher utilization: the appearance of checkpoint/restart on the T3E has been credited as one of the major factors that allowed NERSC's T3E installation to go from 55% to 95% utilization [WONG99]. While networks with such constraining topologies have recently been out of style, IBM's Blue Gene L project plans to use one [IBM02], and more cluster projects may use them in the future.

2.3 Periodic Backup

Finally, checkpoint/restart can provide an increased level of reliability in the face of node crashes and/or nondeterministic application failures (such as infrequently occurring synchronization bugs). If intermittent checkpoints are taken, a job can be restarted from the most recent checkpoint, and should continue successfully, assuming its actions are idempotent. Some special handling for files may be used by a checkpoint system to 'rollback' changes to an application's open files, making many otherwise nonidempotent applications idempotent. This is particularly true for scientific applications, which tend to write their output files in a serial, log-like fashion: a simple truncation of files to their length at checkpoint time suffices for rollback in such cases.

2.4 Implementation Priorities

BLCR values these three advantages of checkpoint/restart in descending order of importance, based both on input from our funding sources and on the degree of difficulty involved. Gang scheduling and process migration are both high priorities, and present a similar level of difficulty. Process migration is slightly more difficult in that an efficient implementation (which moves the job directly over a network connection, rather than through an intermediate file) requires that access to the checkpoint file be contiguous, without seek() calls. Migration across non-homogenous nodes also requires support for copying any shared libraries and files that an application may be using as part of the checkpoint. BLCR currently does all of its checkpoint file accesses without seeks, and will soon provide the needed behavior for files, so it should prove equally useful for both gang scheduling and process migration.

Using checkpoint for periodic backup is more challenging, in that the overhead of taking a checkpoint becomes a key issue. The more that checkpoints delay the normal operation of a program, the less likely it is that users will use them for backup. BLCR currently uses an algorithm that stops an application during the entire time that a checkpoint is written, making it unattractive for frequent checkpointing of large jobs. Even if the delay to application execution is lessened, system administrators may be unwilling to accept the large amounts of I/O and disk space needed to implement periodic checkpointing on a large cluster, particularly if the checkpoint files need to travel over the network to reach stable storage. Finally, large user applications often already do their own checkpointing for fault tolerance, and can often do it much more efficiently than an automated checkpoint system can, since they know exactly which parts of their application need to be saved and which can be discarded or regenerated. Periodic backup is thus less of a priority for BLCR, though in our future work section we discuss some possible optimizations that should make it more suitable for that purpose.

3. The Design of BLCR

There are at least three axes of design for any cluster-oriented checkpoint/restart implementation: 1) whether to

implement checkpoint and restart at the user level or the kernel level; 2) how comprehensive a list of program behaviors to support; and 3) how tightly the checkpoint/restart capability should be integrated with other aspects of the cluster operating environment. BLCR takes the position that the best design for checkpoint/restart for Linux clusters today is one which 1) is kernel based; 2) provides automatic support for checkpointing the widest range of program features without building distributed operating system functionality or other cluster awareness into the kernel; and 3) provides a user-level callback interface to allow libraries and applications to support behaviors not automatically handled in the kernel's own checkpoint logic.

3.1 User vs. Kernel level checkpoint/restart

Many existing checkpoint/restart projects (particularly open source ones) use a user-level strategy for implementation [REFS]. In this approach, the operating system is unmodified, and remains completely unaware of checkpoints and restarts. In order to save and restore the state of a program, user level checkpoint/restart implementations intercept a wide range of system calls, in order to keep track of program state (such as memory mapped regions and open file handles) that needs to be saved during a checkpoint and restored during a restart. While this user-level approach is certainly viable, in the eyes of the BLCR designers it comes with a number of problems. From a basic design standpoint, it seems undesirable to replicate kernel data structures in user-space: this type of shadowing is easy to get wrong, and the addition of a new system call or other kernel change can easily cause the user-level tracking to become unsynchronized in ways that are not obvious to trace. In contrast, a kernel-level checkpoint implementation can simply access the data structures it needs right in the kernel, reducing the potential for such inconsistency.

Also, user-level checkpoint/restarts by their very nature cannot fully support the restoration of any resources that are not fully specifiable by user APIs. The process id and session id of a job, for instance, cannot be restored to their original values in a user-level implementation. This rules out checkpointing a wide range of applications (such as some standard Unix shells and scripting languages) that may rely on their parent's and/or children's pids remaining constant. Users in production supercomputing environments are accustomed to being able to wrap their jobs in a script that sets up and invokes their main application, then cleans up and/or postprocesses results after it exits. This is thus an important class of applications to support. While the current implementation of BLCR only supports the checkpointing of single processes (including multithreaded processes), work is underway to support the checkpointing of process groups and sessions, and it is expected that a full range of the standard Unix tools such as shells and scripting languages like Perl and Python will eventually be checkpointable.

3.2 Supported and unsupported program behaviors

Any production checkpoint implementation must carefully examine the range of Unix program features and behaviors to decide which to support. BLCR is no exception. Certain behaviors were considered too hard to support correctly, and/or are seldom used by the types of parallel scientific applications that are the project's focus, and have been left to either future work or user-level callbacks. Table 1 provides a list of the features that BLCR currently supports, as well as those for which support is planned, and those for which it is not. The initial version of BLCR support multithreaded programs, but not file handles, which may seem like a rather curious implementation priority, given our initial target of MPI applications (which are not typically multithreaded): this is due to LAM's need for thread-based callbacks, as described in section 4.2. TCP, UDP, and other network resources are not supported for the reasons discussed in the next section. Ptraced programs, and those which use asynchronous I/O, were considered too hard to promise as deliverables to our funding agents, although we intend to try to make them supportable within the larger BLCR design as a future extension, if possible. System V IPC mechanisms were not seen as frequently used by scientific applications, and were thus also excluded from our list of promised features, although we intend to implement as large a subset of them as proves possible given the project's resources. Finally, certain features, such as the ability to restart programs that have opened device and /proc files, are partially supported. The very common /dev/null and /dev/zero files are the only supported device

Feature	Implemented	Planned	Won't Support
Single-threaded processes	X		
Threaded processes	X		
Process groups		X	
Sessions		X	
User checkpoint hooks	X		
Signal state/handlers	X		
Restart system calls	X		
Ptraced processes			?
PID & PPID	X		
UID/GID		X	
Resource limits		X	
rusage info		X	
Regular files		X	
Stdin/stdout/stderr	X		
Unnamed & named pipes		X	
TCP/UDP sockets			X
mmapped files	X		
Shared mmap regions		X	
Block device files		/dev/null, /dev/zero	
/proc files		/proc/<pid>, if pid in checkpoint	
File locks		X	
Asynchronous I/O			?
Pinned memory		X	
mprotect()ed memory	X		
System V IPC			?
Pthread mutexes/conditions	X		

Table 1: Features supported by BLCR

files, and files within the Linux-specific /proc filesystem will only be supported if they are located within /proc/<pid> directories that belong to processes within the checkpoint.

3.3 Modularity versus tight integration with cluster software

It would seem natural to expect that a checkpoint/restart implementation geared towards a cluster environment would handle automatic checkpointing of processes that use network connections. BLCR, however, excludes TCP/UDP sockets from its list of supported features. This exclusion is deliberate, and reflects our philosophy that 1) it is preferable to not implement a feature than to implement it only partially or in a way that is not completely transparent; and 2) checkpoint/restart, at least at this moment in time, is best implemented for Linux clusters in a manner that does not tie the implementation to any particular clustering software, and does not try to add distributed operating system features to the kernel.

Programs that use network sockets may rely on the IP address of their sockets. This address cannot reliably be restored if process migration is to be supported. While it would certainly be feasible for a checkpoint/restart implementation to support the virtualization of the IP address returned by getsockaddr() so that it remained the same across process migration, this would either require the implementation to coordinate extensively with user-level clustering software, or force distributed operating system features into the kernel. Unless a fully distributed, cluster-wide IP stack were implemented (such that virtual IP addresses could be used by external machines and still get forwarded to the correct process, even if it were migrated to a different node), transparency would still not be achieved, as there would still be a need to distinguish between 'in-cluster' and exogenous sockets. An ftp server that wishes to hand out an IP/socket combination to a client to setup a file transfer, for instance, must give it a real IP address that will actually reach the server, not a virtual one that only has meaning within the cluster.

Such a fully distributed approach has a place in a richly integrated, top-down approach to cluster design: Hewlett-Packard's Single System Image project for Linux, for instance, uses precisely such a mechanism as part of an effort to systematically make system resources logically separable from the individual cluster nodes that provide them [REF]. BLCR, however, is designed in a more traditional, "bottom-up" manner to be a component that can be used by a variety of different clustering systems. We feel this approach has an important place in the current design space for scientific Linux clusters, in which a large number of competing projects and approaches currently exist, and where the one that will eventually predominate (if a single one ever does) is not yet clear. By avoiding any assumptions about the nature of the cluster environment it is operating within, BLCR can provide checkpoint/restart capability to any cluster software whose authors are willing to interoperate with its interface.

Fortunately, given that the vast majority of scientific applications use a library like MPI to do their communication rather than directly using network sockets, BLCR's lack of automated support for saving and restoring sockets does not come at the cost of usability. Once a communication library has been modified to interact with BLCR's user interface to correctly handle checkpointing, user applications gain the benefit of checkpointing without any modification to user code.

4. The User-Level Interface

We now describe BLCR's user interface, and how it has evolved though our collaboration with the LAM MPI Project.

4.1 Callbacks and Critical Sections

At a conceptual level, BLCR presents a very simple interface to libraries/applications that need to interact with checkpoint/restart. BLCR provides a way to register user-leve callback functions, which are triggered whenever a checkpoint is about to occur, and which continue when a restart is initiated (or a checkpoint done for backup purposes completes). These callbacks allow the application to quiesce its network activity and close its sockets (or perform analogous actions on some other uncheckpointable resource) before a checkpoint is taken, then restore them afterwards. Callbacks are designed to be written in the following style:

```
void my_callback(void *data_ptr)
{
        struct my_data *pdata = (struct my_data*) data_ptr;
        int did_restart;

        // do checkpoint-time shutdown logic

        // tell system to do the checkpoint
        did_restart = cr_checkpoint();

        if (did_restart)
                // we've been restarted from a checkpoint
        else
                // we're continuing after being backed up
}
```

The state of the callback's execution is itself saved during checkpoint (at the cr_checkpoint call), and restored at restart, or after the checkpoint is complete, for periodic backups. This allows the callback to pass state from checkpoint to restart context directly on the stack, via local variables. At registration time, an arbitrary pointer value may be provided, and it will be passed to the callback as an argument whenever it is run. Finally, the callback can tell from the return value of cr_checkpoint() whether it is returning as part of a restart, or is continuing after a periodic checkpoint (most callbacks will not care, but some may be able to perform optimizations based on this information). A single application may have multiple callback functions, and these callbacks do not need to know of each other's existence (this is needed to allow multiple unrelated libraries with checkpoint hooks to be linked into a single application). The cr_checkpoint function takes care of determining whether there are more callbacks to run, or whether all callbacks have completed and the process should perform the system call needed to initiate the actual checkpoint (the framework also handles rogue callback functions that never call cr_checkpoint, treating them as if a cr_checkpoint call were inserted as the last line of the routine).

BLCR also provides user-level code with "critical sections," in order to allow groups of instructions to be performed atomically with respect to checkpoints. These critical sections make it easier for libraries to be made checkpointable: hooks only need to be added to handle the regular operating state of the library, while special cases (such as network initialization) can be guaranteed not to be interrupted by a checkpoint. In some cases such atomicity is not merely a matter of convenience, but is vital to correct program operation, such as in the following code, in which a callback function has been provided for a closed source library that requires special checkpoint handling:

```
CR_CS_ENTER()
handle h = closed_lib_init();
cr_register_callback(&my_callback, &handle);
CR_CS_EXIT()
```

Without the critical section, this code could be interrupted by a checkpoint in between the initialization of the closed library and the registration of the callback that is required to checkpoint it correctly.

To allow the widest range of application behaviors, BLCR's critical sections are designed to permit nesting of critical sections. A library implementer can thus safely call a function in another library within a critical section, without worrying whether the called function will enter its own critical section. Critical section entries/exits do not even need to be properly nested: a function can enter a critical section and return without freeing it, so long as a matching CR_CS_EXIT call is eventually made at some point by the program. Care has also been taken to allow critical section entries/exits within signal handler context. These last features have been added with an eye to allowing programs that use asynchronous I/O to operate with checkpointing: while BLCR will not support taking a checkpoint while an asynchronous I/O operation is in progress, applications that need to use them can enter a critical section before initiating their I/O, then leave the critical section within the SIGIO signal handler that is

called when the I/O has completed.

4.2 Initial design: signal-based callbacks

To initiate a checkpoint within an application, BLCR sends it a signal (one of the real time signals is currently reserved for this use: in the future work section we discuss an alternative method that will avoid reserving any standard signals). User applications that may need to be checkpointed must be loaded with the user-level BLCR checkpoint library, which registers a signal handler for the checkpoint signal. Libraries or applications that need to register callbacks and/or use critical sections must #include a BLCR header file and compile their application against the library. Other applications do not need to be recompiled, and can simply be run with the LD_PRELOAD environment variable set to the pathname of the BLCR library, which will cause the library to be loaded in at startup. This LD_PRELOAD can be made part of the standard parallel job launching script ('mpirun', etc.), making it invisible to users.

Given this implementation, a natural first implementation for BLCR was to have the checkpoint signal's handler call any user-registered callback functions at checkpoint time. Critical sections in this approach can simply be implemented by doing the equivalent of a pair of sigprocmask() calls, to temporarily block the checkpoint signal from being delivered. Since doing a system call for each entry/exit is expensive, sigprocmask is not actually used, and instead several words in memory are used to achieve a similar effect. A critical section counter is atomically incremented when a critical section is entered. If a checkpoint signal arrives during the critical section, the signal handler is still run, but if the critical section counter is nonzero, it does not initiate the checkpoint, and instead marks a 'checkpoint pending' bit. The CR_CS_EXIT macro decrements the critical section counter, and when it reaches 0 checks the 'pending checkpoint' bit: if it has been set, a checkpoint is initiated.

One significant drawback to this implementation is that it requires that user-registered checkpoint callbacks be able to run within signal handler context. Unfortunately, the set of standard library functions that are supported with signal handler context is quite limited. Buffered I/O, memory allocation, and many other common tasks are not safe for use. Most importantly for checkpointing purposes, virtually none of the standard synchronization mechanisms (such as pthread mutexes and condition variables, or System V semaphores) are officially safe under POSIX for use in signal context (POSIX declares one such function to be signal safe: sem_post, which unlocks a POSIX semaphore. But since the POSIX semaphore API is not currently implemented under Linux, this is provides little utility) [POSIX ref]. While synchronization within signal context is not impossible (mechanisms such as blocking reads on pipes can be used to achieve effects similar to waiting on a mutex), it is at best quite awkward. As we worked with our collaborators to make the LAM MPI library checkpointable, it quickly became apparent that signal handler context was going to be far too restrictive, and that we'd need to find a solution that allowed a more regular range of standard APIs to be called from within user callback functions.

4.3 Expanded design: thread-based callbacks

To allow freer implementation of user callbacks, we supplemented the signal-based callback mechanism with a thread-based one. A different function is used to register these callbacks, and they are guaranteed to run in a separate thread from the rest of the application (we currently use a single global thread to run all thread-based callbacks, but this is not specified in the interface, so we may later change to run each callback in its own thread to increase concurrency).

Since thread-based callbacks do not run in signal context, they are free to use any standard library calls they need. The application must, however, deal with any synchronization issues that their concurrency may introduce. The regular threads of the application are not stopped by BLCR during the execution of the threaded callbacks (doing so might cause them to be stopped while they held an important C library internal lock, which would then cause deadlock if the callback thread called a function that tries to acquire the same lock). Left to their own devices they will continue normal execution until all threaded callbacks complete, at which time the checkpoint signal is sent to the regular threads in the application. The common solution for any problems introduced by this concurrency is for library authors to place a large granularity 'checkpoint lock' into their code, which is grabbed at every API entry

point (a reader/writer lock can be used if concurrent access by multiple user threads is needed). This lock is obtained (in 'write mode') by the thread-based callback as its first action, and held throughout the checkpoint and restart, until the callback completes. The callback can proceed with shutting down sockets or other uncheckpointable resources, and if the regular application thread tries to enter the library (to send an MPI message, for instance), it will be blocked until the completion of the checkpoint. Blocking the application thread(s) is presumed to be safe at these times, as no internal C library locks should be held when a third party's library calls are entered (it is up to the library/application writer to ensure that their own code does hand out locks to user code in one API call and release them in another, if those locks are needed by any callbacks).

While thread-based callbacks make writing user-level checkpointing code easier, they complicate the implementation of BLCR's critical sections. Originally a critical section was semantically similar to a sigblock, in that it was a thread-local form of protection against an asynchronous, signal-like checkpoint occurring on the current thread. The addition of thread-based callbacks introduces cross-thread locking issues, which must be handled carefully to avoid deadlock (actually livelock, since BLCR critical sections use optimistic, compare-and-swap based locking in order to be usable within signal handlers). The sample code given above in the introduction to critical sections demonstrates this problem. Registration of thread-based callbacks needs to be done atomically with respect to checkpoints: once the registration function returns, any following code must execute with the guarantee that the callback will be invoked before a checkpoint is taken. But since the regular application threads run concurrently with callback threads, and may continue running after they finishes (until the checkpoint signal is delivered to them), there is no way to provide this guarantee without coordination between the threads. A lock can be used for this coordination: it is locked by the registration function to prevent a checkpoint from initiating until the registration is complete, and is locked by the callback thread to ensure that no registrations are allowed to occur once it begins a checkpoint until the checkpoint is complete. But in the example above, if the critical section is entered, then the callback thread is awoken and grabs the lock to commence the checkpoint, the checkpoint will never complete: the callback thread will run its functions and enter the kernel with the registration lock held, but the checkpoint signal that is then sent to the application thread will never be processed (since the application code is in a critical section, waiting for the registration lock, which will never be released). To avoid this situation, entering a critical section must be made equivalent to holding the registration lock. To allow multiple user application threads to enter critical sections at the same time, this lock should have reader/writer-like semantics, except that we may also wish to allow multiple writers, too (if we decide to run threaded callbacks in multiple threads). We thus implement critical sections with what we call "red-black" locks, whose purpose is to ensure that a lock is only held by members of one of two 'colors' of lock holders at any given moment. A counter starting from 0 is used, with 'red' lock attempts incrementing the lock, and 'black' ones decrementing it. Either type of lock attempt will succeed immediately if the counter is set to 0, but red ones will block if the value is already set to a negative number, until the value returns to 0 or becomes positive. Black ones behave in an inverse fashion. To prevent starvation, a bit is used to indicate if a lock attempt of the non-dominant color is pending: if it is set any new, non-nested attempts by the dominant color are denied access. Since nested critical sections will already hold the lock, they must be allowed to proceed. While this solution presents no problem when critical sections are nested properly (the last level of critical section will eventually be exited), starvation may still occur in the asynchronous I/O usage case outlined earlier (if an application always has at least one such I/O operation pending, the callback thread will never get a chance to run). To work around this problem, we may add a interface that users can call that will block if a checkpoint is pending: this will stop the application from issuing new asynchronous requests, so the checkpoint can proceed once those already in progress are completed.

5. The Initial Implementation

BLCR is currently implemented as a Linux kernel module (for recent 2.4 versions of the kernel, such as 2.4.18) and a user-level shared library. So far, it has been possible to implement all needed kernel functionality within kernel module context (one kernel symbol that BLCR needs—do_fork—is not made visible to modules in the official kernel.org sources, but the one-line patch that makes it visible is part of the kernels shipped by most vendors, including Red Hat, which has been our test platform so far). A kernel module implementation has the great benefit that it allows BLCR to be easily deployed by new users without requiring them to patch, recompile, and reboot their

kernel.

Figures 1 and 2 show the high-level steps that occur in the system as a checkpoint and a restart are performed. Time flows from top to bottom in each diagram, and the processes and threads involved in the checkpoint/restart are placed from right to left. Activities performed in the kernel are surrounded by dotted lines. We use the same three-threaded process as the example case for both of these scenarios. This process corresponds to a single-threaded LAM MPI process that has been enabled for checkpointing. Since LAM uses a threaded callback function, a second thread is used, and under LinuxThreads (the standard pthreads implementation for the 2.4 kernel), the creation of any additional threads in an application besides the first causes a 'manager' thread to be created. This manager thread is the child of the first thread in the application, and the parent of all other threads (in this case, just the callback thread).

5.1 Checkpointing an Application

Figure 1 begins with the application fully initialized: the application has registered a threaded callback, which has caused the callback thread to be spawned. This thread blocks in the kernel until a checkpoint occurs. The other two threads (including the pthreads manager) are running normally. At some point it is decided that the application should be checkpointed, and the 'checkpoint' utility that is provided as part of BLCR is invoked. This utility initiates a checkpoint by opening up a special file (/proc/checkpoint/status: this file is created when the checkpoint module is first loaded), then calling ioctl() on it with a structure argument containing the type of checkpoint (in this case, a process), a target identifier (in this case, the process id of one of the threads), and a file handle to which the checkpoint contents should be written (since BLCR does not use seeks during the writing of checkpoints, this file descriptor can point to a socket or pipe). We use the mechanism of ioctls on a special file, rather than a set of new system calls, for two reasons: 1) adding a new system call requires patching the kernel, and is generally discouraged; and 2) using a file allows us to piggyback on the standard reference counting implemented by the kernel for open files, so that if the checkpoint program dies unexpectedly before the checkpoint is completed, the kernel data structures for the checkpoint will still be cleaned up at the appropriate time. The ioctl to initiate the checkpoint is nonblocking: although the 'checkpoint' utility does not take advantage of this, and promptly issues another ioctl that causes it to block until the checkpoint is complete, smarter applications (particularly those that may be managing multiple simultaneous checkpoints and/or other system events) may choose to perform other work while the checkpoint proceeds, and may periodically poll to see if it has completed. In the near future we plan to make this polling interoperate with the standard Unix select() and poll() calls, allowing checkpoint events to be handled the same way as other file descriptor events.

The initial effect of checkpoint's ioctl call is to unblock the callback thread in the application, which upon returning to user-space runs the application's thread-based callback functions. The ioctl also sets up a periodic kernel function, which checks to see if any of the processes/threads involved in the checkpoint have died unexpectedly, or if the checkpoint has gone past its configurable time limit without completing (in which case the checkpoint is cancelled and cleanup up). When the last thread-based callback has entered cr_checkpoint, the callback thread reenters the kernel, and triggers the 2nd stage of the checkpoint. In this 2nd stage, the checkpoint signal is sent to each of the remaining threads in the application. Upon delivery of the signal (or, if the application is within a critical section when it arrives, upon the last exit from all critical sections in a given thread), a BLCR library call is invoked. This routine runs any signal-based callbacks the application has provided, and then enters the kernel (again, via an ioctl call).

Once all the threads in the application have completed their callbacks and entered the kernel, the hardest part of checkpointing—coordination and synchronization—is done. The threads leave the barrier, and one is chosen to write out the header of the checkpoint file, which contains information on the threads and their parent/child relationships. After another barrier the threads takes turns dumping out their state to the checkpoint file. Resources that are shared among the threads (such as memory mappings and file descriptors) are written out only once, by the first thread. The remaining threads then only need to write out their registers, their signal information, and

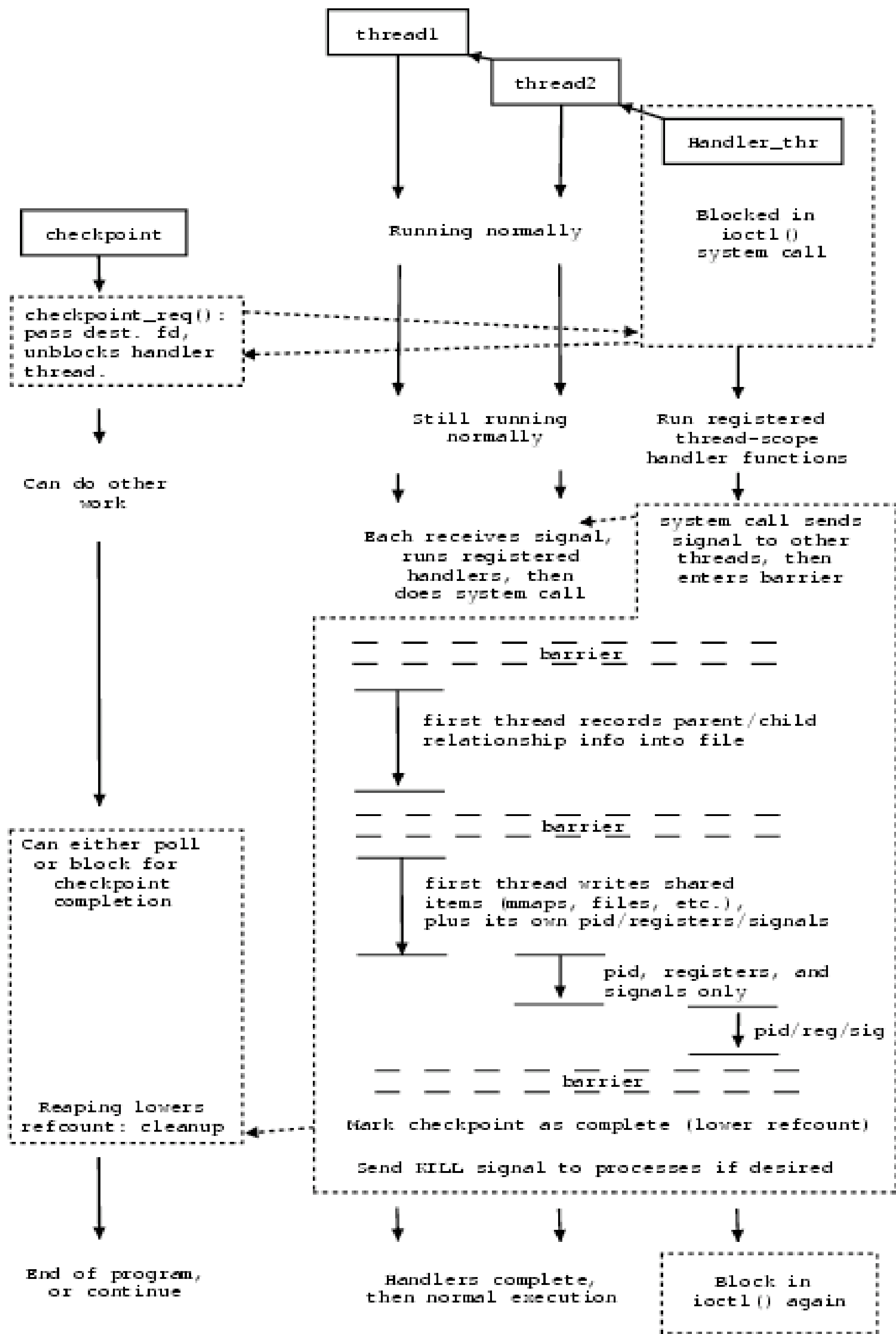

Figure 1: Checkpointing an Application with Three Threads

their pid (the LinuxThreads pthreads implementation is not POSIX-compliant, in that each thread in an application has a different pid). To do this work BLCR largely relies on a pre-existing kernel module called VMADump, which is part of the Beowulf project. This module is designed to serialize the memory regions of a process (and its registers and pid and signal information) to a socket, as part of a remote fork operation that the Beowulf cluster environment provides to users. While it is not designed explicitly for checkpointing, and is also intended for use only with simple newborn processes (single-threaded, without any open file descriptors), its modular design has made it a useful starting point for our own process serialization implementation. We have so far patched the VMADump module to add the barriers, semaphores, and other logic needed for it to serialize multithreaded processes, and plan to further add support in it for files, pipes, and other features. We have retained the modularity of VMADump—it continues to exist as a separate kernel module, upon which the BLCR module depends—and we plan to submit our changes back to the maintainers as a patch.

Once all the threads in a process have completed dumping their state, they enter a final barrier. Upon exiting the barrier, the checkpoint is marked as complete for garbage collection purposes (and any processes waiting or polling for it are notified). If the checkpoint initiator specified that the application was to be terminated after the checkpoint, the job is now killed. Otherwise, the threads all return to userspace, where the restart halves of their callback functions first run to completion, and then the regular application code continues.

5.2 Restarting an Application

As figure 2 shows, restarting an application from a checkpoint file is largely the mirror image of checkpointing one. There are fewer tricky synchronization issues, however, as the kernel has more control over the process from the beginning. The 'restart' utility is provided to resume jobs, and takes the name of a checkpoint file as its main argument. After filling out a structure containing the file descriptor that points to the opened checkpoint file (this can conceptually be either a regular file, a socket, or a pipe, though currently only regular files are supported), the restart utility performs an ioctl call, which results in its being forked. The parent returns to user space, and waits for the restart to complete, while the child is cloned as many times by the kernel as there are threads in the application that is being restarted. The newly cloned threads then perform a 'thaw' using VMADump, taking turns as they read their information from the checkpoint file. One of the threads then uses the data from the checkpoint header to restore the pids of each of the threads, and their process relationships. After a final barrier the processes exit the kernel and enter user space, where their callback functions return from cr_checkpoint and continue until they exit, after which regular application code is resumed.

When the restart is complete, we see the same three-threaded process at the bottom of figure 2 as we saw at the top of figure 1, with one addition: the process now has a new parent. The restart utility interjects itself in between the shell (or other application) that invoked it and the process it restarts. It does nothing more than call wait() and then exit, but its existence is needed so that the parent of restart still sees the same pid for its child, rather than the restored pid of the restarted application (restart is written expressly to handle its children's pids from changing out from under it, while most Unix shells behave badly in this circumstance).

If any of the pids involved in a restart are not available at the time a restart is requested, the restart fails. This detection is done twice in the current implementation: once as soon as the pid information is available (to avoid as much work as possible if the restart is doomed to fail), and again when the kernel's process table is locked during the pid and process relationship restoration phase (to make sure the pids are still available).

Finally, while BLCR currently does not support the checkpointing and restoration of arbitrary file handles, the restart utility arranges for restarted processes to inherit its stdin, stdout, and stderr file handles, so that interactive programs are set to use the terminal of the shell invoking restart. This is sufficient for running test applications, including the NAS Parallel Benchmarks [NPB02], which store all their source benchmark data within their executables' data sections, and write all their results to stdout.

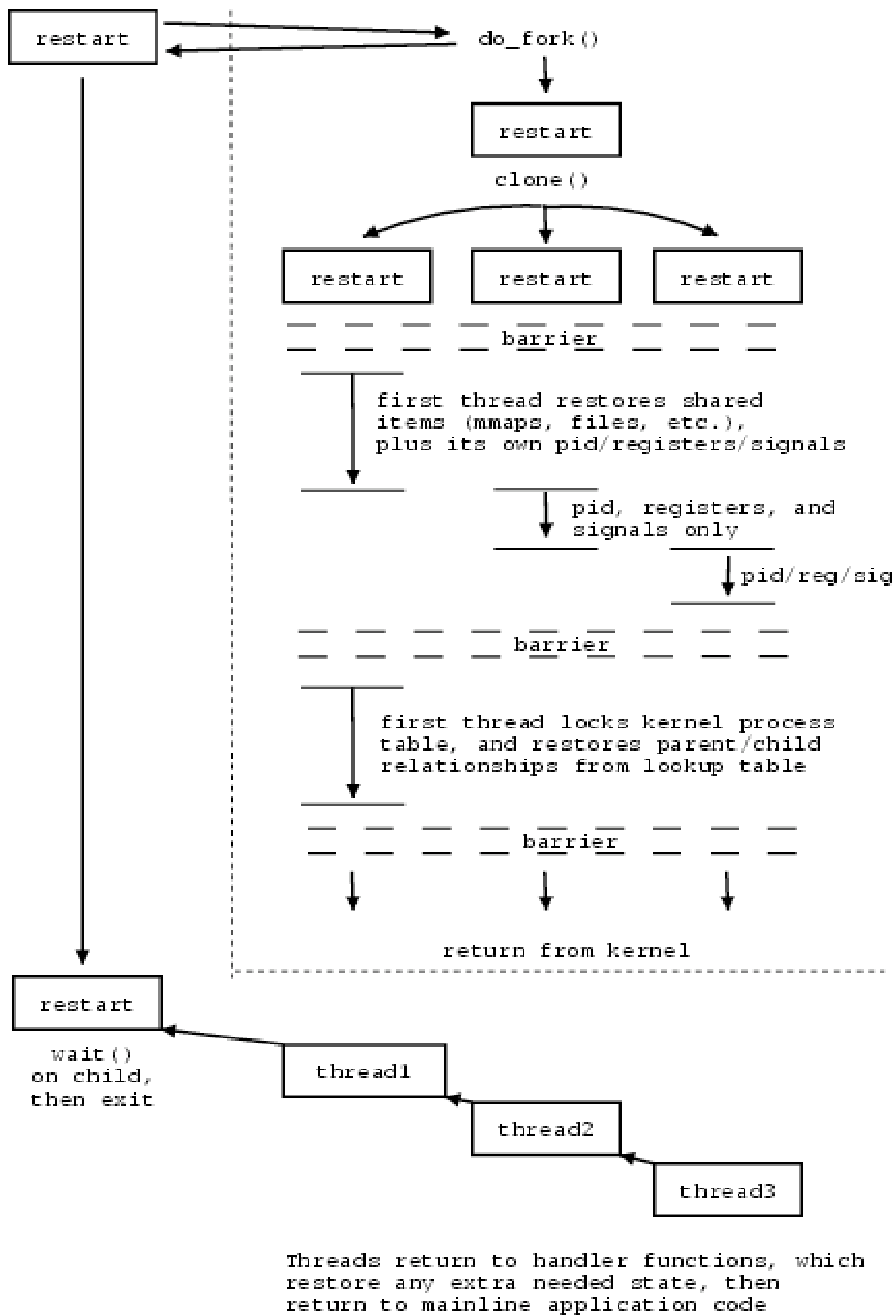

Figure 2: Restarting the Same Application

6. Performance

Figure 3 shows the benchmark times to checkpoint a simple application (with no callbacks) at various sizes. The tests were performed on a 400 MHz Pentium III machine with 128 MB of RAM, using a local IDE disk as the target filesystem for the checkpoint file.

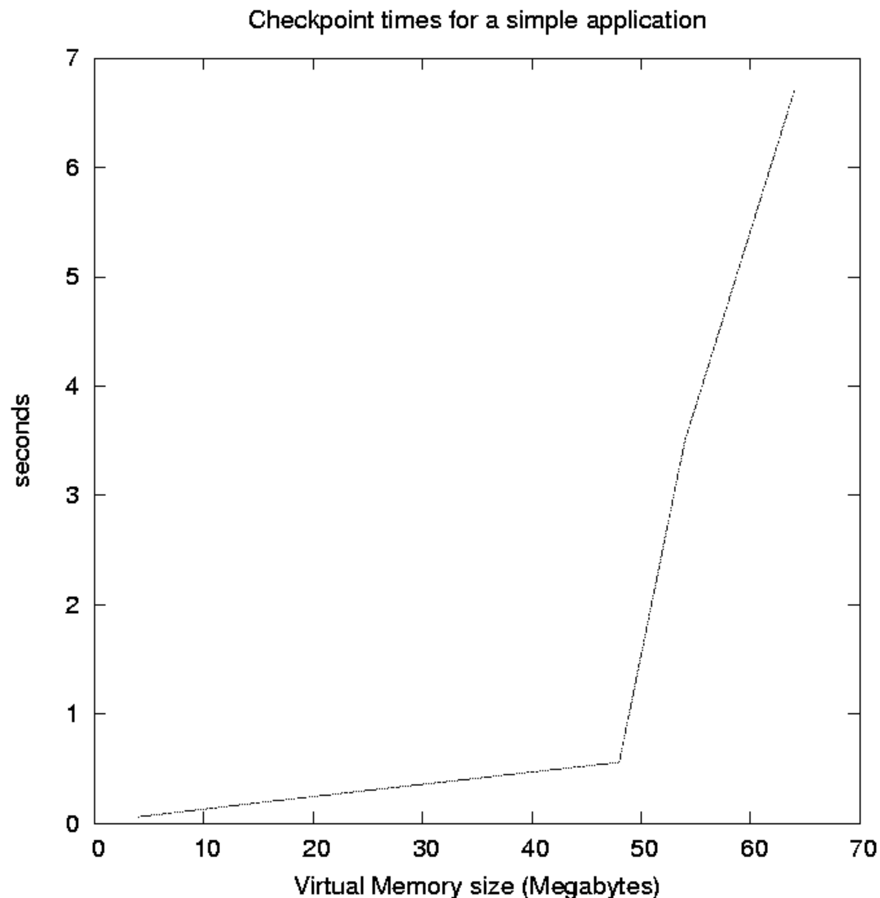

Figure 3: BLCR Performance

As the figure demonstrates, checkpoint times increase linearly with application virtual memory size for processes that consume significantly less than half of the system's physical memory. When the process to be checkpointed has a VM size approaching or exceeding half the physical memory on the system, however, timings go up by an order of magnitude (their variance also becomes considerable). This is due to the fact that VMADump is currently not optimized for serializing large applications, and uses a kernel I/O mechanism that makes copies of each page as it is written. For processes that already consume a significant fraction of the system's physical memory, these copies result in the exhaustion of physical memory, which then causes the kernel's VM system to swap out existing pages to make room for the copies. The fact that the VM system targets the least recently used pages is unfortunate in this case, in that these will typically be the pages that the checkpoint has not yet saved, which means they will just wind up getting paged back in again in order to be saved. BLCR is thus a victim of thrashing in its current form, and will need to be optimized to avoid making copies of pages during checkpoint writing. Given that BLCR guarantees that the memory pages of the application will not be modified during the checkpoint, there is no inherent need for such copies.

7. Future Work

BLCR is still in the early phases of implementation. While we are happy to have simple MPI applications already working, and to have developed the interfaces for both signal- and thread-based callbacks, we still have a great deal to do before BLCR is ready for production use. Our first priority is to finalize the interfaces that BLCR presents to libraries and applications, so that more Linux cluster projects can be encouraged to begin integrating BLCR's checkpoint functionality into their systems. We then need to implement these interfaces, and the list of planned items in Table 1. Finally, we can focus on optimizing the implementation and implement optional extenstions.

7.1 File handles.

Finalizing and implementing the semantics for handling applications' files across checkpoint/restart is the most important interface issue remaining for BLCR. Unfortunately, there is no one most desirable behavior for handling an application's files. We have identified the following behaviors as each having a use in some contexts:

- **Simple reopen**. In this model, no information about the file is saved in the checkpoint, except for the name of the file, and the current file offset at the time of the checkpoint. At restart time, the file is expected to exist in the same filesystem location, and it is reopened and seeked to the checkpoint-time file offset.
- **Checksum and reopen**. This model is identical to simple reopen, except that a checksum of the file is also stored at checkpoint time. The checksum is reapplied to the file at restart time, and if the file has changed, the restart fails.
- **Truncate**. This mode is really a modification that can be applied to the first two approaches. The size of the file is recorded at checkpoint time, and at restart the file is truncated to its checkpoint-time length. In the checksum case, this is done before the checksum is reapplied. Truncation provides a primitive rollback mechanism for files that are written to in a continuous stream, which is the access pattern used by many scientific applications. It allows such applications to be restarted in an idempotent fashion multiple times and still get a coherent output file.
- **Backup and restore**. In this model, a full copy of the file is made at checkpoint time, and stored as part of the checkpoint. When the application is restarted, the stored copy is restored onto the filesystem, overwriting any existing version.
- **Backup and anonymize**. As with the previous model, a copy of the file is made at checkpoint and restored at restart, but the restored file is only visible to the restarted application. This mode may be useful for handling a fairly common idiom in Unix programming, in which a file is opened by an application then deleted from the filesystem, remaining visible only to the application and existing only while the application keeps its file handle open.

In general, these semantics provide a trade-off between efficiency and the robustness of their guarantee that a restarted program will find its files in the state it left them. Backup is clearly superior for restoring files to the correct state, but is much more expensive than the reopen approaches. It may be possible to mitigate the cost of backup mode by using a filesystem that supports copy-on-write (or diff-based) 'snapshots', which could provide more efficiently provide backup guarantees. These snapshops could be reference counted by checkpoint files, so that the deletion of a checkpoint would result in the cleanup of any snapshots particular to it.

Besides determining a useful set of file handling semantics, BLCR will also need to come up with a sensible set of mechanisms for communicating the preferred semantic for a given file or application. This information should probably be settable on a per-file basis, and it may be desirable to allow it to be set in various places. A user may explicitly specify behaviors when he or she starts the program or submits it to the batch queue; a system administrator may wish to specify the semantics based on information they control. And a checkpoint-aware library may use its own files, and wish to specify how they are handled via API calls to the BLCR interface.

7.2 Potential Optimizations

There are a number of optimizations that may make sense to implement in BLCR. Some of these may reduce the impact of taking a checkpoint to the extent that it becomes practival to use BLCR for periodic checkpointing.

- **Optimizing checkpoint file I/O**. As the performance section of this paper clearly shows, there is room for improvement in BLCR's handling of checkpoint file I/O, especially for large applications. A strategy that avoids making copies of the application's pages during writing is clearly a necessity. Also, vmadump currently writes each page of memory to disk with a separate write() call. Disk access patterns may be improved by writing out chunks of contiguous physical pages, and/or using vector-like mechanisms for writing out groups of pages that are not contiguous, and this could result in considerable throughput gains. It may also be possible to implement a checkpoint-specific filesystem that takes advantage of the predictable qualities of checkpoint file access (such as the fact that the total size of a checkpoint file can be known before any data is written) to improve disk throughput and sector allocation. Clusters whose compute nodes have no local disk (such as blade-based systems), and share access to a smaller number of I/O nodes may benefit from an approach where incoming checkpoint file blocks from multiple nodes that are part of a single parallel job are 'shotgunned' together into a single interleaved file (which can then be written with a fast, write-ahead log style disk access pattern), which can then be read out and multiplexed out in reverse at restart time.
- **Fork & copy-on-write**. BLCR currently blocks an application from executing during the entire time that the checkpoint file is being written. This is undesirable for periodic checkpoints. One possible optimization for this would be to use a strategy that did the equivalent of a fork on checkpointed processes after their callbacks have all reached their cr_checkpoint calls. One side of the fork would continue executing as if the checkpoint I/O were already completed, while the other would retain (via fork's copy-on-write behavior for sharing pages between parents and children) a pristine copy of the program at checkpoint time, which could then to written out in the background. Several challenges would need to be met, however. First, use of the actual fork() system call would not be suitable for multithreaded processes (POSIX specifies that a fork call by a thread in a multithreaded program results in a child that has only a single thread—a copy of the one that called fork—which is obviously not the desired state required for a checkpoint). Second, since many scientific applications use virtually all the physical memory on the machine, the copy-on-write mechanism would have to be carefully monitored, to avoid creating so many copies that physical memory is exceeded and paging behavior triggered. To avoid this, the side of the fork that does the checkpoint I/O might need to track the behavior of the continuing side, and give priority to writing pages that have been modified by the continuing execution, so the original pages can be discarded. Such a tracking mechanism could be implemented by piggybacking on the standard copy-on-write mechanism, which uses protection faults to know when to make a copy of a shared page: the page could be marked as high-priority for the checkpoint at the same time.
- **Allowing user applications to specify regions that don't need saving**. As previously mentioned, many large scientific applications already do some form of application-specific checkpointing, and these checkpoints can leverage application-specific knowledge to save smaller amounts of state. BLCR could be made to interoperate with such schemes by providing an API that allows a user application to demarcate regions of its memory that do not need to be checkpointed by BLCR. Certain parallel languages (such as Unified Parallel C [REF], Titanium [REF], or Co-Array FORTRAN [REF]) might be able to determine such regions automatically at runtime.

7.3 Allowing signal-free use

Currently BLCR reserves a real time signal number to perform checkpoints, and requires that checkpointable applications load in a shared library that registers the signal handler for that signal. While these are not likely to be significant intrusions in a Linux cluster environment (where real-time signals are seldom used, and where parallel jobs are usually compiler and run with scripts that can take care of loading the BLCR shared library), it is possible to eliminate them by modifying the kernel to allow checkpoints to use a signal-like mechanism that does not use

any of the standard Unix signal numbers. This mechanism could also directly insert the appropriate ioctl call onto an application's signal stack if the application did not require any callback functions, avoiding the need to load any BLCR shared libraries. However, it appears unlikely that this modification could be done completely within kernel module context, and so BLCR might have to supply a kernel patch that would need to be applied to use this method.

8. Related Work

A number of existing open source checkpoint implementations for Linux clusters are available. The majority of them are implemented as user-level libraries, and thus have the limitations that were earlier described. Many of them, however, implemented interesting ideas from which BLCR has borrowed liberally. Libckpt [PLAN95] was one of the first open source checkpoint libraries available. While it only supported single-threaded processes, it introduced the 'fork and continue' concept we describe in our optimizations section. It also contained support for incremental checkpoints, in which only pages modified since the last checkpoint are saved, and for user application to mark regions of memory that do not need to be saved. Condor [LITZ97], a loosely coupled network of workstations system, implements its own checkpoint/restart mechanism, and uses the approach we follow of using dynamic loading of a checkpoint library to avoid requiring all checkpointable applications to be recompiled. CoCheck [PRUY96] implements checkpoint/restart on top of the Condor system for PVM and MPI jobs, using markers piggybacked on top of network messages. Libckp [WANG95] is one of the few libraries primarily focused on fault tolerance. It has a rich set of behaviors for handling file descriptors (including supporting opened and deleted files, as in our 'copy and anonymize' semantic), and also provides a setjump/longjump-like mechanism for allowing application code to roll itself back to different execution points. Libtckpt [DIET01] supports checkpointing multithreaded that use either LinuxThreads of Solaris threads. Like BLCR, it uses a separate checkpoint thread, and users can register handler functions that can be called at checkpoint, restart, and continue points.

Several kernel-level checkpoint and/or process migration systems are also available for Linux. Beowulf's VMADump has already been described. The Mosix single system image version of Linux [REF] implements a limited process migration scheme (jobs remain reliant on the continued operation of their original home node, so migrations geared toward avoiding imminent failures of nodes are not supported). The HP Single System Image project for Linux [REF] plans a more comprehensive and fault-tolerant approach to migration, but this migration requires tight kernel-level integration at the cluster level, and does not support checkpointing per se. Finally, the CRAK (Checkpoint/Restart As a Kernel module) project [ZHON01] provides a kernel implementation of checkpoint/restart for Linux. CRAK inspired us to implement our own checkpoint logic as a kernel module rather than a patch, but certain aspects of its approach proved troublesome for our purposes (it saves applications to disk directly from the process that initiates the checkpoint, which makes it difficult to allow user hooks, and its checkpoint writing/reading code uses seeks extensively, making it difficult for use in process migration), causing us to ultimately use VMADump as our starting point instead. Recent versions of CRAK support checkpointing TCP sockets.

In the context of this related work, the NLCR project's contribution can be seen as mainly being a matter of taking some of the better ideas from user-level approaches, and combining them with a kernel-level implementation. In this way we hope to be able to provide a flexible base component for Linux cluster checkpoint/restart, while still providing the most comprehensive support for items that only a kernel-level checkpoint system can provide. We also hope to be able to take advantage of BLCR's kernel-level context to implement optimizations that are impossible to do from a user-level checkpoint implementation. As individual jobs on cluster nodes routinely grow into the gigabytes in size, and overall parallel applications take up terabytes of memory, these optimizations may prove critical to making checkpoint/restart a practical reality on large production Linux clusters.

9. References

[CAF02] Co-Array Fortran home page. Located at http://www.co-array.org/.

[CARO02] Christopher D. Carothers and Boleslow K. Szymanski. Checkpointing Multithreaded Programs. Dr. Dobb's Journal. August 2002.

[DIET01] William R. Dieter, and James E. Lumpp, Jr.. User-level Checkpointing for LinuxThreads Programs. FREENIX Track: USENIX 2001 Annual Technical. Conference. pp. 81-92. June, 2001

[HEND00] Erik Hendriks. VMADump. http://bproc.sourceforge.net. . 2002

[HPQ02]. The HP Single System Image Clusters Project for Linux. Homepage at http://sourceforge.net/projects/ssic-linux.

[IBM02] The IBM Blue Gene Project. Information available at http://www.research.ibm.com/bluegene/.

[PLAN95] J. S. Plank, M. Beck, G. Kingsley, and K. Li.. Libckpt: Transparent Checkpointing Under UNIX. Conference Proceedings, Usenix Winter 1995. Technical Conference, pages 213-223. January 1995.

[LITZ97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed System. http://www.cs.wisc.edu/condor/doc/ckpt97.ps. .

[MOXI02] The MOSIX project home page. Located at http://www.mosix.org/.

[NBP02] The NAS Parallel Benchmarks. Information available at http://www.nas.nasa.gov/Software/NPB/.

[PRUY96] Jim Pruyne and Miron Livny. Managing Checkpoints for Parallel Programs. Job Scheduling Strategies for Parallel Processing. IPPS 96 Workshop v. 1162 pp. 140-154. 1996

[SCOR00] T. Takahashi, S. Sumimoto, A. Hori, H. Harada and Y. Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network. Environments. http://www.sc2000.org/techpapr/papers/pap.pap205. .pdf. 2000

[STEL96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. Proceedings of the 10th International Parallel. Processing Consortium (IPPS 96). 1996

[UPC99] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, Karen Warren. Introduction to UPC and Language Specification. IDA Center for Computing Sciences, 1999. Available at http://www.gwu.edu/~upc/pubs.html.

[WANG95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. Proceedings of the International Symposium on. Fault-Tolerant Computing. pages 22-31, June 1995.

[WONG99] Adrian T. Wong, Leonid Oliker, William T. C. Kramer, Teresa L. Kaltz and David H. Bailey. System Utilization Benchmark on the Cray T3E and IBM SP. Fifth Workshop on Job Scheduling, May 1999; LBNL-45141.

[YELI98] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. Concurrency: Practice and Experience, Vol. 10, No. 11-13, September-November 1998.

[ZHON01] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint / Restart As a Kernel Module. Technical Report CUCS-014-01. Department of Computer Science. Columbia University, November 2002