Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance

José Carlos Sancho Fabrizio Petrini Kei Davis Roberto Gioiosa
Song Jiang
Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
{jcsancho,fabrizio,kei,gioiosa,sjiang}@lanl.gov

Abstract

Checkpoint/restart is a general idea for which particular implementations enable various functionalities in computer systems, including process migration, gang scheduling, hibernation, and fault tolerance. For fault tolerance, in current practice, implementations can be at user-level or system-level. User-level implementations are relatively easy to implement and portable, but suffer from a lack of transparency, flexibility, and efficiency, and in particular are unsuitable for the autonomic (self-managing) computing systems envisioned as the next revolutionary development in system management. In contrast, a system-level implementation can exhibit all of these desirable features, at the cost of a more sophisticated implementation, and is seen as an essential mechanism for the next generation of fault tolerant—and ultimately autonomic—large-scale computing systems. Linux is becoming the operating system of choice for the largest-scale machines, but development of system-level checkpoint/restart mechanisms for Linux is still in its infancy, with all extant implementations exhibiting serious deficiencies for achieving transparent *fault tolerance. This paper provides a survey of extant implementations in a natural taxonomy, highlighting their strengths and inherent weaknesses.*

Keywords: *Fault tolerance, checkpoint/restart, autonomic computing, Linux.*

1 Introduction

Checkpointing refers to the action of recording the state of a computational process such the process could be restarted at the point of progress represented by this state. Checkpointing, in various forms, is useful for process migration (e.g. for load balancing), gang scheduling, 'hibernation' (to preserve an entire machine state across power-downs), or 'suspension' (as implemented in the commercial virtual machine software VMware Workstation (tm), to save memory space or to allow rolling back to known states), and as a mechanism for enabling fault tolerance.

The need for fault tolerance in the largest-scale current and proposed parallel computers is becoming critically important. Such machines are built primarily for *capability* computing, that is, with the intention of dedicating all or most of the computational capacity to a single application at any given time. For scientific computing, such applications may run for days, weeks, or longer until completion; examples include the US DOE ASC codes [2] among innumerable others. However, because of the extraordinarily large component count of such machines—for instance, the IBM's BlueGene/L supercomputer currently under construction will have 65,536 nodes—their mean time between failures (MTBF) may be orders of magnitude shorter than the execution times of the applications they are intended to run [1]. The current state of practice with such systems is that in the absence of some mechanism for fault tolerance a component failure is catastrophic for the running application; it is all-too-common practice to run an application, or a part of it, many times to achieve one successful completion.

In this scenario checkpoint/restart mechanisms are advocated as a straightforward solution for providing fault tolerance. They are based on periodically saving the process state to stable storage so that in the event of a failure the application can be restarted, on a functioning set of nodes, at the point of the most recent checkpoint. These mechanisms are quite promising assuming *fail-stop semantics* [33] where faults can always be detected—a reasonable assumption in practice.

Additionally, it is implicit in the goals of proposed autonomic computing systems [14]—systems that are self-managing—that a checkpoint/restart mechanism be completely transparent to the application programmer and application user, that is, that the application source code need not be modified, recompiled, or relinked. Such a system should be capable of both automatic and user-initiated checkpoint and restart operations at any time during the execution of the application. For example, checkpointing could be initiated at regular fixed intervals, or in a more sophisticated scheme the interval could be automatically and dynamically optimized with respect to a number of parameters, such as the failure rate of the system. User-initiated checkpointing would allow a system administrator to safely suspend a process to allow another to run, or to accommodate repair or maintenance.

In its simplest form checkpointing saves the entire state of a process. *Incremental checkpointing* [27] is a well-known optimization wherein only that part of a process's state that has changed since the last checkpoint operation is saved. Optimization is achieved when the size of the *delta*—the subset of the application's memory that changed since the last checkpoint operation—is small compared to its entire memory. The page protection mechanism implemented in virtual memory systems is commonly used to keep track of the modifications to the process state, so changes in the application memory are traced at the page granularity. This technique has been recently evaluated with respect to current hardware performance at user level (specifically that of the current bottlenecks, namely I/O bus, disk, and interconnection network) [31]. Experimental results show that the reduction in the size of the checkpoint data depends strongly on the application, but for most relevant scientific applications current hardware is adequate to provide feasible (efficient) solutions. To the best of our knowledge, this technique has seen very few implementations in Linux at user level, and never before at the operating system level. As will be discussed, a system-level implementation allows a number of essential advantages over a user-level implementation.

The primary contribution of this paper is to provide a comprehensive survey of existing checkpoint/restart mechanisms in a natural taxonomy that exposes the fundamental reasons for their potential strengths and unavoidable weaknesses, and in so doing argues that a particular subspace of the taxonomy represents the most desirable area for further development.

The remainder of this paper is organized as follows. Section 2 describes a useful taxonomy for implementations. Sections 3 and 4 analyzes current user- and system-level checkpoint/restart mechanisms, respectively. Section 5 concludes.

2 Checkpoint/Restart Implementations

Checkpoint/restart mechanisms can be roughly classified along three dimensions: the context, the agent that provides the checkpoint/restart functionality, and particular specifics of implementation. To illustrate this classification, Figure 1 depicts the space of checkpoint/restart implementations. In the coarsest dimension, context, an implementation may be *user-level* or *system-level*.

A user-level implementations may be directly programmed in the application's source code by the user or automatically inserted by a pre-compiler. Usually in these cases a specific checkpointing library provides the necessary checkpoint/restart primitives, eliminating the need to directly program them. Alternatively, instead of modifying the source code of the application, the checkpoint/restart primitives may be invoked by signal handlers defined at user level. Another implementation is based on the use of the LD_PRELOAD environment variable wherein signal handlers are installed and the checkpoint library loaded without recompilation or relinking of the application.

In contrast, system-level implementations may be in the *operating system* or in *hardware*. In the operating system there are various techniques for implementing the checkpoint/restart mechanisms: as a *kernel-mode signal handler*, *system call*, or *kernel thread*. In principle the classification may not be entirely clear-cut, but in practice the taxonomy is useful.

3 User-level Implementations

As has been discussed in detail elsewhere [31], implementations at user-level suffer from lack of transparency because the application needs to be modified and recompiled, or relinked against a checkpoint library. However, the upside of these schemes is that the implementation is both simpler and more portable than modifying the kernel. Some representative examples are libckpt [27], libckp [38], Thckpt [39], Esky [15], and Condor [21] to checkpoint simple single-threaded processes; libtckpt [10] for multi-threaded processes; and the Pittsburgh Supercomputing Center's checkpoint library [35], PM2 [37], Dynamite [19], CoCheck [28], CLIP [7], and CCIFT [4] for parallel applications. Most of them are automatically initiated at user-level because the application itself is periodically checkpointed when some *checkpoint* calls are executed. Therefore, the lack of flexibility is a principal concern on these implementations. On the other hand, only a few of these implement automatic-initiated at system-level, and implement incremental checkpointing.

A common scheme is to install a signal handler for a default signal offered by the kernel to automatically initiate the checkpoint operation at system level. The signal handlers are defined at user level and invoked by the kernel. This signal can be triggered by a timer that periodically interrupts the application with the default signal *SIGALARM*; libckpt and Esky use this approach. Others, like Condor, may use some general purpose signals such as *SIGUSR1*, *SIGUSR2*, and *SIGUNUSED*. While they were primary designed for automatic initiation, they may be invoked by the user by explicit use of the *kill* command. Unfortunately these solutions are not general because in many cases signal handlers interfere with the application or the resource manager. Another problem inherent with checkpointing in user space is efficiency: it entails much context switching between user and kernel modes because of the number of

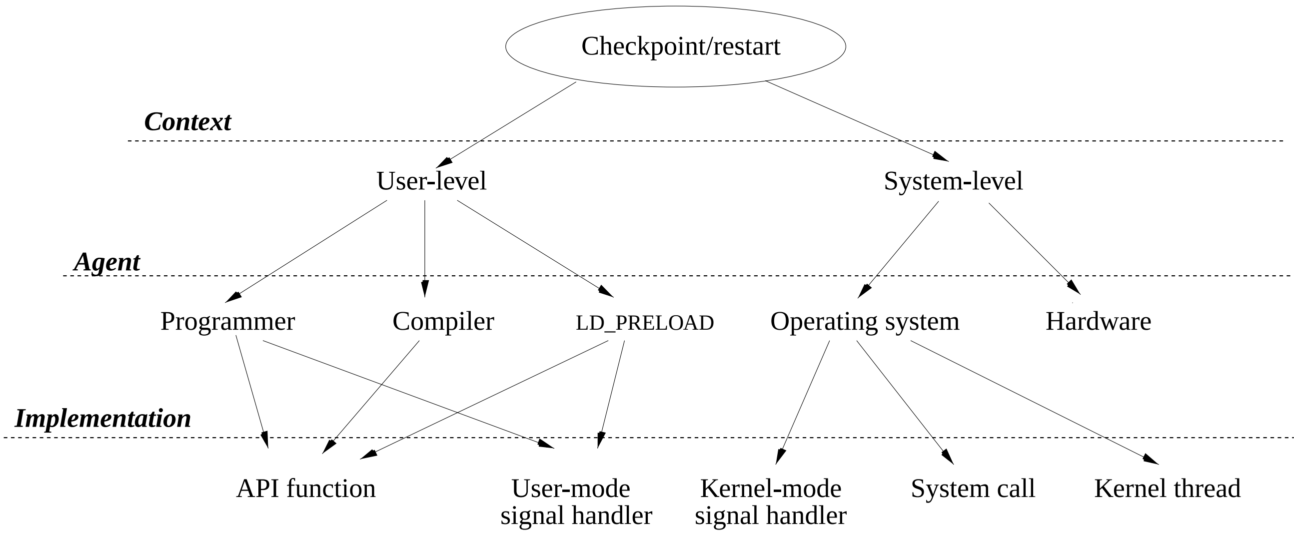

Figure 1. Classification of the checkpoint/restart implementations.

system calls needed to extract from the kernel certain information about the process's state. Though context switching has been highly optimized in Linux [20] it still represents a more costly solution than directly accessing kernel data structures because most CPU registers must be saved and restored every time a system call is performed. For example, in Linux the *sbrk(0)* system call is used to extract the heap boundaries, *lseek()* is used to extract the file indexes, and *sigispending()* is used to extract the signals pending on the process. All of this information is directly accessible in the kernel's process state structure.

Even worse is the fact that some kernel data structures embodying process state are not accessible even indirectly at user level. Short of making extensive kernel modifications it is necessary to replicate these structures in the user space by intercepting system calls, for example *mmap()* and *unmmap()* to trace the dynamic memory, *dlopen()* to trace the dynamic shared libraries, and *open()* or *dup()* to extract file attributes. This approach is extremely undesirable because of added run-time overhead. Moreover, user-level implementations are limited to applications that do not depend on persistent state belonging to the operating system, for example sockets, shared memory, PIDs, and IP addresses. In contrast, a system-level approach can virtualize these resources, implementing completely transparent checkpointing [24]. Further, the *user signaling* scheme represents a more complex scenario in which to program because the use of non-reentrant functions in the signal context can cause deadlock or corruption of the system. For example, some functions of the C library such *malloc()* and *free()* are not reentrant. In contrast, the kernel is designed to be reentrant.

In a user-level implementation incremental checkpointing is realized by tracing modifications to the process's state at the page granularity [27]. The protection of each page in memory is set to read-only using the *mprotect* system call at the beginning of the checkpoint interval. When the application attempts a write access the operating system sends a *SIGSEGV* signal to the process which can then be used to track page modifications. Recently, a novel technique called *probabilistic checkpointing* [23] allows the implementation of incremental checkpointing at a finer granularity. Changes in the application memory are tracked at the granularity of a memory block whose size can be much lower than the size of a entire page. A further development of this scheme is based on using different block sizes in order to provide an attractive compromise between performance and efficiency [1].

4 System-level Implementations

There are two main approaches to checkpointing at system level: an implementation entirely performed by the operating system, and a hardware/software co-design that involves the operating system and special-purpose hardware. In the former, incremental checkpointing is implemented by using the page protection mechanism: when the process tries to access the write-protected page a page fault exception is generated, and changes to application memory are traced at page granularity. As will be explained later, with support by special-purpose hardware modifications of the process's state can be traced at much finer granularity.

4.1 Operating System

In kernel space every data structure relevant to a process's state is readily accessible: these include registers, memory regions, file descriptors, signal state, and more. This accessibility enormously simplifies the implementation of checkpoint/restart operations, though it requires somewhat more knowledge of kernel internals. On balance, though, decreased complexity and increased efficiency practically mandate this approach if the goals of efficiency, transparency, and generality are to be achieved.

There are three main approaches to providing checkpoint/restart functionality at system-level: via a *system call*, a *kernel signal*, or a *kernel thread*.

- **System call.** This approach entails introducing new system calls into the operating system to invoke the checkpoint and restart operations [17, 18, 5]. The common practice is to perform the automatic initiation at user level, that is, the application directly invokes the system calls, thus lack of transparency and flexibility are major concerns.

- **Kernel-mode signal handler.** This approach is based on the signaling mechanism offered by the kernel, but now rather than using a general purpose signal at the user level, a new specific signal is added to the kernel for this purpose [6, 36]. The default action of this signal is to checkpoint the application. The advantage is that the checkpoint is performed at system level instead of the user level. The signal may be generated with the *kill* command at user level or directly at system level.

- **Kernel thread.** Here a kernel thread is created to perform the checkpoint/restart activities [40, 13, 32, 24, 11]. The interaction with the kernel thread can be performed at user level through three possible interfaces: (1) using the standard file operations like *read*, *write*, and *ioctl* to communicate with a device file (usually in */dev*); (2) via the */proc* pseudo file system using the *read* and *write* operations; or, (3) a new system call that may be invoked by another user-level process (like a process monitor) to inform the kernel thread to checkpoint a specific process. Alternatively, checkpoint operations can be initiated at system level using internal mechanisms to start the kernel thread.

All of these approaches require some changes to the kernel, though often it is possible to write most of the code as a kernel module. This improves portability and modularity, and is helpful during development because a module can be loaded and unloaded dynamically.

The system call and kernel-mode signal handler approaches have the advantages of being executed in kernel mode behind the process that is to be checkpointed. Thus the current process address space is that of the process to be checkpointed. In contrast the kernel thread does not have the correct process address space (because kernel threads always use kernel addresses that are independent of processes'), and it uses the page tables of the task it interrupted, which may not be the process that is to be checkpointed. In such a case a process address space switch is required and this may invalidate the TLB cache and so decrease performance.

In the first two approaches the application is executing the checkpointing code (either the system call or the signal handler), so data do not change during the checkpoint operation. A kernel thread, instead, is a different process that in a multiprocessor system might run in parallel with the application. That application could change data while the kernel thread is saving it. In this case a mechanism to stop the application is necessary (like removing the application from the run queue) in order to guarantee data consistency. An alternative approach is to fork (duplicate) the process and let the parent process run while saving the (not yet started) child process before destroying it.

A process may be in a state that is difficult to record or reproduce, for example it could be waiting for an external event such as an interrupt from a device. This problem affects the kernel mode signal handler but not the system call mechanism (if the process is not using some asynchronous function) because in that case it is the application itself that calls the checkpoint function.

The system call approach requires some changes in the application source code in order to call the checkpoint function, resulting in a loss of transparency. Flexibility is also lost with this approach: given that it is the application that invokes the system call, there is little control of when the checkpoint will actually occur. This may introduce undesired delays, and global control of large scale parallel computations could be difficult to implement. The kernel mode signal handler method is more transparent than the system call approach but the execution of the signal handler is deferred until next transition from kernel mode to user mode in the process context. Given that it is difficult to provide accurate estimates on how many processes will be running at any given time, there is no way to know when the signal handler will be executed.

Another problem is related to the time-sharing scheduling algorithm: the process could be suspended by the kernel because there is another process with a higher priority waiting for the CPU (the priority is dynamic and changes during the process execution). Interrupts could also stop the checkpointing. A new scheduler algorithm could alleviate this problem but all the computing processes should have the same (high) priority.A kernel thread is a different process that can have a higher priority policy (like the SCHED_FIFO priority), this will ensure that the thread will be executed as soon as it wakes up and it will run until it has completed its work. Processes can not interrupt a kernel thread with this schedule priority if they do not have the same priority. A new priority can be introduced in order to make sure that the kernel thread will not be interrupted. Interrupts can still stop the thread and a mechanism to delay these events is needed in order to be sure the kernel thread will never be interrupted.

The development of system-level checkpoint/restart functionality for Linux is a relatively recent phenomenon, first appearing around 2001. The first implementations were deployed primarily to provide process migration in clusters. Later implementations provided more advanced functionalities for gang scheduling, hibernation, and fault tolerance. They are briefly described in the following paragraphs.

The original implementations are VMADump [17], EPCKPT [26], and CRAK [40]. The VMADump (Virtual Memory Area Dumper) provides checkpoint/restart capabilities to individual Linux processes via system calls. Applications directly invoke these system calls to checkpoint themselves by writing the process state to a file descriptor. Thus, this approach lacks transparency and flexibility. One advantage of this tool is that the relevant data of the process can be directly accessed through the *current* kernel macro because VMADump is called by the process to be checkpointed. VMADump was designed as a part of the BProc project [18] which is an implementation in the static part of the kernel. This project aims to implement single system image and process migration facilities in clusters.

In EPCKPT the checkpoint/restart operation is also provided through system calls and is very similar to the VMADump scheme; like the VMADump scheme, EPCKPT is also implemented in the static part of the kernel. EPCKPT provides more transparency than VMADump because the process to be checkpointed is identified by the process ID (*pid*) rather than directly by the *current* macro. A new default kernel signal is created to invoke the checkpoint operation. EPCKPT provides some command line tools to user-initiate the checkpoint operations. Application must be launched via one of these tools in order to initialize the checkpoint and trace some information about the application's execution at run time.Then, checkpoints are performed using another tool, passing as parameter the process's pid corresponding to the application to be checkpointed.

CRAK [40] is a process migration utility implemented as a kernel thread. Unlike the previous schemes CRAK is a kernel module. To communicate with the kernel thread CRAK creates a new device in */dev* and the *ioctl* device-file interface is used. The pid of the application to be checkpointed is passed as parameter in the *ioctl* call. The process migration operation can also be disabled by users. In this case, it stores the process's state locally or remotely without performing a process migration. A later development of this tool is ZAP [24]. ZAP improves on CRAK by providing a virtualization mechanism called *Pod* to cope with resource consistency, resource conflicts, and resource dependencies that arise when migrating processes between machines with different persistent states, as commented earlier in Section 3. However, that virtualization introduces some run-time overhead because system calls must be intercepted.

Other checkpoint/restart mechanisms have been subsequently developed, such as the BLCR, the Berkeley Lab's Linux Checkpoint/Restart project [11]. This is a kernel module implementation that, unlike prior schemes, also checkpoints multithreaded processes. Like CRAK it is based on kernel threads and uses the *ioctl* device-file interface to specify the pid's process to be checkpointed. But BLCR needs an initialization phase to register a signal handler for an available general purpose signal and also requires to load a shared library, hence it is not totally transparent. Also, users can specify whether the process state is saved locally or remotely via the *ioctl* system call. A further development of this tool, LAM/MPI [32], allows checkpointing of MPI parallel applications. But, although it is completely transparent to the application, is not transparent to the MPI library because some MPI functions must be modified in order to make the initialization phase of the BLCR scheme automatic.

Another checkpoint/restart package is UCLiK [13] which inherits much of the framework of CRAK, but additionally introduces some improvements like restoring the original process ID and file contents, and identifies deleted files during restart. Process states are saved only locally.

CHPOX [36] is a checkpoint/restart package very similar to EPCKPT, but is implemented as a kernel module that stores the process state locally. It creates a new entry in the */proc* pseudo file system and also a new kernel signal (*SIGSYS*). The user application must be registered by sending the *pid* to the new created entry in */proc*. Then, checkpoints are initiated by sending the new signal to the process. This package has been tested and tuned as part of the MOSIX project [3].

PsncR/C [22] is another checkpoint/restart package for SUN platforms. It is a kernel thread implemented as a kernel module which saves process state to local disk. A new entry in */proc* is created and all checkpoint operations are realized via the *iotcl* interface. Unlike other packages it does not perform any data optimization to reduce the checkpoint data size, so all of the code, shared libraries, and open files are always included in the checkpoints.

Software Suspend [6] is a hibernation mechanism implemented in the official Linux kernel source code. Software Suspend provides a script to start this operations at user-level. A new default kernel signal initiates the hibernation, and is delivered to every process in the system to freeze their execution. When all processes are stopped the image of the RAM is saved on the swap partition in the local disk. After that, the system is powered down. At start-up the image is restored from disk and all the processes are restarted. Additionally, it also provides some standby functionality by saving the image to memory rather to disk.

Finally, there is a recent proposal for checkpoint/restart of multithreaded processes that we will refer to as *Checkpoint* [5]. Checkpoint/restart operations are provided through system calls implemented in the kernel static part. The innovation of this approach is that the checkpoint operations are performed by a thread running concurrently with the application. The *fork* mechanism is used to guarantee the consistency of data between the thread and the application process. However, this approach is not transparent—it requires direct invocation of system calls.

Table 1 summarizes the main features of these mechanisms. As can be seen, most provide full transparency—the application source code needs not be modified, recompiled, or relinked. By counterpart, most of them are totally transparent to the kernel static part. They are implemented as a kernel module, which increases portability.

Most of the implementations provide a user-initiated

TABLE **1. Comparison of Linux System-level Checkpoint/Restart Packages**

Name	Incremental checkpointing	Transparency	Stable storage	Initiation	kernel module
VMADump	no	no	local,remote	automatic	no
BPROC	no	no	none	automatic	no
EPCKPT	no	yes	local,remote	user	no
CRAK	no	yes	local,remote	user	yes
UCLik	no	yes	local	user	yes
CHPOX	no	yes	local	user	yes
ZAP	no	yes	none	user	yes
BLCR	no	no	local,remote	user	yes
LAM/MPI	no	no	local,remote	user	yes
PsncR/C	no	yes	local	user	yes
Software Suspend	no	yes	local	user	no
Checkpoint	no	no	local	automatic	no

checkpoint that relegates the management of the checkpoint operations to system administrators. Thus, they provide rudimentary flexibility and little self-managing capabilities. The common practice to provide flexibility is by integrating the user-initiation operations within a batch management software such as LSF [9] that initiates the checkpoint operations automatically. This system software is layered on top of the operating system providing a set of tools to allocate, monitor, and manage the networked resources in a cluster. In addition, some self-management capabilities are recently incorporated in those software packages [8]. Although, these tools provide flexibility and self-managing capabilities at user-level, we believe that the lack of these capabilities at system-level is a limiting factor for two main reasons: (1) they are relegated to systems that support this special software reducing the applicability of autonomic computers; and (2) reduces the scalability and fault tolerance of autonomic computers because the management is centralized to this software.

In addition, these solutions typically provide only rudimentary support for fault tolerance. Most store the checkpoint locally instead of remotely, thus checkpointed data cannot be retrieved in case of a failure of the machine. Fault tolerance is limited to the case of restarts in the event of power outages or reboots.

Further, incremental checkpointing has not yet been implemented in any of the packages. It has been implemented at system-level in other operating systems like Genesis [30] and V-System [12], but as far as we know, there is no implementation of incremental checkpointing for Linux at the time of this writing.

4.2 Hardware

Checkpointing may be supported by purpose-designed hardware. As with operating-system-level implementations, this approach can be entirely transparent to users. But hardware-level checkpointing is of limited importance precisely because it relies on custom hardware, in contrast to the trend of building clusters from commodity components.

Hardware-based schemes typically implement incremental checkpointing at much finer granularity than is done at the operating system level: modifications of the address space of the application are traced at the granularity of cache lines. There are two recent proposals for hardware-supported checkpointing for shared-memory multiprocessors, *Revive* [29] and *Safetynet* [34]. In *Revive* checkpointing is supported by modifications of the hardware related to the directory controller of the machine. In comparison, *Safetynet* requires more hardware resources than *Revive*. The processor's caches must be modified, and it also requires an additional buffer to store the checkpointing data.

5 Conclusions

We have surveyed the current state of the art of checkpoint/restart mechanisms, identifying the significant advantages and disadvantages of each.

Unlike user-level schemes, those at operating system level can provide the flexibility, transparency, and efficiency required to support the envisioned paradigm of autonomic computing, even on commodity hardware. The checkpoint/restart functionality implemented at the operating system level can be automatically invoked without user intervention and can be integrated with the system management tools. We believe that the automatic-initiated functionality at system-level brings new management capabilities in large scale computers.

Acknowledgments

This work was supported in part by the U.S. Department of Energy through LDRD-ER grant 2001034ER and the Advanced Simulation and Computing (ASC) program, and by the Defense Advanced Research Projects Agency under the High Productivity Computing Systems Program, under Los Alamos National Laboratory contract W-7405-ENG-36.

References

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive Incremental Checkpointing for Massively Parallel Systems. In *Proceedings of the International Conference on Supercomputing*, Malo, France, June 26–1, 2004.

[2] Asci codes. Available from `http://www.nnsa.doe.gov/asci`.

[3] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the Principles and Practice of Parallel Programming*, San Diego, California, June 11–13, 2003.

[5] C. Carothers and B. Szymanski. Checkpointing of Multithreaded Programs. *Dr. Dobbs Journal*, 15(8), August 2002.

[6] F. Chabaud, N. Cunningham, and B. Blackham. Software Suspend for Linux. Available from `http://softwaresuspend.berlios.de/Software-suspend.html`.

[7] Y. Chen, J. S. Plank, and K. Li. CLIP – A Checkpointing Tool for Message-Passing Parallel Programs. In *Proceedings of the Supercomputing*, San Jose, California, November 15–21, 1997.

[8] Platform Computing Corporation. LSF Administrator's Guide, Version 4.1. Available from `http://www.lions.odu.edu/docs/lsf4.1/admin_4.1/index.html`.

[9] Platform Computing Corporation. PLATFORM LSF HPC, Version 6.1. Available from `http://www.platform.com/products/HPC/`.

[10] W. R. Dieter and J. E. Lumpp. User-level Checkpointing for LinuxThreads Programs. In *Proceedings of the 2001 USENIX Technical Conference*, Boston, Massachusetts, June 25-30, 2001.

[11] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Available from `http://old-www.nersc.gov/research/FTG/checkpoint/blcr.pdf`.

[12] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, TX, October 5–7, 1992.

[13] M. Foster. Pursuing the AP's to Checkpointing with UCLiK. In *Proceedings of the 10th International Linux System Technology Conference*, Saarbrucken, Germany, October 14–16, 2003.

[14] A. G. Ganek and T. A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1), January 2003.

[15] D. Gibson. Checkpoint/restart for Solaris and Linux. Available from `http://cap.anu.edu.au/cap/projects/esky/index.html`.

[16] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of System Overhead on Parallel Computers. In *The 4th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2004)*, Roma, Italy, December 2004.

[17] E. Hendriks. VMADump. Available from `http://cvs.sourceforge.net/viewcvs.py/bproc/vmadump`.

[18] E. Hendriks. BProc: The Beowulf Distributed Process Space. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, New York City, June 22–26, 2002.

[19] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The implementation of Dynamite – An Environment for Migrating PVM Tasks. *Operating Systems Review*, 34(3), July 2000.

[20] K. Lai and M. Baker. A Performance Comparison of UNIX Operating Systems on the Pentium. In *Proceedings of the USENIX Technical Conference*, San Diego, CA, January 22–26, 1996.

[21] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Computer Sciences Technical Report 1346, University of Wisconsin-Madison, Madison,Wisconsin, April 1997.

[22] N. Meyer. User and Kernel Level Checkpointing. In *Proceedings of the Sun Microsystems HPC Consortium Meeting*, Phoenix, Arizona, November 15-17, 2003.

[23] H. C. Nam, J. Kim, S. J. Hong, and S. G. Lee. Probabilistic Checkpointing. *IEICE Transactions on Information and Systems*, E85-D(17):1093–1104, June 2002.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 9–11, 2002.

[25] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003.

[26] E. Pinheiro. EPCKPT. Available from `http://www.research.rutgers.edu/~edpin/epckpt`.

[27] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter 1995 Technical Conference*, New Orleans, Louisiana, January 16–20, 1995.

[28] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, Honolulu, Hawaii, April 15–19, 1996.

[29] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, May 25–29, 2002.

[30] J. Rough and A. Goscinski. Exploiting Operating System Services to Efficiently Checkpoint Parallel Applications in GENESIS. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, Beijing, China, October 23-25, 2002.

[31] J. C. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium*, Santa Fe, New Mexico, April 26–30, 2004.

[32] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *Proceedings of the LACSI Symposium*, Santa Fe, New Mexico, October 12-14, 2003.

[33] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerance Computing Systems. *ACM Transactions on Computer Systems*, 1(3), August 1983.

[34] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, May 25–29, 2002.

[35] N. Stone, J. Kochmar, R. Reddy, J. Ray Scott, J. Sommerfield, and C. Vizino. A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System. Technical Report CMU-PSC-TR-2001-0002, Pittsburgh Supercomputing Center, November 2001. Available from `http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.PDF`.

[36] O. O. Sudakov and E. S. Meshcheryakov. Process Checkpointing and Restart System for Linux. Available from `http://www.cluster.kiev.ua/eng/tasks/chpx.html`.

[37] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: A High Performance Communication Middleware for Heterogeneous Network Environments. In *Proceedings of the Supercomputing*, Dallas, TX, November 4-11, 2000.

[38] Y. Wang, Y. Huang, K. Vo, P. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, Pasadena, California, June 27-30, 1995.

[39] R. Xue. Thckpt. Available from `http://sourceforge.net/projects/thckpt`.

[40] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, New York, November 2001.