# Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters

**Paul H Hargrove[1] and Jason C Duell[1]**

Computational Research Division, Ernest Orlando Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA

Project E-mail: checkpoint@lbl.gov

**Abstract**. This article describes the motivation, design and implementation of Berkeley Lab Checkpoint/Restart (BLCR), a system-level checkpoint/restart implementation for Linux clusters that targets the space of typical High Performance Computing applications, including MPI. Application-level solutions, including both checkpointing and fault-tolerant algorithms, are recognized as more time and space efficient than system-level checkpoints, which cannot make use of any application-specific knowledge. However, system-level checkpointing allows for preemption, making it suitable for responding to “fault precursors” (for instance, elevated error rates from ECC memory or network CRCs, or elevated temperature from sensors). Preemption can also increase the efficiency of batch scheduling; for instance reducing idle cycles (by allowing for shutdown without any queue draining period or reallocation of resources to eliminate idle nodes when better fitting jobs are queued), and reducing the average queued time (by limiting large jobs to running during off-peak hours, without the need to limit the length of such jobs). Each of these potential uses makes BLCR a valuable tool for efficient resource management in Linux clusters.

## 1. Introduction

Berkeley Lab Checkpoint/Restart (BLCR) [1] is a part of the Scalable Systems Software Suite [2], developed by the Future Technologies Group at Lawrence Berkeley National Lab under SciDAC funding from the United States Department of Energy. It is an Open Source, system-level checkpointer designed with High Performance Computing (HPC) applications in mind: in particular CPU and memory intensive batch-scheduled MPI jobs. BLCR is implemented as a GPL-licensed loadable kernel module for Linux 2.4.x and 2.6.x kernels on the x86 and x86-64 architectures, and a small LGPL-licensed library. In section 2 we describe the motivation for writing BLCR and give some of the design requirements that come from the HPC focus. In section 3 we describe the major design features of BLCR that are dictated by these requirements. Section 4 describes some of the unique implementation strategies used in BLCR; some of which may be of more general use.

## 2. Motivation

### 2.1. System-level vs. application-level checkpointing

[1] Work supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

When available, application-level fault-tolerance solutions are typically the preferred approach to fault-tolerance in HPC. The most popular approach is periodic checkpoints taken at the application-level. This approach is able to use application-specific knowledge to minimize the required I/O volume, making it more time and space efficient than a typical system-level checkpoint taken periodically at the same interval. The use of system-level checkpoints, however, can offer benefits in an HPC environment, which application-level checkpoints cannot. The preemptive nature of system-level checkpoints allow for additional freedom in batch scheduling which can increase utilization while simultaneously reducing average queue wait time. Additionally, preemptive checkpointing can be used for fault-tolerance in response to "failure precursors" rather than for periodic checkpoints. As examples of precursors to failures that might be seen in a cluster running Linux, consider observation of elevated temperatures, of unusually high correctable ECC memory error rates, or of abnormal network CRC failures. It is these scheduling and fault-tolerance benefits, combined with the prevalence of Linux clusters in HPC that motivated the choice to pursue a system-level checkpoint/restart implementation for Linux clusters.

### 2.2. HPC-focused requirements

While not all specific to the HPC environment, we have identified [3] the following requirements, among others:

- Preemptive – not limited to a specific point in the application source code.
- Transparent – should not require source-level modifications to the majority of HPC applications. Recompilation, relinking and use of specialized program launchers are acceptable, because HPC users are accustomed to these differences as they move from one system to another.
- Performance transparent – should not significantly increase the length of runs in which no checkpoints are taken. Zero impact is preferable.
- Wide application coverage – should provide a restartable checkpoint for the largest possible range of HPC applications possible, subject to the resources available to implement and test BLCR.
- Wide system coverage – should not be specific to any one version of the Linux kernel, C compiler, C library, etc.
- Security conscious – should not allow privilege escalation or escape from accounting.

A survey [4] of Linux checkpoint implementations available at the time our work began on BLCR found no implementation that met all our needs. However CRAK [5] and the VMADump component of Bproc [6] were determined to be the closest.

## 3. Design

The requirements for preemption and transparency dictated to us that BLCR either work in the Linux kernel (as a patch or loadable module) or in user-space as a wrapper around the standard C library. The wide system coverage requirement eliminated the option of a kernel patch due to the frequent changes to the kernel source. Wrappers around the C library (or immediately "under" it at the kernel system call interface [7]) were discarded due to concerns over the performance cost of the associated interception or virtualization. We chose, therefore, to design BLCR as a kernel module. This choice was not without associated challenges, some of which are described in section 4. However, the choice to implement in the kernel does allow access to data that, from user-space, are either impossible to modify (e.g. pid) or very difficult to access without intercepting library calls to mirror kernel data structures (e.g. filename of an open file).

It was determined early on that we did not have the resources to write a checkpointable implementation of sockets, nor any interest in supporting TCP/IP other than as required to support MPI and/or other HPC libraries. Therefore, we chose to design an extension "callback" interface (described in [8]) to allow any library or application code to cooperate in the taking of and restoring from a checkpoint. This interface allows MPI implementations (or other parallel programming environments) to perform actions needed to support distributed jobs.

As an example, LAM/MPI 7.x provides integration with BLCR via the callback interface [9]. The approach taken by LAM/MPI uses the mpirun process as a master, or representative, process. When checkpointing mpirun, its callback requests a checkpoint of all the application processes before the checkpoint of mpirun is permitted to complete – providing the global knowledge of the job that BLCR lacks. Within the application, a checkpoint callback in the MPI library uses a simple sync-and-stop mechanism to drain the network of in-flight communication at checkpoint time and to establish new sockets at restart time – providing the communications support that BLCR does not provide (and at much lower complexity than a fully general solution).

## 4. Implementation

In implementing BLCR there were a large number of challenges to overcome. The remainder of this section describes some of the challenges and the solutions we adopted.

### 4.1. Extending VMADump

Based on our survey of the prior work [4], it was decided to begin with the VMADump kernel module and modify and extend it as needed to meet our needs. Primarily for maintenance reasons, we elected to keep a distinct “vmadump” kernel module and make calls to it from the “blcr” kernel module. The main challenge in adapting VMADump to our needs was that while SMP-safe, VMADump was not equipped to deal with multi-threaded processes. However, we felt that support for applications using POSIX threads (pthreads) was required because hybrid MPI/pthreads applications are an effective way to utilize SMP compute nodes.

The solution adopted was to make use of the existing integer flags argument to VMADump’s “freeze” and “thaw” routines. We added a flag bit indicating that we wish to freeze or thaw only the thread-specific portions of a given task: the registers, signal mask and the thread-local-storage pointers (used internally by the implementation of pthreads). Since VMADump was already equipped to read or write an arbitrary file descriptor, it became a simple matter to write/read a BLCR-defined header to a descriptor, then invoke VMADump on the same descriptor without the thread-specific flag (to save/restore the first thread and the process-wide memory map and signal handlers), followed by zero or more calls to VMADump *with* the new flag (to handle the thread-specific portions of the remaining threads, if any). This is followed by more BLCR-defined data – in particular the file table, which VMADump does not handle.

### 4.2. Preemption without a kernel patch

The VMADump code, as well as many of the kernel functions on which it and BLCR rely, implicitly assume that they are operating on the “current” thread – with only a few exceptions the Linux kernel generally provides no mechanism to read or modify resources associated with another process. For this reason the preemption of a process to be checkpointed requires not only stopping all its threads (to prevent modification from racing with the checkpoint) but also forcing each thread to invoke a specific system call to allow checkpointing of its own resources.

In addition to the kernel module, which is responsible for the majority of the checkpoint and restart activity, BLCR is designed with a small shared library that contains the implementation of BLCR’s extension interface. Since this library is already a part of the BLCR architecture, it was leveraged to combine the preemption activity with the management of callbacks, and we require that to be checkpointable a process must load this library. For applications that use the callback interface (e.g. LAM/MPI applications) the library must already be linked explicitly to satisfy this dependence (mpicc handles this for LAM/MPI applications). For applications that have not explicitly linked the BLCR shared library, we provide a “cr_run” command (used as a prefix like “nice”, “time” or “nohup”), which utilizes the LD_PRELOAD environment variable to load in the BLCR shared library[2].

Any process loading the BLCR shared library will automatically run that library’s initialization function, which registers a signal handler for the lowest-priority real-time signal number. When the kernel, in response to a checkpoint request, delivers the corresponding signal[3], this handler runs any callbacks registered though BLCR’s extension interface and handles the implementation of critical sections in the interface (allowing atomicity of certain operations with respect to checkpoints). Once all registered callbacks are run, the signal handler makes the required system call to BLCR to allow the checkpoint to proceed.

## 4.3. The Linux module interface

The Linux kernel’s interface to loadable modules provides less than the full symbol table used to link the general kernel code. The selection of exported functions and data is well chosen to keep the interface as narrow as possible while supporting loading of device drivers, file system formats and executable formats. The primary reason for keeping a narrow interface is maintenance; allowing loadable kernel modules developed and tested separately from the kernel to use only explicitly exported symbols insulates such modules from changes in the kernel. However, to perform its task, BLCR need access to both functions and data that are *not* exported to modules.

Our solution is to require that at build time the files System.map or vmlinux be available. From either of these files we can obtain the absolute addresses of the symbols we need. These addresses are then hard-coded into a separate “blcr_imports” kernel modules, which exists only to provide a symbol table with these addresses. There are initialization-time safety checks to ensure that the addresses obtained in this manner for a few exported symbols match, thus protecting against possible built-time mismatch between the kernel source and symbol information.

## 4.4. The Linux kernel as a moving target

By far the most challenging aspect of implementing BLCR was to keep it working as the Linux kernel continued to evolve. This was mostly a result of our need to work outside the interface exported to kernel modules. We have been successful in doing so, and BLCR works with all 2.4.x and 2.6.x kernels available from kernel.org at the time of this writing, plus nearly all the 2.4 and 2.6 based distribution-specific kernels we have encountered. Had BLCR been implemented as a kernel patch, this wide coverage of kernel versions would have required *many* different versions of the patch. The kernel for Red Hat Linux 9, which BLCR *does* support, would have required a specialized patch as well because it contains portions of 2.6.x kernel code back-ported to a 2.4.x kernel. Furthermore, various other distribution-specific patches could have potentially caused patch failures due to textual conflicts where no semantic conflict existed.

Our solution uses GNU autoconf to probe the configured kernel sources (which we require be present at BLCR build time) for specific features such as presence/absence/type of structure members; presence/absence of functions and macros; and number and type of arguments to functions. The use of “#ifdef” then permits us to maintain a single BLCR source tree. This was somewhat complicated by the fact that unlike most applications of autoconf, we can only compile test cases but not link or run them. This is even more restrictive than a cross-compilation environment where linking is still available. However, under the assumption that prototypes/declarations are not present for non-existent symbols, using “sizeof(some_symbol)” was sufficient to implement existence checks that might otherwise be performed at link time.

[2] An unimplemented alternative to the LD_PRELOAD would be to have the kernel forcibly map the library (or just a small subset of it) into the address space of a process for which a checkpoint request has been received, but that has not yet loaded it

[3] Since it is running in the kernel, BLCR bypasses the signal mask (preventing applications from blocking this signal) and also ensures delivery to every thread of a multi-threaded process.

### 4.5. The Linux "kbuild" tool

The GNU automake tool is a popular way to manage building Open Source software. Together with GNU autoconf and libtool, automake is an important part of BLCR. However, the Linux kernel has its own "kbuild" tool filling a role similar to automake plus providing libtool-like functionality for building kernel modules. While use of the kbuild infrastructure was mostly optional with 2.4.x kernels, it is very difficult to get a loadable kernel module built correctly for a Linux 2.6.x kernel without using kbuild.

To deal with this, we have written a Makefile fragment for inclusion into a Makefile.in that transparently allows kbuild to manage chosen subdirectories within a project managed by automake.

### 4.6. Enforcing security

Checkpointing a process involves writing of process memory and some kernel data structures to a file, while restarting involves reconstructing processes from the contents of a file. We needed to ensure that these operations could not compromise system security. This need is expressed in the following two requirements:

1. A checkpoint must not expose any data (user or kernel) not otherwise visible to a user.
2. A restarted process must not be granted any resources or permissions not otherwise available to the process owner, even in the presence of malicious modification to checkpoint files.

The first requirement was met by adopting the same checks that are used in the Linux kernel when exposing data via "/proc/<pid>" or in core files. Specifically a checkpoint may only be requested by the process owner, or by root (or a user with the equivalent "capability" setting). Additionally, a checkpoint may not be taken of a setuid or setgid program, except by root. This rule prevents exposing sensitive data such as the contents of "/etc/shadow" that may exist in process memory.

To meet the second requirement, we took the approach of creating resources at restart time via the same paths that system calls from the process would use to obtain them. This ensures that the normal permission checks are applied, preventing the restarted processes from obtaining resources that could not be obtained by the requesting process (which runs as the original process owner). For instance to reopen a file BLCR calls the same kernel code as for an open() system call, ensuring the proper permission and resource limit checks are applied.

The enforcement of resource limits is a special case, because they may be under the control of the batch system. For this reason BLCR does not save or restore the resource limits, but instead inherits them from the requesting process, allowing the batch system to assign different limits to the restarted job than to the original job. This approach also eliminates the possibility of intentional "credit fraud" modifications to a checkpoint file, as there is nothing present to modify.

## 5. Conclusion

This article has described some of the design and implementation choices made in building BLCR. It is our hope that this information will prove valuable to others wishing to perform similar work within the Linux kernel.

**Acknowledgements**

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## References

[1] Berkeley Lab Checkpoint/Restart Homepage. http://ftg.lbl.gov/checkpoint

[2] Geist A *et al* 2001 Scalable Systems Software Enabling Technology Center. http://www.scidac.org/ScalableSystems

Bode B *et al* 2005 Scalable system software: a component-based approach *J. Phys.: Conf. Ser.* **16** 546-50

[3] Duell J, Hargrove P and Roman E 2002 Requirements for Linux Checkpoint/Restart *Lawrence*

*Berkeley National Laboratory Technical Report LBNL-49659*

[4] Roman E 2003 A Survey of Checkpoint/Restart Implementations Lawrence *Berkeley National Laboratory Technical Report LBNL-54942*

[5] Zhong H and Nieh J 2002 CRAK: Linux Checkpoint/Restart as a Kernel Module *Columbia University Department of Computer Science Technical Report CUCS-014-01*

[6] Bproc: Beowulf Distributed Process Space. http://bproc.sourceforge.net/

[7] Osman S, Subhraveti D, Su G, and Nieh J 2002 The Design and Implementation of ZAP: A System for Migrating Computing Environments *Proc. 5th Symp. on Operating Systems Design and Implementation (Boston, Mass, USA, December 2002)*

[8] Duell J, Hargrove P and Roman E 2002 The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart *Lawrence Berkeley National Laboratory Technical Report LBNL-54941*

[9] Sankaran S, Squyres J M, Barrett B, Lumsdaine A, Duell J, Hargrove P and Roman E 2003 The LAM/MPI checkpoint/restart framework: system-initiated checkpointing *Proc. Los Alamos Computer Science Institute (LACSI) Symp. (Santa Fe, New Mexico, USA, October 2003)*

Also appearing in 2005 Intl. J. High Performance Computing Applications **19** 479-93