Incremental Checkpointing for Grids

John Mehnert-Spahn, Eugen Feller, Michael Schoettner
Heinrich Heine University of Duesseldorf, Duesseldorf, NRW, Germany
{John.Mehnert-Spahn, Eugen.Feller, Michael.Schoettner}@uni-duesseldorf.de

Abstract

The EU-funded project XtreemOS implements an open-source Linux-based grid operating system. Here, checkpointing (CP) is used to implement fault tolerance and process migration. We have developed an incremental CP solution for saving only memory pages that have been changed since the last CP. We present how we keep track of memory page modifications between CPs using Linux-native radix trees and how we handle virtual memory area changes. We also discuss experiment results with selected examples. Finally, we present a custom memory event connector transparently reporting page write-protection fault of processes to a user mode grid service to adaptively control incremental CP at grid level.

1 Introduction

Secure and efficient resource sharing between institutes and companies is increasingly required by research, engineering and industry. Both is provided by grid technologies implementing distributed computing platforms e.g. middleware-approaches like Globus [4] or grid-functionality integrated into native operating systemes such as XtreemOS [3].
Grid-inherent dynamicity as well as the unpredictable application behaviour require to transparently move applications from heavily-loaded or close-to-fail grid nodes to idle and healthy ones. Coping with grid fault tolerance is an ongoing research topic. Our approach is based on using existing kernel checkpointers, including the latest Linux checkpointer. In this paper we focus on developing incremental checkpointing in Linux to speed-up checkpoint time. Once, a grid service is able to switch between full and incremental checkpointing, based on monitoring information provided by the kernel, the best-suited strategy can be applied.
Recent Linux innovations allow applications, running on one node, to be isolated from each other regarding resources such as cpus, pids, network, ipcs, file system, etc. Container concept [12] implementations such as cgroups [1] allow the same resource identifier to be used by multiple applications residing in a separate cgroup container. The container concept pushes server consolidation. Furthermore, it is significant for fault tolerance to avoid potential resource conflicts at restart. A checkpointing mechanism is about to be developed and can be used for process migration and fault tolerance. Linux container and checkpointer implementations are required by grid computing in order to support load balancing and fault tolerance.
This paper is structured as follows: section 2 provides a short overview of the XtreemOS project, section 3 gives a detailed insight into our concepts and implementation of incremental checkpointing in Linux including a short overview of related work, section 4 presents evaluation results followed by conclusions and outlook.

2 XtreemOS

The EU funded project XtreemOS aims at providing a Linux-based open source Grid operating system coming in three flavours for: single PCs, Single System Images (SSI) cluster and mobile devices. Fundamental XtreemOS functionalities include native Virtual Organisation (VO) support and fault tolerance that have been implemented by extending the native operating system. Basically, applications can transparently exploit resources distributed in the grid spread across administrative domains. Besides state-of-the-art grid applications, legacy applications can be run in the grid unmodified by relying on the POSIX interface provided by XtreemOS. Developers can easily create new grid applications, using the functionality provided by distributed grid services through the XOSAGA API providing access to security, resource and process management. Users are organised in VOs, VO policies can be created to provide fine grained resource access control. Furthermore, XtreemOS comes with a grid file system

called XtreemFS [10], allowing for location transparent file access, file replication and file striping.
XtreemOS ships with an integrated grid checkpointing mechanism. The components of the XtreemOS grid checkpointing architecture [13] are shown in figure 1. The main concept is to rely on existing kernel checkpointer packages, e.g. Berkeley Labs Checkpoint Restart (BLCR) [8], OpenVZ[11], the Linux-native checkpointer for a single PC[1] and the LinuxSSI checkpointer [14] for a SSI cluster.

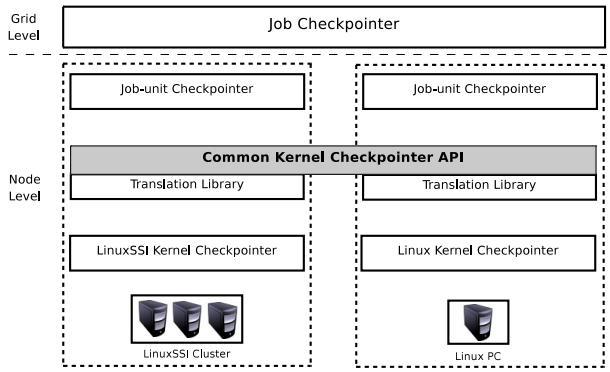

Figure 1: XtreemOS grid checkpointing architecture.

The job checkpointer service at the top is responsible for checkpointing/restarting a job consisting of one or multiple job-units. A job-unit checkpointer focusses on saving and restoring a single job-unit. Therefore, it uniformly addresses an underlying kernel checkpointer using the so-called *common kernel checkpointer API*. This API is implemented per checkpointer in a separate translation library. It bridges semantic differences, e.g. each kernel checkpointer distinguishs from another one by an individual calling semantic. Furthermore, there are different process group types supported by the checkpointers. The translation library must check a process group precisely matches a job-unit's processes in order to enforce checkpoint/restart consistency. The library is responsible for providing a uniform interface to developers to transparently register checkpoint/restart callbacks. The API supports the retrieval of a matching kernel checkpointer for an application at job submission, because most kernel checkpointers are incapable of saving and restoring all possible kernel resources. Besides grid-to-kernel level semantic translations, the API also provides inter-kernel checkpointer translation regarding saving/restoring reliable channels by a generic channel flushing protocol.

3 Incremental Checkpointing in Linux

3.1 Memory page modification detection

Checkpointing overhead can be dramatically reduced by saving only content that has changed since the previous checkpoint. The major challenge here is to *detect* these content modifications. Generally, there are page-based and variable based approaches.
Detection of content changes at variable-granularity-level is described in [7] where a compiler is manually modified to detect variable changes. Thus, the compiler is enabled to insert incremental state saving calls before a variable is changed. In [15] detection of changes variables is enabled without manual intervention, but via an executable editing library. Furthermore, special memory hardware exists that incrementally state saves contained variables [5].
Detection of modified pages can be realised by taking existing page table entry bits, namely the *dirty bit* or the *write bit*, into account.
The *dirty bit* is of high importance for the Linux internal memory management components, especially for the Page Cache. A set *dirty bit* indicates to synchronize cache contained pages with those versions on disk and the swap partition. Just reading the *dirty bit* does not always inidicate changed content. After modified cache contained data are written back to disk, the bit is reset, thus, a past modification is not visible anymore. Book keeping of modified pages includes resetting the dirty bit to detect new modifications after a taken snapshot, which is dangerous since it affects Page Cache consistency. In [6] page modification detection is done based on the *dirty bit* being mirrored into one of the reserved entry bits.
Our approach to detect modified pages is *write bit* focussed. Each time a write-protection page-fault occurs, the *write bit* is set. Such exceptions are detected by the memory management unit. An exception handler is called and resolves the exception by removing the write-protection (*write bit* set to 1). Based on the detection we record modified pages in a book keeping control structure. At the initial checkpoint all pages of a program address space are saved. After each checkpointing operation all writable pages are explicitly made write-protected (*write bit* is reset to 0). In case an application attempts to write on such a page within a checkpoint interval, the triggered page-fault handler removes the write-protection. Thus, detection of modified pages is enabled during the next checkpointing

operation. Depending on the application behaviour, generally just a subset of all application pages needs to be saved. In order to detect future write attempts in subsequent checkpoint intervals, the write-protection will be reactivated by explicitly resetting the page *write bit*. Special handling of newly added read-only pages and partially written pages after a checkpointing operation is detailed under 3.3.
During restart the last version of a physical page needs to be localized out of multiple checkpoint images. Therefore, a dedicated book keeping control structure is required that keeps track of the last page's file location and offset within the checkpoint image file, described in the following section.
We are conscious about TLB entries, which must be flushed explicitly after each incremental checkpoint, since they are not updated when modifying the *write bit*. The latter could result in inconsistency. However, each process context switch anyway results in flushing the TLB. Taking multiple checkpoints within one scheduler time slice is hard to achieve due to its shortness.

3.2 Book keeping of modified pages

Physical page content of an application address space may be spread across multiple incremental checkpoint images each potentially storing all or just a subset of all pages. At restart the complete content and its consistent versions must be loaded.
We use a book keeping control structure that keeps track of page locations and is based on the Linux-native *radix tree*. The *radix tree* provides fast lookup and insertion operations (O(1)) which are needed to keep the structure in sync with the process memory structure. In figure 2 the localisation of a virtual address-related physical page is shown. Each tree node is identified by a virtual

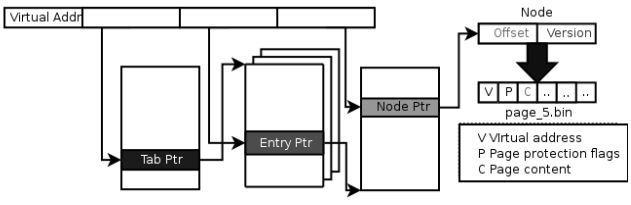

Figure 2: Book keeping control structure for modified pages.

address. A node entry stores the version of a dedicated incremental checkpoint image file (e.g. pages_5.bin for the fifth incremental checkpoint) containing the latest version of the page, and the offset in this file, since more than one page may be modified between two incremental checkpoints. The incremental checkpoint image file stores data blocks each containing a virtual address, page protection flags and the page content itself. Of course, all node entries are also saved to disk at each incremental checkpoint.
During a checkpointing operation the book keeping control structure is updated. That means, book keeping entries targetting not yet referenced pages are added, file locations and offsets of pages that are *present and modified and already referenced* are being updated and saved to disk.
At restart the book keeping control structure is read from disk. Its data is used to localize memory pages out of multiple incremental checkpoint files to restore a process' address space.
The mere write-bit based page modification approach alone is not able to keep the book keeping control structure in sync with the process memory structure, especially if memory pages have been removed. A requirement is to delete such structure entries to avoid wasting memory. Reading from and writing structure content to disk decreases checkpointing performance.Another issue is the unawareness of newly mapped read-only data that cannot be detected and will lead to inconsistency at restart because the appropriate page content is missing. The solution for both cases are detailed in the next section.

3.3 Challenge: memory region changes

Virtual pages belong to a bigger logical unit called memory region or virtual memory area (VMA). Memory regions can be seen as an overlay structure of continuous virtual pages. At one time one virtual address belongs to exactly on memory region. Over time one virtual address may be reassigned to a new memory region. They are implicitly created from user space (e.g. *mmap* system call) and are created/managed by internal memory management mechanisms.
Memory region changes in connection with read-only and writable pages must be taken into account when managing the book keeping control structure. Two criterias must be fulfilled, proper assignment of pages to memory regions, which can be influenced by dynamic region creation/removal, and clean management of

book keeping control structure entries, outdated entries must be deleted. If the later contains inappropriate content, a restart may fail or result in inconsistency. According to Linux memory region management [2] four cases of memory region changes can occur:

Rule 1 (region extension): if a new range of addresses is to be added to a process, the kernel first tries to enlarge an existing memory region. This requires virtual address holes or free address blocks in the process' address space and access rights of the exisiting region and the additional addresses being equal.

Rule 2 (region creation): if a new memory region is created and attached to the process' address space, the kernel tries to merge neighbouring regions, as far as they share the same access rights.

Rule 3 (region shortening): a certain address block can be removed from a region. If this address block resides at the beginning or end of a region, the region is shortened.

Rule 4 (region splitting): if the address block to be removed resides within an existing region, the region is splitted into two smaller regions.

The following examples demonstrate the need for an additional criteria than only checking the *write bit* in order to detect memory region changes, and thus page content changes.

Case 1: An application maps file A in a separate memory region 2. The application gets checkpointed, afterwards file A gets unmapped, memory region 2 vanishes. The application maps file B, accidently having same size and using the virtual address block of former region 2. A new memory region 2 will be created.

In case file B has been mapped as read-only a new in-

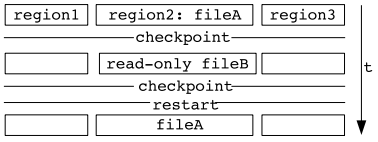

Figure 3: Region2 content with fileB has never been saved, restore old content of fileA

cremental checkpoint does not include the new memory region 2 content, since no *write bit* has been set (see figure 3). At restart memory region 2 will be recreated containing file A (old memory region 2) content which is wrong.

In case file B has been mapped as writable and if it has

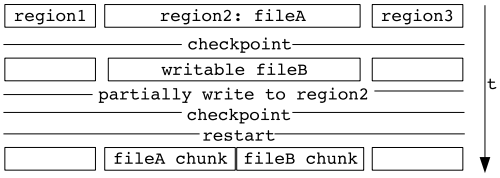

Figure 4: fileB was partially written to, restore mix of old and new content

been partially written to, a new incremental checkpoint results in saving just the pages of region 2 with the *write bit* being set (see figure 4). After restart memory region 2 represents a mixture of file A and file B content which is wrong.

Case 2: this scenario is similar to the first one of case 1. However, a smaller file B is mapped, and thus a smaller memory region 2 will be created, resulting in a hole of the virtual addresses between new region 2 and region 3.

In case file B is mapped read-only, no content of

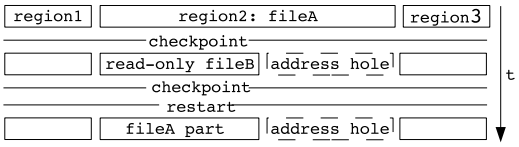

Figure 5: Region2 content with fileB has never been saved, restore part of fileA.

region 2 will be saved because no *write bit* is set (see figure 5). The reduction of virtual addresses of new memory region 2 is not reflected in the book keeping control structure. At restart parts of old memory region 2 will be recreated in the new address range of region 2. These book keeping control structure contains more entries than supposed to be which may result in wasted memory space.

In case file B has been mapped as writable, and if

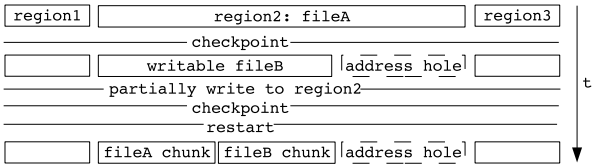

Figure 6: FileB was partially written to, restore mix of old and new content.

it has been partially written to, a mixture of old and

new region 2 content will re reestablished at restart (see figure 6). Furthermore, the book keeping contains out-of-date data, since it does not reflect a memory region shrinkage.

Case 3: three regions exist, a memory hole exists between region 2 and three. At checkpoint time the complete content of all regions is saved. Afterwards, region 2, which maps file A, gets unmapped, a new file B, which is bigger than the previous file A is mapped. New region 2 is placed between region 1 and 3, no memory hole between 1 and 2, as well as 2 and 3 exists.

In case file B is mapped read-only, no new region 2

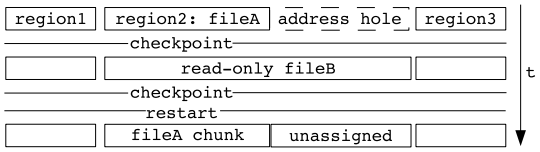

Figure 7: Region2 content with fileB has never been saved, restore fileA and unassigned space

related content is saved at an incremental checkpoint. Especially the additional virtual addresses of new region 2 opposite to old region 2, are not taken into account, since no *write bit* is set. At restart, region 2 contains file A (old region 2) content. Since the book keeping control structure is not aware of additional virtual addresses of new region 2 restart is likely to fail or causes inconsistency.

In case file B is mapped writable, and if it has been

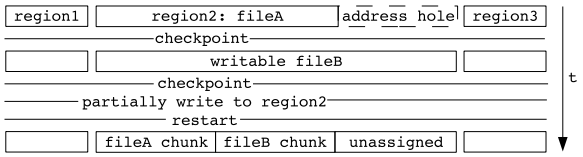

Figure 8: fileB was partially written to, restore mix of old and new content and unassigned space

partially written to, a mixture of old and new content will be reestablished at restart for the address block covered by old region 2. Regarding the additional addresses of new region 2, the same effects are expected as explained shortly before.

Case 4: in the center of memory region 1 an address block is being write-protected having different access rights than the surrounding region 1 parts. Consequently, the Linux memory management enforces region 1 to be *splitted* into three parts. Region 1, 2 and 3, with region 2 containing the pages the *mprotect* call has been applied to.

In case the write-protected region 2 is not written to, or is partially written to, the same effects as described under Case 1 occur.

Case 5: region 1 contains an address block at the end or at the beginning which gets removed. The region got *shortened*. In case region 2 is read-only, the appropriate book keeping control structure entries of the removed address block are not removed. Then, a new region gets created partially or fully covering the previously removed address block.

In case the new region is read-only, or has been partially written to, effects as detailed under Case 1 occur.

3.4 Solution: Memory region modification monitor

The special cases mentioned under 3.3 occur in combination of memory region modifications with read-only and writable content, e.g. such as shared segments, or anonymous memory region content or memory mapped files.

To tackle these issues we introduce an additional logical layer of modification detection - a memory region modification monitor. This monitor keeps track of memory region changes and thus complements the mere write-bit focussed approach of memory page modification detection. Based on monitor data the book keeping control structure can be kept in sync with the actual memory structure of a process at checkpoint time. It is sufficient to update the corresponding book keeping structure entries once, at checkpoint time, instead at each region modification event.

The monitor records region removals and additions. After each checkpoint, monitor data will be flushed. At checkpoint time the monitor entries are used to manage the book keeping control structure. Its entries are compared to control structures entries. In case a region has been removed, the start and end address of each monitor entry is used to delete appropriate book keeping control structure entries. This ensures control structure efficiency and consistency. In case a region has been added, relevant book keeping control structure entries are added and/or updated to reference appropriate checkpoint image contained pages. This allows whole new regions to

be saved initially at the first incremental checkpoint after their creation.
Our monitor supports detection of *mmap* and *munmap* calls. Therefore, we insert a monitor notification function into the kernel functions *do_mmap* and *do_munmap*. Per *mmap* call the start and end address of an affected memory region are inserted into the memory region modification monitor. A detected *munmap* call results in deleting the appropriate entry of the monitor. For example, region creation detection, via the *mmap* call, results in initially saving the whole physical page content at the next checkpoint. Issues as listed under 3.3 can be avoided.
The memory region modification monitor has a similar structure as the book keeping structure. Its entries are

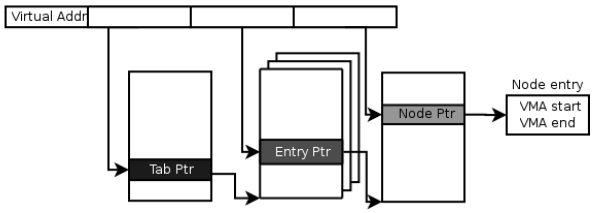

Figure 9: Memory region monitor structure

organised in a radix tree providing fast access for entry removal, addition and retrieval. Each entry contains the memory regions start and end address of the covered virtual address range. The structure is shown in figure 9.

4 Incremental grid checkpointing

In order to realise one flavour of adaptive checkpointing, our incremental checkpointing enhanced kernel checkpointer has been integrated into the XtreemOS grid checkpointing architecture.
For the job checkpointer service to know when it is best to use a full or incremental checkpointing, the number of modified pages of an application must be computed. Therefore, the job checkpointer service has been enabled to detect page modifications in a transparent manner for a given set of processes. Without modifying applications, the service can self-decide which checkpoint approach to be used by keeping efficiency.
Reporting page modifications from the kernel space to the user space domain has been achieved by setting up a new Linux Connector [9]. The service registers at a so-called *memory event connector (MEC)* at kernel-level to

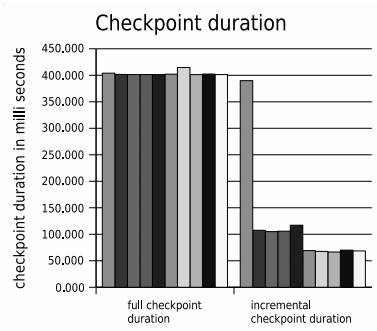

Figure 10: Full and incremental checkpointing duration

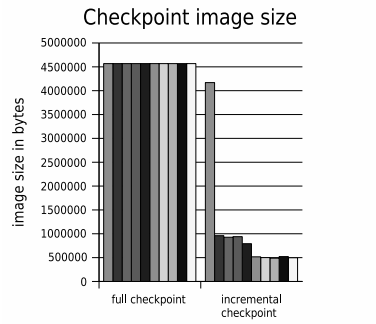

Figure 11: Image size of full and incremental checkpointing

be informed about *do_page_fault* calls triggered by selected processes. The service receives MEC messages at user space and performs accounting on page faults on a per process base. In case the collected data exceed a certain threshold, the job checkpointing service enforces full checkpointing. Otherwise, incremental checkpointing is used.

5 Measurements

The testbed consists of 2 nodes with Intel Core 2 Duo E6850 processors (3 GHz) with 2048 MB DDR2-RAM and being interconnected with gigabit network. A master node runs a tftpboot and a NFS server providing a LinuxSSI image and a Linux environment including the directory for checkpoint image storage to a client node.
Our test application allocates 1 MB of RAM and writes integer values to random locations in 1 millisecond intervals.
Figure 10 shows the checkpoint duration of full and incremental checkpointing if checkpoints are issued in one second intervals.
Both data sets indicate incremental checkpointing taking shorter time especially after the initial checkpoint.

Figure 11 shows the resulting image size of full and incremental cehckpoints. It appears that one inremental checkpoint image file is smaller than a full checkpoint image, especially after the initial checkpoint. Efficiency of incremental checkpointing relies on the write behaviour of an application. Since an additional control structure and a region monitor need to be maintained, incremental checkpointing may become inappropriate, especially the more pages have been changed per checkpoint interval. It is the task of the grid service to figure out the best-suited strategy.
Furthermore, restarting from an incremental checkpoint may result in accessing more than one image file opposite to just one file with full checkpointing. Increased I/O overhead, caused by reading from multiple files, is likely decrease restart performance.

6 Conclusion and Outlook

We described the integration of incremental checkpointing into a Linux-based kernel checkpointer. Our basic approach to detect modified pages, which is the most significant prerequisite to be met for incremental checkpointing, is *write bit* based. We use the *write bit* of pages to detect whether pages have been modified and thus need to be saved. Modification of the *write bit* does not interfere with other Linux memory management functionality, e.g. the Page Cache.
Furthermore, we classified five generic and common scenarios of Linux memory region behaviour, where page content modifications cannot be detected with a mere *write bit* focussed approach. Basically, combinations of memory region creation and removal in connection with read-only and writable content can lead to inconsistency and failure at restart.
We implemented a memory region modification monitor that takes region changes into account in order to save appropriate content at each incremental checkpoint and avoid the scenarios described under 3.3.
Efficient Linux native structures, such as a radix tree, have been used to implement a page-modification book keeping control structure and the memory region modification monitor enabling fast management of incremental checkpointing-specific data at checkpoint and restart.
We profit from recent innovations, namely the generic connector concept. A custom memory event connector (MEC) informs a user-space component of page modifications, which improves the symbiosis of kernel- and grid-level checkpointing components.
We are concious of further events to be monitored regarding region resizings, e.g. caused by *do_mremap*. Besides, we will focus on saving pages contained in the swap area as well. Additionally, we plan to realise concurrent checkpointing to provide more kernel-based functionalities that support adaptive checkpointing at grid level.

References

[1] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, 2008.

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2006.

[3] XtreemOS Consortium. Annex 1 - description of work. Contract funded by the European Commission, April 2006. XtreemOS Integrated Project, IST-033576.

[4] Ian Foster and Carl Kesselman. The globus project: a status report. *Future Gener. Comput. Syst.*, 15(5-6):607–621, 1999.

[5] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan. Design and performance of special purpose hardware for time warp. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 401–409, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[6] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[7] F. Gomes and A. F. Bosco. *Optimizing incremental state-saving and restoration*. PhD thesis, University of Calgary, Calgary, Alta., Canada, Canada, 1996. A.-U. Brian.

[8] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *In Proceedings of SciDAC 2006*, June 2006.

[9] M. Helsey. Process event connector. 2005.

[10] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs - a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(8), 2008.

[11] K. Kolyshkin. Virtualisation in linux. 2006.

[12] D. Lezcano. lxc. 2008.

[13] J. Mehnert-Spahn, T. Ropars, M. schoettner, and C. Morin. Xtreemos grid checkpointing service architecture. 2008.

[14] J. Mehnert-Spahn and M. Schoettner. Design and implementation of basic checkpoint/restart mechanisms in linuxssi d2.2.3. 2007.

[15] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS 96*, pages 78–85, 1996.