Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers

Roberto Gioiosa[1,2] José Carlos Sancho[1] Song Jiang[1] Fabrizio Petrini[3]
Kei Davis[1]

[1]Performance and Architecture Laboratory
Computer and Computational Sciences Division
Los Alamos National Laboratory, NM 87545, USA
{jcsancho,sjiang,kei}@lanl.gov

[2] Dipartimento Informatica Sistemi e Produzione
Universitá degli Studi di Roma "Tor Vergata"
00133 Roma, Italy
gioiosa@ing.uniroma2.it

[3]Applied Computer Science Group
Computational Sciences and Mathematics Division
Pacific Northwest National Laboratory, WA 99352, USA
fabrizio.petrini@pnl.gov

Abstract

We describe the software architecture, technical features, and performance of TICK (Transparent Incremental Checkpointer at Kernel level), a system-level checkpointer implemented as a kernel thread, specifically designed to provide fault tolerance in Linux clusters. This implementation, based on the 2.6.11 Linux kernel, provides the essential functionality for transparent, highly responsive, and efficient fault tolerance based on full or incremental checkpointing at system level. TICK is completely user-transparent and does not require any changes to user code or system libraries; it is highly responsive: an interrupt, such as a timer interrupt, can trigger a checkpoint in as little as 2.5μs; and it supports incremental and full checkpoints with minimal overhead—less than 6% with full checkpointing to disk performed as frequently as once per minute.

1 Introduction

Several projects have been launched by the U.S. Department of Energy, in concert with computer manufacturers, to develop computers of unprecedented size to provide the capability for large high-fidelity physics simulations. For example, the BlueGene/L initiative, a research partnership between IBM and the Lawrence Livermore National Laboratory, is to deliver a system capable of 360 Tflops [2]. On the horizon are massively parallel machines intended to deliver petaflop (Pflop) or multi-Pflop performance, such as the machines envisioned by the DARPA HPCS projects [8].

It is clear that future increases in performance will be achieved, in large part, by increases in parallelism, and thus component count.[1] The large total number of components of these systems will make any assumption of complete reliability entirely unrealistic: though the

[1]That increase may be at different levels of integration: physical boxes, boards, chips, processor cores, or ultimately, transistors.

SC'05 November 12-18, 2005, Seattle, Washington, USA

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

mean time between failures for individual components (e.g., processors, disks, memories, power supplies, network components, and cooling systems) may be very high, the sheer number of components in the system will inevitably lead to frequent individual failures.

The growing complexity of managing failures in such large and sophisticated systems threatens their usefulness as capability engines. Ideally such systems would automatically detect, diagnose, and mitigate localized software and hardware problems, possibly without any change to existing application software. However, this envisioned *autonomic* system management is still a research area in its infancy.

1.1 Checkpoint/restart

A common approach to guaranteeing an acceptable level of fault tolerance in scientific computing is *checkpointing*—periodically saving the state of an application to nonvolatile storage so that if any process of a parallel application fails, all of its processes may be rolled back to the last checkpoint and restarted.[2] While this approach seems straightforward, capturing sufficient state of a parallel application in a scalable and efficient way can have subtle complications. In general the state of a parallel application at a given point in real time can be quite complex: messages may be in flight, memory may be shared between subsets of processes, individual processes may have signals pending, local and global files may be open, processes' memory may be partially stored to disk (swapped), among other possibilities.

The most direct approach to checkpointing is to save the entire state of the application. A well-known optimization is to checkpoint *incrementally*, where at each checkpoint only that part of the application's state is saved that has changed since the previous checkpoint. This requires a mechanism for determining what has changed and can entail considerable bookkeeping in the general case. A recent feasibility study obtained on a state-of-the-art cluster showed that efficient, scalable, automatic, and user-transparent incremental checkpointing is within reach with current technology [12]. Specifically, the study shows that current standard storage devices and high-performance networks provide sufficient bandwidth to allow frequent incremental checkpointing of a suite of scientific applications of interest with negligible degradation of application performance.

[2]Checkpointing provides a solution for *fail-stop* faults only, a model in which faulty processors simply stop on failure. More complex models, such as *Byzantine*, allow other behaviors.

1.2 Scope of problem and contribution

Ideally a checkpointing mechanism is complete in the sense that all process state is saved. In practice, however, many, if not most, implementations are predicated on various simplifying assumptions. Because the most common operating systems (Linux in our case) and run-time libraries were not designed with checkpointing in mind, process state can be highly distributed; besides the memory image of the process itself the operating system may contain other essential state: data structures created on behalf of the process, pending or progressing interrupts or signals, open files and other communication channels, other processes with which a process may be interacting via shared memory, and more. For a parallel application running on a cluster-like architecture with a communication network the situation is more complex: the physical network and communication libraries add yet more state, and system-wide checkpoints must be consistent.

Our approach is not to attempt to handle all of these issues with a single solution, rather to apply a natural factorization of concerns. The primary contribution of this paper is a description of a checkpoint/restart mechanism that is applicable on a per-CPU basis. It is intended to be a truly general-purpose, very-low overhead basis for a system-wide solution for cluster-like parallel machines. For example, it could be seamlessly integrated with a parallel job scheduler or with an MPI implementation.

The major contributions of our implementation, TICK (Transparent Incremental Checkpointing at Kernel-level), are

- *Full transparency*: TICK doesn't require any change to applications or system libraries. A typical approach to implementing system-level checkpointing is through signal-based callbacks that execute a system call within the context of the checkpointed process [9]. While this is not a significant intrusion in a Linux cluster where signals are seldom used, with TICK we seek to eliminate the need for *any* change to user software. With TICK we explore a new design avenue, namely using a Linux kernel thread that can access the user process irrespective of its scheduling status.

- *Full or incremental checkpointing*: With TICK it is possible to save just those segments of memory that have changed since the last checkpoint. TICK uses an efficient algorithm to compute the incremental changes, at page granularity, in the memory geometry of a user process. It also provides a choice of mechanisms for detecting the

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

changed, or *dirty*, pages.

- *Very low overhead*: TICK can fully or incrementally checkpoint a process with modest overhead. When checkpointing to memory as frequently as every minute the overhead is less than 4% for all of the applications tested. For checkpointing to disk the worst-case overhead is slightly greater, on the order of 6%.

- *High responsiveness*: TICK can be triggered by an external event, such as a timer interrupt or a global cluster heartbeat, in as little as 2.5μs.

- *Flexibility*: TICK provides a number of features that can be easily controlled through a */proc* file system interface, readily supporting more complex, cluster-wide coordinated checkpoint strategies.

1.3 Achieving fault-tolerance in a large-scale parallel computer

On a parallel computer, to achieve fully-transparent fault tolerance based on frequent checkpointing, several problems need to be adequately addressed. At a first level of analysis we can identify three important dimensions:

1. Global orchestration to reach a consistent recovery line across a large number of processing nodes;

2. Adequate support at the level of the local operating system to save, restore, and transfer large amounts of data; and,

3. Adequate hardware support for efficient global coordination. Elsewhere we have addressed the functional and performance impact of scalable coordination mechanisms with very encouraging results [16, 14, 17].

In the absence of explicit program points at which to checkpoint, the run-time system must identify or induce global recovery lines at which to save the state of the parallel application. The processes of an application may interact by sending messages or signals so it is necessary to keep track of the outstanding communication to ensure that a checkpoint can be safely re-played, for example by draining the network before a checkpoint [11].

We plan to attack the problem of global orchestration by using an innovative methodology called *buffered coscheduling* (BCS) [25]. The vision behind BCS is that both the size and complexity of system software may be greatly reduced by constraining system-wide asynchronicity. In brief, the entire cluster marches to the beat of a global strobe that is issued every few hundreds of microseconds. In each *timeslice*—the period between strobes—newly-issued communication calls are buffered until the next timeslice. At every strobe, nodes exchange information on pending communication so that every node has complete knowledge of the required incoming and outgoing communication for the next timeslice. The nodes then proceed to globally schedule those communications that will actually be carried out during the timeslice, and proceed to execute them. The advantage of this model is that all the communication is controlled and is in a known state at every timeslice, so that problems arising from congestion, out of order arrival, and hot spots can be avoided. We have shown that these constraints result in minimal performance degradation for our scientific applications, while obtaining the advantages of a more deterministic, controllable machine [13]. This global coordination supports a truly parallel operating system that combines the instances of the local operating systems into a single, cohesive system. We have already demonstrated a scalable resource management system, called STORM, that achieves performance orders of magnitude faster than existing production-grade software by using the BCS paradigm [18].

One of the major advantages of the global orchestration strategy of BCS is that it can automatically enforce *global recovery lines*—points in time where there are no messages in transit and where all of the processes of the application can be easily checkpointed. This approach is in contrast to classical techniques that first stop the processes of a parallel application and then wait for the network to drain before checkpointing [11].

Finally, the local operating system of each processing node needs to be augmented with new mechanisms, for example to transparently checkpoint user processes or to provide hot swapping of hardware devices. With TICK we explore the design and technical challenges that need to be solved to provide fully-transparent checkpoint and restart for the Linux operating system.

2 Background

Checkpointing schemes may be usefully categorized as being *user level* or *system level*. In practice the distinction between user- and system-level checkpointing is not clearcut. However, there is a clear dichotomy between a checkpoint/restart mechanism that is completely transparent to, and independent of, the application, and one that is not.

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

2.1 User-level checkpointing

A typical approach to avoiding many of the complexities of checkpointing is based on taking an application-centric point of view, and exploiting knowledge of the structure and behavior of a given application. In this approach checkpointing is initiated, and to some degree managed, from within the application. The application programmer identifies program points at which all essential state can be captured from within the application. For communication, for example, the state could be known following a global synchronization. The saving of state is performed explicitly by the application, in most cases with library support [27]. On restart, restoration of state is similarly explicitly handled. This approach is called *user-level* because it is handled in user space rather than by the OS. User-level checkpointing has some significant attractions. By knowing, at a given program point, what data is essential for restarting, the programmer can selectively save only that data, often substantially reducing the size of the saved state. User-level checkpointing only requires the availability of a parallel file system, and therefore can be easily adapted to other parallel machines.

User-level checkpointing has several drawbacks. Because the points at which the program is in a sufficiently well-known state may be temporally distant—on the order of minutes or even hours apart—loss of progress in the event of a rollback can be substantial. This may be mitigated by the programmer introducing 'artificial' points at which the state is manageable, which results in added runtime, program complexity, and programmer effort. In any case, determining the optimal frequency for checkpointing may be non-trivial.

Perhaps most critical is the issue of correctness: whether sufficient state is recorded, and potential race conditions are identified, among numerous other considerations. Correctness depends entirely on the programmer, and the use of user-level checkpointing introduces whole a new dimension of potential bugs that only appear on restart and so are very difficult to test for. Finally, all of this effort must be expended for each application.

2.2 System-level checkpointing

In contrast, in a generally-applicable approach the checkpoint mechanism is independent of the application to be checkpointed—the application is 'unaware' that it is being checkpointed and, possibly many times, rolled back and restarted. This approach is then *transparent* to the application, and when it is implemented at the operating system level, it is usually referred to as *system-level*.

3 TICK

TICK is designed to be a modular building block for implementing checkpointing in large-scale Linux clusters. TICK is implemented primarily as a Linux 2.6.11 kernel module, and consists of about 4,000 lines of code, with an additional 400 lines in the static part of the kernel. TICK is neutral with respect to where checkpoint data is saved: its function is to correctly capture and restore process state. The actual management of the checkpoint data is handled by one or more separate agents, which in our prototype implementation are other Linux kernel modules. For lack of space this paper will not describe the algorithms for data movement that can be used to implement a global checkpointer for parallel applications. The checkpoint data may be saved locally if a process restart is all that is needed, for example after a machine crash, or to a file system when the instances of TICK on each CPU of a cluster are globally coordinated.

While our primary goal is fault tolerance in large-scale parallel computers, we believe that TICK could be useful in other environments such as distributed or grid computing, and much more directly, for load balancing via process migration. The essential properties of TICK are that it is

- *Kernel level:* TICK is implemented at kernel level to allow unrestricted access to processor registers, memory allocation data structures, file descriptors, signals pending, etc.
- *Implemented as a kernel module:* Writing, debugging and maintaining kernel code can be time consuming and non-portable. Most of TICK's code is in a kernel module that can be loaded and removed dynamically.
- *General purpose:* The TICK checkpoint/restart mechanism works with any type of user process, and processes may be restarted on any node with the same operating environment.
- *Flexibly initiated:* The checkpointing mechanisms of TICK can be triggered by a local event, such as a timer, or a remote event, such as a global strobe, in a very short and bounded time interval.
- *User transparent:* The user processes are not involved in the checkpointing or restarting and there is no need to modify existing applications or libraries. This implies that TICK can support ex-

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

isting legacy software, written in any language, without any changes.

- *Efficient:* TICK tries to minimize degradation of performance of a user process when checkpointing its state. TICK also implements fast process restarts.
- *Incremental:* TICK can perform frequent incremental checkpointing on demand.
- *Easy to use:* TICK provides a simple interface based on the */proc* file system that can be used by a user or system administrator to dynamically checkpoint or restart a user process on demand.

3.1 Design of TICK

There are several possible approaches to implementing a kernel-level checkpoint/restart mechanism. It is useful to divide these approaches in two main categories: *internal* and *external*, to distinguish whether the checkpoint is taken within the context of the user process or within a possibly different process context. TICK is implemented as an external checkpoint system in order to ensure determinism, high responsiveness, and transparency. The considerations and trade-offs are briefly discussed.

3.1.1 Internal checkpointing

In an internal checkpoint the kernel code is executed within context of the checkpointed process via a system call or signal handler: this ensures that the checkpointer uses the correct process address space (because there is no page table switch when a process makes a system call or receives a signal). This would be the easier choice to implement. Unfortunately the system call method requires source code modification or library support, which we regard as undesirable. The signal approach does not require source code modification but it would not ensure enough determinism: in general it is not possible to predict when an application will receive a signal, and so when the checkpoint signal handler will be executed. Processes can only receive signals when they switch from kernel to user mode (for example when the system is returning from an interrupt or an exception handler), just before resuming execution of the user code. The fastest case is when the application is already running on another processor, then the kernel can send an IPI (Inter-Processor Interrupt) to that processor and, at the end of the IPI, the signal handler is executed. In this case the latency is only a few microseconds (the time needed by the IPI interrupt handler). In contrast, an application that frequently sleeps (is I/O-bound, for example) may delay indefinitely before executing the signal handler. Generally speaking, a running process goes into kernel mode every 1/HZ seconds (typically HZ≈1000 in a standard Linux kernel, so every 1 ms). On average the kernel checks for pending signals to a running process every 1/(2*HZ) seconds (every 500 μs). Even in this simple case the variance appears to be too large if determinism is one of the goals. The Linux scheduler may introduce further non-determinism: if, when the signal is generated, the process is not running, it will wait until the scheduler assigns it to a processor.

3.1.2 External checkpointing

In external checkpointing a separate process checkpoints the running application. The obvious choice of mechanism at kernel level is a kernel thread. In this case the main problem is to ensure that this thread is working in the running process' address space. A kernel thread is a normal process (with a process descriptor, status, etc.), but it does not have its own process address space: it uses the page tables of the interrupted process. This means that the kernel thread may need to change the address space, invalidating the TLB cache. Having an external process gives better control of the system and a clean separation of the application and the checkpointer, and can help achieve a greater degree of determinism. A kernel thread is like any other process: it can sleep, wait, access resources, etc., so it can also be used as a global coordinator of the threads on the other nodes of the cluster. The Linux kernel already provides some interesting scheduling policies for real-time processes that can be used to avoid processor sharing and to guarantee that the thread will be executed as soon as it wakes up. The wake-up time of a kernel thread with the SCHED_FIFO scheduling policy is between 5000 and 7000 clock cycles (that is, between 2.5μs and 3.5μs on a 2Ghz processor) irrespective of the number of standard processes running in the system. If other real-time processes were present a new scheduling policy could be implemented. This latency is significantly lower than that of a signal handler, and the variance is also lower. Moreover, this policy ensures no preemption from other processes: the thread will run as long as it needs without sharing the CPU with any other process in the system.

3.2 The TICK algorithm

TICK is implemented as a kernel thread for each processor in a node: if N processes are running on distinct processors, N parallel checkpoints can be taken

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

at the same time. Each kernel thread is bound to one CPU, so each checkpointed process must also run on the same processor using the processor affinity features provided by the Linux kernel. Because full checkpointing requires only a subset of the functionality required for incremental checkpointing we will describe only the latter.

3.2.1 Checkpointing

In order to checkpoint a process a */proc* interface is used to pass the process ID and the associated processor ID (which also identifies the kernel thread). When the kernel thread incrementally checkpoints the process, it reconstructs the *process history*—what files it opened, how many memory regions it holds or it released, etc. The thread initially write-protects the relevant sections of the address space so that the first time the application tries to write to one of these pages the access can be detected and recorded. During process execution, if the incremental checkpointing algorithm is used, events such as file accesses and changes to the geometry of the address space (for example allocation or deallocation of segments of *mmapped* memory) must also be recorded.

To checkpoint a process the kernel thread performs the following steps.

1. Stops the running process;
2. Switches to the process address space, if needed;
3. Saves the contents of the registers in the kernel stack of the process, and some other information from the process descriptor (such us the floating point or debug registers);
4. Saves the signal status (signals pending reception by the process), and the signal handlers defined by the process;
5. Saves the file descriptors;
6. Saves memory region descriptors and updates (in case of incremental checkpoint) and (dirty) pages;
7. Restores the address space, if needed;
8. Restarts the process.

The kernel thread stops monitoring the process if it receives an appropriate command through the */proc* filesystem, is unloaded, or if the process terminates.

3.2.2 Restart

TICK can restart a user process on either the same node from which it was checkpointed, or on another node. In the latter case no assumptions can be made about the memory mapping of the new node. All memory regions must be re-mapped, both for code and shared libraries. When a process is started the loader relocates some memory regions starting from the binary file, but afterward there is no correlation between these regions and the files that originated them. In our approach we fork a container process running the same binary. After loading the binary image in memory the process address space is appropriately modified. It may not be possible to restore exactly the same state, for example, the parent relationship of the process in the new node, or the physical location of a shared library in memory. Creating a container process is an easy way to avoid this kind of problem. Once this process is running, we pass further information, such as the process ID, to the kernel through the */proc* file system. The kernel thread then performs the following actions.

1. Opens the process checkpoint file. In this file there may be many *sections* depending on the algorithm used for checkpointing; each section S_i contains the information related to a checkpoint. The first section, the *initial state* (S_0), is a full checkpoint. Restoring only this section is equivalent to restarting the process from the beginning. Subsequent sections contain the incremental changes. (For full checkpointing the file contains only one section.)
2. Stops the process;
3. Switches the address space, if needed;
4. Deletes all of the process address space except the regions that map executable code (these regions are marked as read-only);
5. For each section (in incremental checkpoint):
   (a) Restores the registers;
   (b) Restores pending signals;
   (c) Restores the memory regions and all of the dirty pages;
6. Switches the address space if necessary;
7. Restarts the process.

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

3.3 Saving Dirty Pages

When the incremental checkpointing option is selected, TICK provides a choice of three mechanisms to keep track of the dirty pages.

1. **Dirty bit.** After a checkpoint, all the writable pages are marked as clean (non-dirty). When the application writes to one of those pages the hardware sets the dirty bit in the corresponding page table entry, with no overhead. At the next checkpoint all of the page table entries are examined to determine whether the page has been changed since the previous checkpoint. In order to maintain the correct kernel functionalities, the original dirty bit is mirrored by one of the unused bits in the page table entry. The low-level functions used by the kernel to access this bit are properly updated.

2. **Bookkeeping.** After a checkpoint all the writable pages are marked as read-only. When a page is overwritten, a page fault exception occurs and the page fault exception handler saves the address of the page in an external data structure. At the end of the checkpoint interval it is only necessary to scan the data structure that tracks the dirty pages.

3. **Bookkeeping and Saving.** This mechanism is the same as the previous one except that the page is also copied into a buffer. No copying is needed at the end of a time slice.

The best choice depends on the algorithm used for global coordination, the application behavior, and the checkpoint interval: if the application overwrites many pages during a checkpoint interval, using the dirty bits may be the best option because it saves time during the computational phase, and the number of pages that need to be examined at the end of the time slice is almost the same in all of the three cases. If the application changes only few pages during the time slice (for example, because the checkpoint interval is very short) the bookkeeping (possibly with saving) may be the best choice.

Depending on the coordination algorithm, only a subset of these mechanisms may be feasible. In some cases we might need to save the changes during the checkpoint interval with the copy-on-write approach so, irrespective of its performance, we may not be able to use the dirty bits.

3.4 Saving and restoring memory regions

The most expensive operation of the checkpointing algorithm, both in terms of time and space, is the copying of the memory pages. With incremental checkpointing, ideally we would save only the dirty pages generated between two consecutive checkpoints. However, because of complications introduced by changes in the geometry of the memory regions, keeping track of the dirty pages is not enough.

The memory address space of a process consists of a sequence of memory regions whose descriptors are linked in the order of the addresses they represent. Each region has its own properties, such as access permissions. Because a page belongs to a memory region and is subject to the region properties, operations such as *mmap* and *munmap* affect all the pages in their scope. In the Linux kernel, unmapping a region causes all its pages to be discarded. Thus incremental checkpointing could be very inefficient if regions were un-mapped and later re-mapped whenever there were a change in its address space. This would force all of the pages in the region to be saved.

One way to avoid this problem is to record all of the memory region operations such as *mmap* and *munmap*. Then the same sequence of operations can be replayed on restart. Though this scheme minimizes the pages to be saved, it can multiply the number of region operations, many of which might be redundant. This is because we only need to recreate the memory region geometry at the next checkpoint, and many of region operations between the two checkpoints may not contribute to the final geometry. Such faithful but wasteful saving and restoring could severely degrade the efficiency of many applications.

To eliminate the unnecessary saving of pages we keep track of the address space changes between two consecutive checkpoints, and save the address space ranges that are unchanged from the last checkpoint. By comparing the incremental changes in the memory geometry between two temporally adjacent checkpoints we avoid tracking every change to the memory geometry.

4 Performance Evaluation

The usefulness of a tool such as TICK depends critically on its performance. We have chosen a set of scientific applications for our performance evaluation, BT, LU and SP taken from the NAS Suite of Benchmarks [3], and Sage [21] and Sweep3D [34], that are representative of scientific computations performed at Los Alamos National Laboratory. In previous work,

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

Table 1: Checkpoint Latency

	Disk		Memory	
Memory Footprint Size	Time	Bandwidth	Time	Bandwidth
100MB	1.67s	59.8MB/s	0.139s	719MB/s
200MB	3.60s	55.5MB/s	0.286s	700MB/s
300MB	5.38s	55.7MB/s	0.424s	707MB/s

Table 2: Restart Latency

	Disk		Memory	
Memory Footprint Size	Time	Bandwidth	Time	Bandwidth
100MB	4.39s	22.7MB/s	0.185s	540MB/s
200MB	8.176s	22.8MB/s	0.361s	554MB/s
300MB	13.18s	22.7MB/s	0.495s	606MB/s

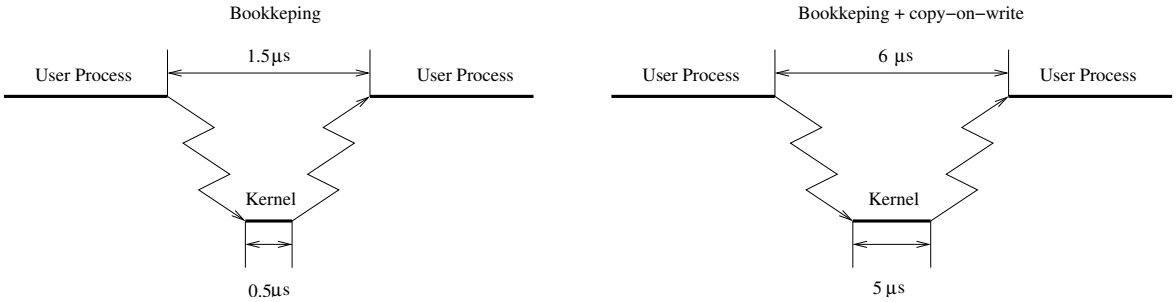


Figure 1: Overhead of two basic incremental checkpointing mechanisms

Sage was found to be the most demanding test for checkpointing algorithms because of its large memory footprint and lack of data locality [30].

4.1 Experimental Platform

The experimental platform is a dual-processor AMD Opteron cluster that uses *Quadrics QsNet^{II}* as the high-performance interconnection network [1]. Each processing node contains two AMD Opteron Model 246 processors, 3GB RAM, and a Seagate Cheetah 15K SCSI disk. The proposed checkpoint/restart mechanisms have been implemented in the Linux kernel version 2.6.11 with the page size configured to 4KB. This system architecture is representative of large-scale clusters such as the ASC *Lightning*, installed at Los Alamos National Laboratory, that has 2,816 Opteron processors [22].

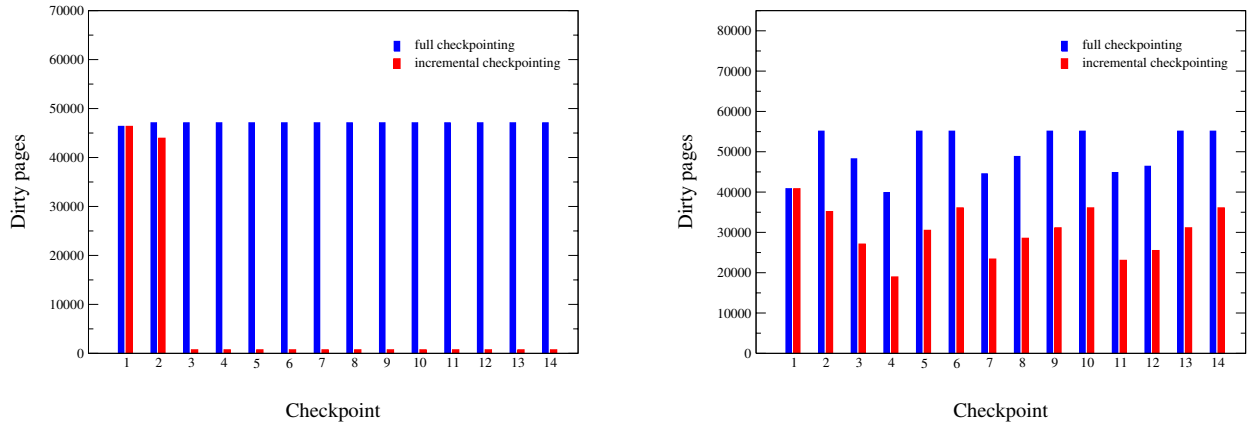
4.2 Basic Performance

Tables 1 and 2 report the save and restore latencies for a sequence of test processes as a function of the size of the memory footprint. The checkpoint is stored in main memory and on local disk. In both cases the asymptotic performance is determined by the speed of the storage media. These numbers provide the baseline overhead that will be incurred every time a user process is saved or restored With a slower local disk the save and restore times are in the order of a few seconds, while using main memory brings these delays down to less than one second.

Figure 1 shows where the time is spent in two of the three incremental checkpointing mechanisms provided by TICK when handling a page fault. The mirroring of the dirty bit, as already discussed, introduces no overhead during a timeslice, but a delay is incurred at the end of the timeslice when the list of dirty bits is scanned.

In the simplest case, when a user process tries to access a write-protected page, TICK only keeps track of the page access in a kernel data structure. As shown in Figure 1 the kernel takes approximately 0.5μs to perform the bookkeeping, while the application process is halted for 1.5μs. Adding the copy-on-write of the user page to a kernel buffer increases the basic delay to 4.5μs.

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE



(a) Number of dirty pages obtained for the application CG Class C using a checkpointing timeslice of 40s.

(b) Number of dirty pages obtained for the application Sage 200MB using a checkpointing timeslice of 20s.

Figure 2: Number of dirty pages obtained with full and incremental checkpointing for sequential checkpoints taken during the execution of the applications CG and Sage. After the initialization, CG overwrites only a small subset of its address space. In contrast, Sage overwrites a large fraction of its address space.

4.3 Application Performance

Next we evaluate the level of intrusion of TICK on user applications in two different scenarios: when performing full checkpoints and with incremental checkpoints. In both cases the checkpoints are triggered by a timer interrupt at regular intervals. We intentionally choose very small intervals—in all experiments one minute or less—that may be unreasonably short for state-of-the-art clusters running scientific applications. These demanding tests show the performance of TICK in the extremely disfavorable case of very frequent checkpoints.

In these experiments checkpoints are stored locally in memory buffers allocated at kernel level or on the local SCSI disk. Major differences between the two storage targets are the access bandwidth and the degree of overlap between computation and data transfer: while the bandwidth to memory is much higher than to local disk, writing to disk can overlap with the execution of the application.

We consider three instances of Sage and Sweep3D with differing memory footprints. Figure 3 shows that the overhead introduced by TICK is remarkably low. When the checkpoints are stored in main memory every minute the worst case is only 4% with Sage-300MB. With disk checkpointing, the worst case is slightly greater, 6% with BT-C.

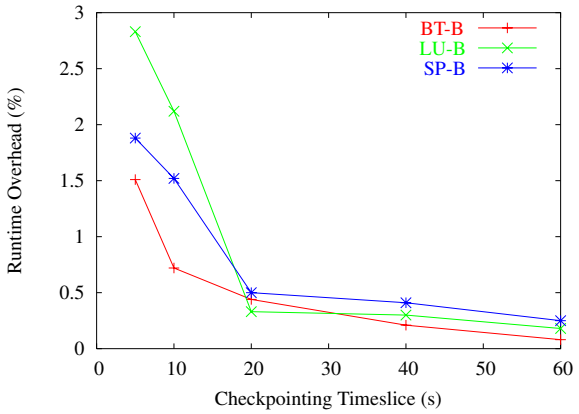
The level of intrusion of incremental checkpointing is strongly influenced by the characteristics of user application and can vary widely. Figure 4 compares the performance of full and incremental checkpointing of a subset of four applications. In most cases incremental checkpointing is beneficial with small timeslices; with large timeslices there is little difference between incremental and full checkpointing. Some applications, such as CG (Figure 4(a)), overwrite a small fraction of their address space in each timeslice. Figure 2(a) shows the number of dirty pages of CG for the first 14 timeslices. After initialization CG overwrites only a small subset of its address space.

The situation is different with other applications, for example Sage (Figure 4(c)): with timeslices larger than 20s, there is no measurable difference between incremental and full checkpointing. Figure 2(b) provides more insight on the properties of the two types of checkpointing algorithms in this scenario. While with full checkpointing we need to save a larger number of pages, disk access can be overlapped with computation, providing the same performance as the incremental algorithm.

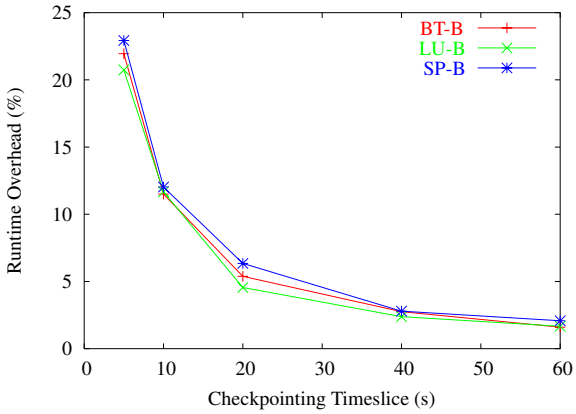
5 Related Work

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, others were released to provide other advanced functionalities such as gang scheduling, hibernation, and fault-tolerance.

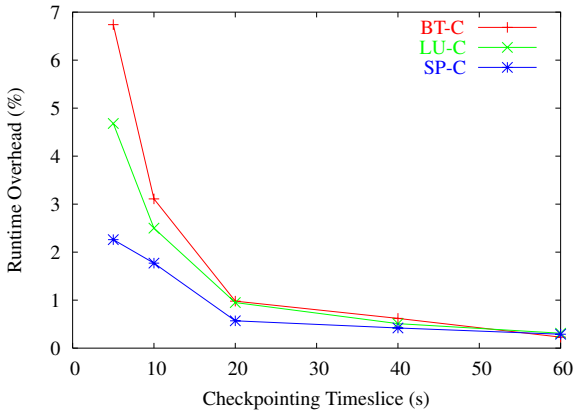
Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE



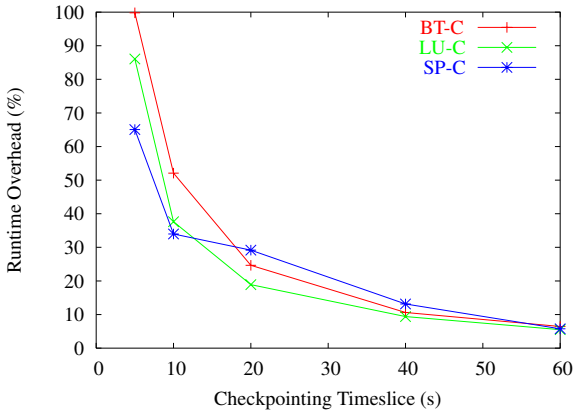
(a) Runtime overhead for NAS benchmarks, class B when storing the checkpoints to main memory.



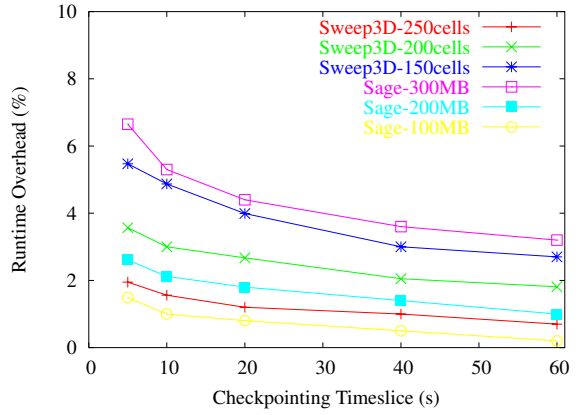
(b) Runtime overhead for NAS benchmarks, class B when storing the checkpoints to the local disk.



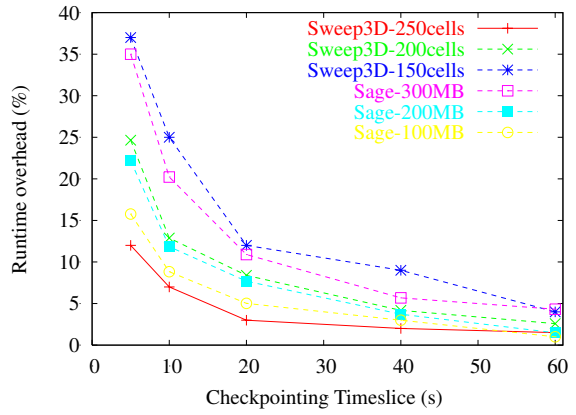
(c) Runtime overhead for NAS benchmarks, class C when storing the checkpoints to main memory.



(d) Runtime overhead for NAS benchmarks, class C when storing the checkpoints to the local disk.



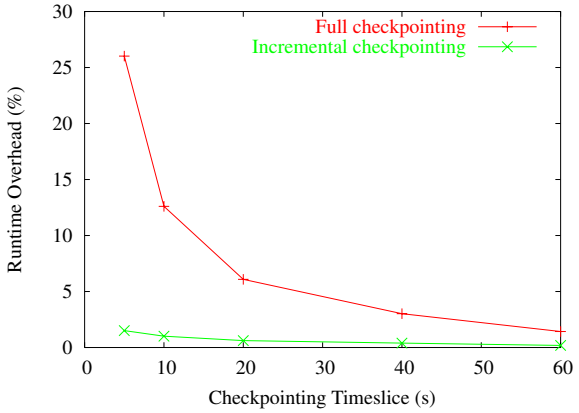
(e) Runtime overhead for Sage and Sweep3D when storing the checkpoints to main memory.



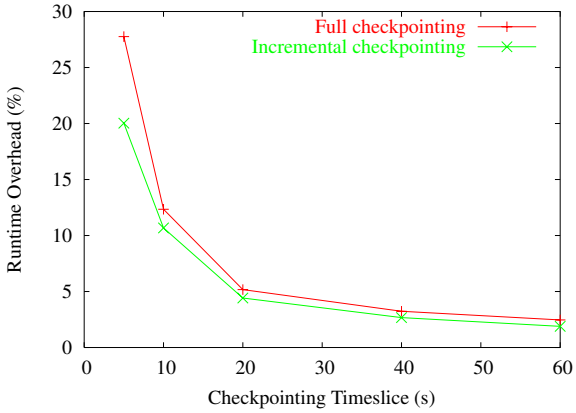
(f) Runtime overhead for Sage and Sweep3D when storing the checkpoints to the local disk.

Figure 3: Runtime overhead of full checkpointing for various checkpoint intervals when storing the checkpoints to main memory, and also to the local disk for the NAS benchmarks and Sage and Sweep3D.

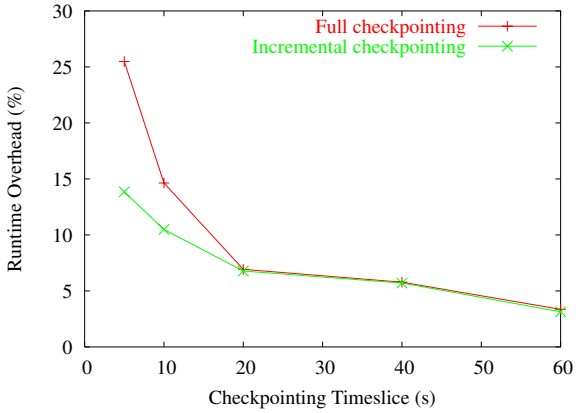
Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE



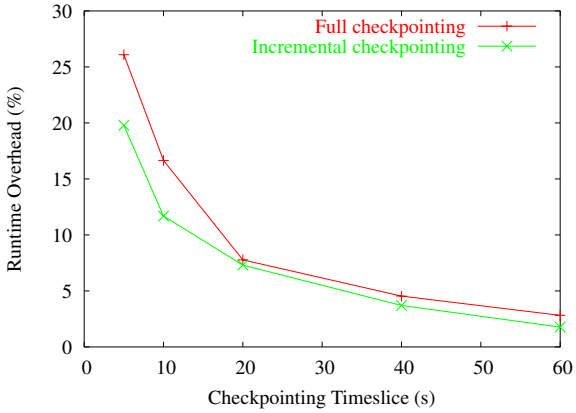
(a) Runtime overhead for the application CG Class C.



(b) Runtime overhead for the application SP Class B.



(c) Runtime overhead for the application Sage 200MB.



(d) Runtime overhead for the application Sweep3D.

Figure 4: Runtime overhead of the full and incremental disk checkpointing for different checkpoint intervals for the NAS benchmarks, and the applications Sage and Sweep3D.

Early implementations include VMADump [19], EPCKPT [26], and CRAK [35]. 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. 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. However, this approach lacks transparency and flexibility of the checkpoint interval. VMADump was designed as a part of the BProc system [20]. 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. EPCKPT provides more transparency than VMADump because the process to be checkpointed is identified by the process ID rather than directly by the `current` macro. A user-defined signaling scheme is used to invoke the checkpoint operation. Prior to the execution of an application a system call must be made to initialize the checkpoint and set up the checkpoint signal handler.

CRAK [35] is a process migration utility implemented as a kernel module, hence it provides more portability than the previous schemes. Processes are automatically checkpointed through the *ioctl* device file interface of the checkpoint module. 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. CRAK does not make use of a special signal like EPCKPT; processes are suspended with the default signal *SIGSTOP*. A later development of this tool is ZAP [24]. ZAP improves on EPCKPT by providing a virtualization mechanism called *Pod* to cope with the resource consistency, resource conflicts, and resource dependencies that arise when migrating pro-

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

Table 3: Comparison of Linux System-level Checkpoint/Restart Packages

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

cesses. However, that Virtualization introduces some run-time overhead since system calls must be intercepted.

Other checkpoint/restart mechanisms have been subsequently developed such as the BLCR [9] (Berkeley Lab's Linux Checkpoint/Restart project). This is a kernel module implementation that, unlike prior schemes, also checkpoints multithreaded processes. Users can specify whether the process state is saved locally or remotely. An initialization phase, as in EPCKPT, is required. A further development of this tool, LAM/MPI [31], allows checkpointing of MPI parallel applications.

Another checkpoint/restart package is UCLiK [15] which inherits much of the framework of CRAK, but additionally introduces some improvements like restoring the original process id, the file contents, and identifies deleted files during restart. Process states are saved only locally.

CHPOX [33] is a checkpoint/restart package very similar to EPCKPT, but it is implemented as a kernel module that stores the process state locally. This package has been tested and tuned as part of the MOSIX project [4].

PsncR/C [23] is a checkpoint/restart package for SUN platforms. It is implemented as a kernel module that saves process state to local disk. Unlike other packages it does not perform any optimization to reduce the size of the checkpoint file, hence the text code, shared libraries and open files are always included in the checkpoints.

Software Suspend [7] is a checkpoint/restart package intended to provide hibernation in Linux. It is invoked interactively by the user to initiate the hibernation by sending a user-defined signal to every process to freeze the execution. When all processes are stopped the checkpoint can take place, saving each process' state to the local disk. At start-up, every process' state is restored from the checkpoint file on disk.

Finally, there is a recent proposal for checkpoint/restart of multithreaded processes; we will refer to it as *Checkpoint* [6]. Checkpoint/restart operations are provided through system calls. 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 3 summarizes the main features of these mechanisms. As can be seen, most provide full transparency and system initiation, but the incremental checkpointing optimization has not yet been implemented in any of them. Operating systems like Genesis [29] and V-System [10] provided some basic checkpointing mechanisms, but to the best of our knowledge there is no implementation of incremental checkpointing for Linux at the time of this writing.

Other related efforts are implementations of process migration mechanisms for distributed operating systems. A notable example of a kernel-level implementation is DEMOS/MP [28].

Recently, user-level checkpointing has been combined with a compilative approach in order to overcome some of the limitations of the user-level approach. Schulz et. al., and Bronevetsky et. al., describe a system that can checkpoint MPI programs [32, 5]. Their systems have two components: a pre-compiler for source-to-source transformation of applications, and a

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

run-time system that implements a global protocol for coordinating the processes of a parallel application. The pre-compiler, currently limited to C programs, transforms the original user program into an intermediate form where checkpoint points may be inserted. Their system, remarkably, is also able to move processes across heterogeneous architectures. However, the transformation introduces a machine-dependent overhead of 1.7-9.6%.

6 Conclusions

We have described the motivation, design, functionality, and performance of the full- and incremental-checkpoint mechanism TICK. TICK embodies all of the properties we believe are necessary at node level to implement an efficient checkpoint/restart mechanism for parallel computers. TICK provides high responsiveness: the checkpoint can be triggered by an external event such as a global heartbeat in as little as 2.5μs. It provides several mechanisms to implement incremental checkpointing at fine granularity with little overhead. It is also very modular, and allows quick prototyping of distributed checkpointing algorithms. TICK is implemented as a Linux 2.6.11 kernel module that will be released as open source.

The experimental results, obtained on a state-of-the-art cluster, show that TICK can be used as a building block for various checkpointing algorithms. We have demonstrated that with TICK it is possible to implement frequent incremental checkpointing, with intervals of just a few seconds, with a run-time increase that is less than 10% in most configurations.

7 Future Work

Currently, the checkpoint/restart mechanism developed has the capability of checkpointing only sequential applications running on a single process that does not use interprocess communication (sockets, pipes, FIFOs, and IPC) or dynamic loaded shared libraries. We plan to support all of these features as well as checkpointing of MPI parallel applications.

Acknowledgments

This work was supported by the ASC Institutes program and by LDRD ER 20040480ER "Self-Healing High-Perfomance Parallel Computers" at Los Alamos National Laboratory.

Los Alamos National Laboratory is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36.

References

[1] D. Addison, J. Beecroft, D. Hewson, M. McLaren, and F. Petrini. Quadrics QsNet II: A Network for Supercomputing Applications. In *Hot Chips 14*, Stanford University, California, August 18–20, 2003.

[2] N. R. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the Supercomputing 2002, also IBM research report RC22570 (W0209-033)*, Baltimore, Maryland, November 16–22, 2002.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Moffett Field, California, December 1995.

[4] 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.

[5] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level Checkpointing for Shared Memory Programs. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, October 2004.

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

[7] F. Chabaud, N. Cunningham, and B. Blackham. Software Suspend for Linux.

[8] High Productivity Computing Systems (HPCS) initiative in DARPA. Available from `http://http://www.darpa.mil/ipto/programs/hpcs/index.html`.

[9] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart.

[10] 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.

[11] Yoav Etsion and Dror G. Feitelson. User-Level Communication in a System with Gang Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, IPDPS2001*, San Francisco, CA, April 2001.

[12] Fabrizio Petrini and Kei Davis and José Carlos Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In *In 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04)*, Santa Fe, NM, April 2004.

[13] Juan Fernández, Eitan Frachtenberg, and Fabrizio Petrini. BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003.

[14] Juan Fernández, Eitan Frachtenberg, Fabrizio Petrini, Kei Davis, and José Carlos Sancho. Architectural Support for System Software on Large-Scale Clusters. In *The 2004 International Conference on Parallel Processing, (ICPP-04)*, Montreal, Quebec, Canada, August 2004.

[15] 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.

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE

[16] Eitan Frachtenberg, Kei Davis, Fabrizio Petrini, Juan Fernández, and José Carlos Sancho. Designing Parallel Operating Systems via Parallel Programming. In *Euro-Par 2004*, Pisa, Italy, August 2004.

[17] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernández, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *ACM/IEEE SC2002*, Baltimore, MD, November 2002.

[18] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernández, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of SC2002*, Baltimore, Maryland, November 16–22 2002.

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

[20] 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.

[21] D. J. Kerbyson, H. J. Alme, A. Hoisie, F.Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the Supercomputing*, November 10–16, 2001.

[22] Lightning Linux Cluster. Available from `http://www.lanl.gov/worldview/news/releases/archive/03-107.shtml`.

[23] N. Meyer. User and Kernel Level Checkpointing. In *Proceedings of the Sun Microsystems HPC Consortium Meeting*, Phoenix, Arizona, November 15-17, 2003. Available from `http://checkpointing.psnc.pl/Progress/sat_nmeyer.pdf`.

[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] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 16:123–144, 2001.

[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] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *ACM symposium on Operating systems principles*, Bretton Woods, New Hampshire, 1983.

[29] 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.

[30] 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.

[31] 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.

[32] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, , and Paul Stodghill. Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs. In *ACM/IEEE SC2004*, Pittsburgh, PA, November 10–16, 2004.

[33] 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`.

[34] The ASCI Sweep3D Benchmark. Available from `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/`.

[35] 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.

Proceedings of the 2005 ACM/IEEE SC|05 Conference (SC'05)
1-59593-061-2/05 $20.00 © 2005 IEEE