DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop

Jason Ansel* Kapil Arya† Gene Cooperman†

*Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA
jansel@csail.mit.edu

†College of Computer and Information Science
Northeastern University
Boston, MA
{kapil,gene}@ccs.neu.edu

Abstract

DMTCP (Distributed MultiThreaded CheckPointing) is a transparent user-level checkpointing package for distributed applications. Checkpointing and restart is demonstrated for a wide range of over 20 well known applications, including MATLAB, Python, TightVNC, MPICH2, OpenMPI, and runCMS. RunCMS runs as a 680 MB image in memory that includes 540 dynamic libraries, and is used for the CMS experiment of the Large Hadron Collider at CERN. DMTCP transparently checkpoints general cluster computations consisting of many nodes, processes, and threads; as well as typical desktop applications. On 128 distributed cores (32 nodes), checkpoint and restart times are typically 2 seconds, with negligible run-time overhead. Typical checkpoint times are reduced to 0.2 seconds when using forked checkpointing. Experimental results show that checkpoint time remains nearly constant as the number of nodes increases on a medium-size cluster.

DMTCP automatically accounts for fork, exec, ssh, mutexes/semaphores, TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals), terminal modes, ownership of controlling terminals, signal handlers, open file descriptors, shared open file descriptors, I/O (including the readline library), shared memory (via mmap), parent-child process relationships, pid virtualization, and other operating system artifacts. By emphasizing an unprivileged, user-space approach, compatibility is maintained across Linux kernels from 2.6.9 through the current 2.6.28. Since DMTCP is unprivileged and does not require special kernel modules or kernel patches, DMTCP can be incorporated and distributed as a checkpoint-restart module within some larger package.

1. Introduction

Checkpointing packages have been available for over 20 years. They are now often used in high performance computing and batch environments. Yet, they have not widely penetrated to ordinary applications on the desktop. There is a need for a simple transparent checkpointing package for commonly used desktop applications (including binary-only commercial applications provided by a third-party), that can at the same time handle distributed, multithreaded production computations on clusters. Unlike the most widely used systems today, DMTCP is *user-level*, requiring no system privileges to operate. This allows DMTCP to be bundled with the application, thereby opening up entirely new applications for checkpointing.

As one striking example use case, many programs have a CPU-intensive first phase, followed by a second phase for interactive analysis. The approach described here immediately enables a user to run the CPU-intensive portion of a computation on a powerful computer or cluster, and then migrate the computation to a single laptop for later interactive analysis at home or on a plane.

Another application of checkpointing that has been well received among users of DMTCP is the ability to easily debug long-running jobs. When bugs in the middle of a long-running job are discovered, the programmer can repeatedly restart from a checkpoint taken just before the bug occurred and examine the bug in a debugger. This reduces the debug-recompile cycle for such cases.

Checkpointing is an inherently hard problem to solve in a robust way. The most widely used checkpointing approaches today are based on custom kernel modules. This approach limits the applications of checkpointing because it can only be deployed in controlled environments. Additionally, kernel modules are hard to maintain because they directly access internals of the kernel that change more rapidly than standard APIs. As evidence of this, the web site checkpointing.org includes many earlier attempts at checkpointing. Most of the attempts that were based on modification of the kernel do not work on current kernel versions.

Checkpoint/restart has been successful to date in batch systems for production use (but sometimes with restrictions on distributed or multithreaded applications). Here, it is reasonable to spend large amounts of manpower to maintain kernel-specific checkpointing schemes. This motivates why many batch queues make checkpointing available, but in contrast, "checkpointing on the desktop" is not as widely

available today. DMTCP tries to support both traditional high performance applications and typical desktop applications. With this in mind, DMTCP supports the critical feature of transparency: no re-compilation and no re-linking of user binaries. Because it supports a wide range of recent Linux kernels (2.6.9 through the current 2.6.28), it can be packaged as just one module in a larger application. The application binary needs no root privilege and need not be re-configured for new kernels. This also painlessly adds a "save/restore workspace" capability to an application, or even to a problem-solving environment with many threads or processes.

Ultimately, the novelty of DMTCP rests on its particular combination of features: user-level, multithreaded, distributed processes connected with sockets, and fast checkpoint times with negligable overhead while not checkpointing. Those features are designed to support a broad range of use cases for checkpointing, which go beyond the traditional use cases of today.

1.1. Use Cases

In this section, we present some uses of checkpointing that go beyond the traditional checkpointing of long-running batch processes. Many of these additional uses are motivated by desktop applications.

1) save/restore workspace: Interactive languages frequently include their own "save/restore workspace" commands. DMTCP eliminates that need.
2) "undump" capability: programs that would otherwise have long startup times often create a custom "dump/undump" facility. The software is then built, dumped after startup, and re-built to package a "checkpoint" along with an undump routine. One of the applications for which we are working with the developers; cmsRun, has exactly this problem: initialization of 10 minutes to half an hour due to obtaining reasonably current data from a database, along with issues of linking approximately 400 dynamic libraries: unacceptable when many thousands of such runs are required.
3) a substitute for PRELINK: PRELINK is a Linux technology for prelinking an application, in order to save startup time when many large dynamic libraries are invoked. PRELINK must be maintained in sync with the changing Linux architecture.
4) debugging of distributed applications: all processes are checkpointed just before a bug and then restarted (possibly on a single host) for debugging.
5) checkpointed image as the "ultimate bug report"
6) applications with CPU-intensive front-end and interactive analysis of results at back-end: Run on high performance host or cluster, and restart all processes on a single laptop
7) traditional checkpointing of long-running distributed applications that may run under some dialect of MPI, or under a custom sockets package (e.g. iPython, used in SciPy/NumPy for parallel numerical applications.)
8) robustness: upon detecting distributed deadlock or race, automatically revert to an earlier checkpoint image and restart in slower, "safe mode", until beyond the danger point.

1.2. Outline

Section 2 covers related work. Section 3 describes DMTCP as seen by an end-user. Section 4 describes the software architecture. Section 5 presents experimental results. Finally, Section 6 presents the conclusions and future work.

2. Related Work

There is a long history of checkpointing packages (kernel- and user-level, coordinated and uncoordinated, single-threaded vs. multithreaded, etc.). Given the space limitations, we highlight only the most significant of the many other approaches.

DejaVu [29] (whose development overlapped that of DMTCP) also provides transparent user-level checkpointing of distributed process based on sockets. However, DejaVu appears to be much slower than DMTCP. For example, in the Chombo benchmark, Ruscio et al. report executing ten checkpoints per hour with 45% overhead. In comparison, on a benchmark of similar scale DMTCP typically checkpoints in 2 seconds, with essentially zero overhead between checkpoints. Nevertheless, DejaVu is also able to checkpoint InfiniBand connections by using a customized version of MVAPICH. DejaVu takes a more invasive approach than DMTCP, by logging all communication and by using page protection to detect modification of memory pages between checkpoints. This accounts for additional overhead during normal program execution that is not present in DMTCP. Since DejaVu was not publicly available at the time of this writing, a direct timing comparison on a common benchmark was not possible.

The remaining work on distributed transparent checkpointing can be divided into two categories:

1) *User-level MPI libraries for checkpointing* [4], [5], [12], [14], [15], [32], [34], [36], [37]: works for distributed processes, but only if they communicate exclusively through MPI (Message Passing Interface). Typically restricted to a particular dialect of MPI.
2) *Kernel-level (system-level) checkpointing* [13], [16], [18], [19], [30], [31], [33]: modification of kernel; requirements on matching package version to kernel version.

A crossover between these two categories is the kernel level checkpointer BLCR [13], [30]. BLCR is particularly notable because of its widespread usage. BLCR itself can only checkpoint processes on a single machine. However some MPI libraries (including some versions of OpenMPI, LAM/MPI, MVAPICH2, and MPICH-V) are able to integrate with BLCR to provide distributed checkpointing.

Three notable distributed kernel-level solutions based on the Linux kernel module Zap are provided by Laadan and Nieh [18], [19] and Janakiraman et al. [16], and Chpox by Sudakov et al. [33]. This approach leads to checkpoints being more tightly coupled to kernel versions. It also makes future ports to other operating systems more difficult.

Much MPI-specific work has been based on *coordinated checkpointing* and the use of hooks into communication by the MPI library [14], [15]. In contrast, our goal is to support more general distributed scientific software.

In addition to distributed checkpointing, many packages exist which perform single-process checkpointing [1], [2], [6], [8], [20]–[22], [24]–[26].

For completeness, we also note the complementary technology of virtual machines. As one example, VMware offers both snapshot and record/replay technology for its virtual machines. The process-level checkpointing of DMTCP is inherently a lighter weight solution. Further, process-level checkpointing makes it easier to support distributed applications. VMware players require system privilege for installation, although snapshot and record/replay can thereafter be used at user level.

Further discussion of checkpointing is found in the following surveys [10], [17], [28].

3. Usage and Features

The user will typically use three DMTCP commands:

```
dmtcp_checkpoint [options] <program>
dmtcp_command <command>
dmtcp_restart_script.sh
```

The restart script is generated at checkpoint time. Each invocation of `dmtcp_checkpoint` by the end user causes the corresponding process to be registered as one of the set of processes that will be checkpointed. All local and remote child processes are checkpointed recursively. As an example, to run an MPICH-2 computation under DMTCP the user would first run:

```
dmtcp_checkpoint mpdboot -n 32
dmtcp_checkpoint mpirun <mpi-program>
```

Note that the MPI resource management processes are also checkpointed. The first call to `dmtcp_checkpoint` will automatically spawn the checkpoint coordinator. `mpdboot` will call `ssh` to spawn remote processes, these calls are transparently intercepted and modified so the remote processes are also run under DMTCP. To request a checkpoint, the user would then run:

```
dmtcp_command --checkpoint
```

Checkpoints may also be generated at regular intervals by using the `--interval` option or requested by the application via the DMTCP programming interface. The checkpoint images for each process are written to unique filenames in a user specified directory. Additionally, a shell script, `dmtcp_restart_script.sh`, is created containing all the commands needed to restart the distributed computation. This script consists of many calls to `dmtcp_restart`, one for each node.

A more detailed list of options and commands for controlling the behavior of DMTCP are described in the manpages shipped with DMTCP.

3.1. Programming Interface

DMTCP is able to checkpoint unmodified Linux executables. We envision the typical use case as having the checkpointed application completely unaware of DMTCP. (This is the configuration used in experimental results.) However, for those wishing to have more control over the checkpointing process, we provide a library for interacting with DMTCP called `dmtcpaware.a`. This library allows the application to: test if it is running under DMTCP; request checkpoints; delay checkpoints during a critical section of code; query DMTCP status; and insert hook functions before/after checkpointing or restart.

4. Software Architecture

DMTCP consists of 17,000 lines of C and C++ code. DMTCP is freely available as open source software and can be downloaded from:

http://dmtcp.sourceforge.net/

DMTCP is built upon our previous work, MTCP (MultiThreaded CheckPointing) [27]. MTCP is assigned responsibility for checkpointing of individual processes, while DMTCP checkpoints and restores socket/file descriptors and other artifacts of distributed software. This novel two-layer design greatly aids in maintenance and portability.

4.1. Design of DMTCP

DMTCP refers both to the entire package, and to the distributed layer of the package. The two layers of DMTCP, known as DMTCP and MTCP, consist of:

1) DMTCP allows checkpointing of a network of processes spread over many nodes. After DMTCP copies all inter-process information to user space, it delegates single-process checkpointing to a separate checkpoint package.
2) We base single-process checkpointing on our previous work, MTCP (MultiThreaded CheckPointing) [27].

These two layers are separate, with a small API between them. This two-layer user-level approach has a potential advantage in non-Linux operating systems, where DMTCP can be ported to run over other single-process checkpointing packages that may already exist.

Checkpointing is added to arbitrary applications by injecting a shared library at execution time. This library:

- Launches a checkpoint management thread in every user process which coordinates checkpointing.
- Adds wrappers around a small number of `libc` functions in order to record information about open sockets at their creation time.

System calls and the `proc` filesystem are also used to probe kernel state.

We use a *coordinated checkpointing* method, where all processes and threads cluster-wide are simultaneously suspended during checkpointing. Network data "on the wire" and in kernel buffers is flushed into the recipient process's memory and saved in its checkpoint image. After a checkpoint or restart, this network data is sent back to the original sender and retransmitted prior to resuming user threads. A more detailed account of our methodology can be found in Section 4

The only global communication primitive used at checkpoint time is a barrier. At restart time, we additionally require a discovery service to discover the new addresses for processes migrated to new hosts.

4.2. Initialization of an application process under DMTCP

At startup of a new process `dmtcp_checkpoint` injects $dmtcphijack.so$, the DMTCP library responsible for checkpointing, into the user program. Library injection is currently done using `LD_PRELOAD`. Library injection can also be done after program startup [35] and on other architectures [9].

Once injected into the user process, DMTCP loads $mtcp.so$, our single process checkpointer, and calls the MTCP setup routines to enable integration with DMTCP. MTCP creates the checkpoint manager thread in this setup routine. DMTCP also opens a TCP/IP connection to the checkpoint coordinator at this time. This results in a copy of our libraries and manager residing within each checkpointed process.

DMTCP adds wrappers around a small number of `libc` functions. This is done by overriding libc symbols with our library. For efficiency reasons, we avoid wrapping any frequently invoked system calls such as `read` and `write`. The wrappers are necessary since DMTCP must be aware of all forked child processes, of all attempts to create remote processes (for example via an exec to an ssh process), and of the parameters by which all sockets are created. In the case of sockets, DMTCP needs to know whether the sockets are TCP/IP sockets (and whether they are listener or non-listener sockets), UNIX domain sockets, or pseudo-terminals. DMTCP places wrappers around the following functions: socket, connect, bind, listen, accept, setsockopt, fexecve, execve, execv, execvp, fork, close, dup2, socketpair, openlog, syslog, closelog, ptsname and ptsname_r. The rest of this section describes the purposes for these wrapper.

4.3. Checkpointing under DMTCP

Checkpointing proceeds through seven stages and six global barriers. Global barriers could be implemented efficiently through peer-to-peer communication or broadcast trees, but are currently centralized for simplicity of implementation.

The following is the DMTCP distributed algorithm for checkpointing an entire cluster. It is executed asynchronously in each user process. The only communication primitive used is a cluster-wide barrier. The following steps are depicted graphically in Figure 1.

1) *Normal execution:* The checkpoint manager thread in each process waits until a new checkpoint is requested by the coordinator. This is done by waiting at a special barrier that is not released until checkpoint time.
2) *Suspend user threads:* MTCP suspends all user threads, then DMTCP saves the owner of each file descriptor. DMTCP then waits until all application processes reach Barrier 2: "suspended", then releases the barrier.
3) *Elect shard FD leaders:* DMTCP executes an election of a leader for each potentially shared file descriptor. We trick the operating system into electing a leader for us by misusing the F_SETOWN flag of `fcntl`. All processes set the owner, and the last one wins the election. In Step 4, a process can test if it is the election leader for a socket `fd` by testing if `fcntl(fd, F_GETOWN)==getpid()`. The original value for F_SETOWN is restored after kernel buffers are refilled. DMTCP then waits until all application processes reach Barrier 3: "election completed", then releases the barrier.
4) *Drain kernel buffers and perform handshakes with peers:* For each socket, the corresponding election leader flushes that socket by sending a special token. It then drains that socket by receiving until there is no more available data and the special token is seen. DMTCP then performs handshakes with all socket peers to discover the *globally unique ID* of the remote side of all sockets. The *connection information table* is then written to disk. DMTCP then waits until all application processes reach Barrier 4: "drained", then releases the barrier.

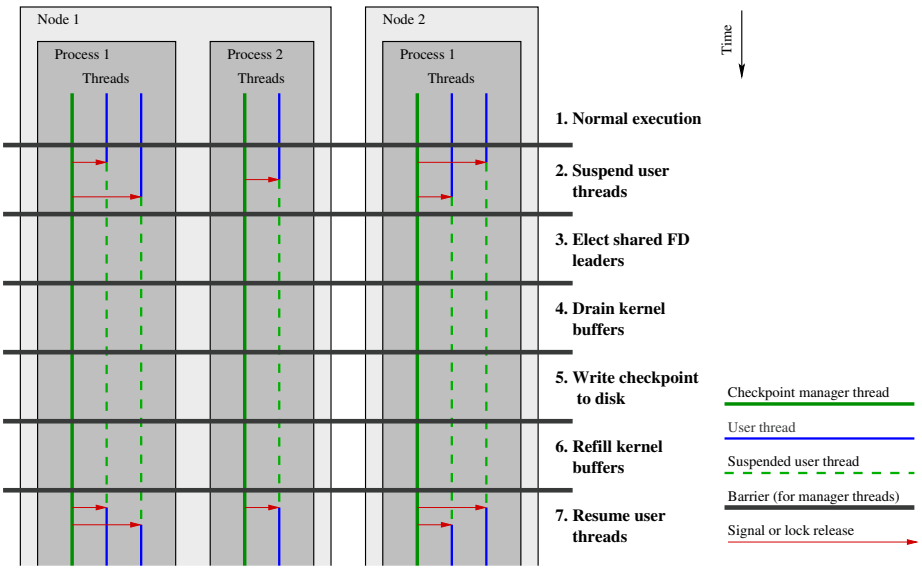


Figure 1: Steps for checkpointing a simple system with 2 nodes, 3 processes, and 5 threads.

5) *Write checkpoint to disk:* The contents of all socket buffers is now in user space. MTCP writes all of user space memory to the checkpoint file. DMTCP then wait until all application processes reach Barrier 5: "checkpointed", then release the barrier.
6) *Refill kernel buffers:* DMTCP then sends the drained socket buffer data back to the sender. The sender refills the kernel socket buffers by resending the data. DMTCP then waits until all application processes reach Barrier 6: "refilled", then releases the barrier.
7) *Resume user threads:* MTCP then resumes the application threads and DMTCP returns to Step 1.

4.4. Restart under DMTCP

The restart process undergoes some complexity in order to restore shared sockets. Under UNIX semantics multiple processes may share a single socket connection. When a process forks all open file descriptors become shared between the child and parent. To handle this, we refer to sockets by a *globally unique ID* (hostid, pid, timestamp, per-process connection number) and thus can detect duplicates at restart time. These globally unique socket IDs (and other meta information), were recorded at checkpoint time in the *connection information table* for each process. To recreated shared sockets, a *single* DMTCP restart process is created on each host. This single restart process will first restore all sockets, and then fork to create each individual user process on that host.

The following algorithm restarts the checkpointed cluster computation. It is executed asynchronously on each host in the cluster. The steps of this algorithm are depicted graphically in Figure 2.

1) *Reopen files and recreate ptys:* File descriptors, excluding sockets connected to a remote process, are regenerated first. These include files, listen sockets, uninitialized sockets, and pseudo-terminals.
2) *Recreate and reconnect sockets:* For each socket, the restart program uses the cluster-wide discovery service to find the new address of the corresponding restart process. Once the new addresses are found the connections are re-established. The discovery services is needed since processes may be relocated between checkpoint and restore.
3) *Fork into user processes:* The DMTCP restart program now forks into N processes, where N is the number of user processes it intends to restore.
4) *Rearrange FDs for user process:* Each of these processes uses `dup2` and `close` to re-arrange the file descriptors to reflect the arrangement prior to checkpointing. Unneeded file descriptors belonging to other processes are closed. Shared file descriptors will now exist in multiple processes.
5) *Restore memory and threads:* The MTCP restart routine is now called to restore the local process memory and threads. Upon completion the user process will resume at Barrier 5 of the checkpoint algorithm in Section 4.3
6) *Refill kernel buffers:* The program resumes as if it had just finished writing the checkpoint to disk, in Step 6 of checkpointing.

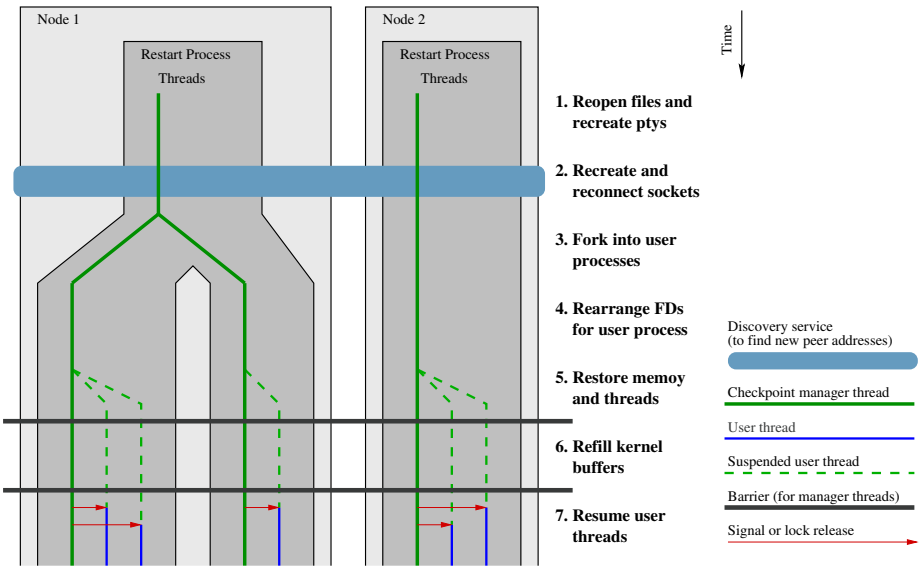


Figure 2: Steps for restarting the system checkpointed in Figure 1. The unified restart process and subsequent fork are required to recreate sockets and pipes shared between processes.

7) *Resume user threads:* The program continues executing Step 7 of checkpointing.

Step 2 above bears further explanation. Recall that prior to checkpointing, whenever a new connection was accepted, wrappers around the system calls `connect` and `accept` had transferred information about the `connect`or to the `accept`or. This information includes a globally unique socket ID that remains constant even if processes are relocated.

At restart time, the `accept`or for each socket advertises the address and port of its restart lister socket to the discovery service. When the `connect`or receives this advertisement, it opens a new connection to the `accept`or who sent the advertisement. The two sides then perform a handshake and agree on the socket being restored. Finally, `dup2` is used on each side to move the socket descriptor to the correct location. This process continues asynchronously until all sockets are restored. Our methodology supports both sides of a socket migrating. It also supports loopback sockets.

4.5. Implementation Strategies

In the implementation, some less obvious issues arise in the support for pipes, shared memory (via mmap), and virtual pids.

Pipes present an issue because they are unidirectional. As seen in Sections 4.3 and 4.4, the strategy for checkpointing network data in a socket connection is for the receiver to drain the socket into user space, then write a checkpoint image, and finally re-send the network data through the same socket back to the sender. In order to support pipes, a wrapper around the pipe system call promotes pipes into sockets.

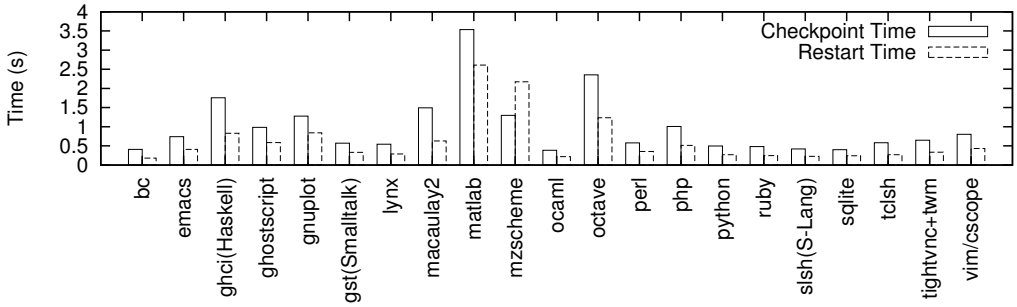
In the case of shared memory, if the backing file of a shared memory segment is missing and we have directory write permision, then we create a new backing file. Next, assuming the backing file is present and we have write access, we overwrite the shared memory segment with data from the checkpoint image. If two processes share this memory, they will both write to the same shared segment, but with the same data, since the segment was also shared at the time of checkpoint.

If we do not have write access (for example, read-only access to certain system-wide data), then we map the memory segment by the current data of the file, and not the checkpoint image data.

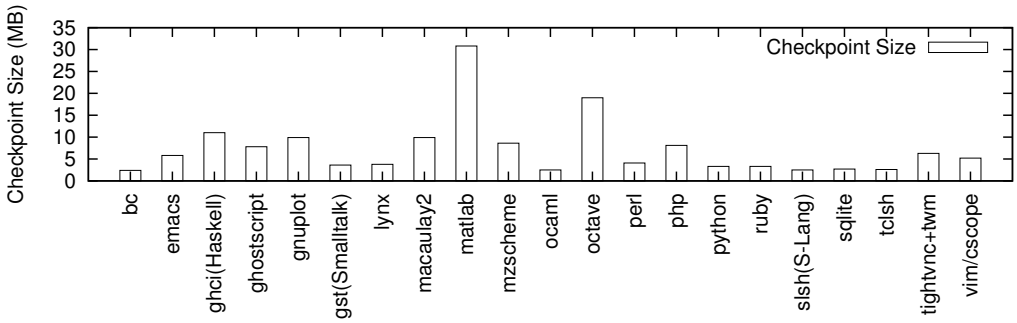
In order to support virtual pids (process ids), one must worry about pid conflicts. The original pid When a process is first created through a call to fork, its pid also becomes its virtual pid, and that virtual pid is maintained throughout succeeding generations of restarts. Hence, a new process may have pid A. After checkpoint and restart, a second process may be created with the same pid A. Our wrapper around fork detects this situation, terminates the child with the conflicting virtual pid,, and forks once again.

5. Experimental Results

DMTCP is currently implemented for GNU/Linux. The software has been verified to work on recent versions of Ubuntu, Debian, OpenSuse, and Red Hat Linux with Linux



(a) Checkpoint/Restart timings.



(b) Checkpoint sizes.

Figure 3: **Common shell-like languages** and other applications. All are run on a single node with compression enabled.

kernels ranging from version 2.6.9 through version 2.6.28. DMTCP runs on x86, x86_64 and mixed (32-bit processes in 64-bit Linux) architectures.

Experiments were run on two broad classes of programs: shell-like languages intended for a single computer (e.g. MATLAB, Perl, Python, Octave, etc.); and distributed programs across the nodes of a cluster (e.g. ParGeant4, iPython, MPICH2, OpenMPI, etc). Reported checkpoint images are after gzip compression (unless otherwise indicated), since DMTCP dynamically invokes gzip before saving, by default.

In Section 5.1, our goal was to demonstrate on 20 common real-world applications. An emphasis on shell-like languages were chosen for their widespread usage, and for their tendency to invoke multiple processes and multiple threads in their implementation. The languages were chosen from the applications listed under "Interactive mode languages" (shell-like languages) in the article "List of programming languages by category" on Wikipedia.

Section 5.2 is concerned with testing for scalability. The parallel tools and benchmarks were chosen for their popularity in the computational science community. They were augmented with some computational packages that had already been configured and installed as tools used by our own working group.

5.1. Common Shell-Like Languages

These tests were conducted on a dual-socket, quad-core (8 total cores) Xeon E5320. This system was running 64-bit Debian "sid" GNU/Linux with kernel version 2.6.28.

To show breadth, we present checkpoint times, restart times, and checkpoint sizes on a wide variety of commonly used applications. These results are shown in Figure 3. These applications are: BC (1.06.94) – an arbitrary precision calculator language; Emacs (2.22) – a well known text editor; GHCi (6.8.2) – the Glasgow Haskell Compiler; Ghostscript (8.62) – PostScript and PDF language interpreter; GNUPlot (4.2) – an interactive plotting program; GST (3.0.3) – the GNU Smalltalk virtual machine; Lynx (2.8.7) – a command line web browser; Macaulay2 (2-1.1) – a system supporting research in algebraic geometry and commutative algebra; MATLAB (7.4.0) – a high-level language and interactive environment for technical computing; MZScheme (4.0.1) – the PLT Scheme implementation; OCaml (3.10.2) – the Objective Caml interactive shell; Octave (3.0.1) – a high-level interactive language for numerical computations; PERL (5.10.0) – Practical Extraction and Report Language interpreter; PHP (5.2.6) – an HTML-embedded scripting language; Python (2.5.2) – an interpreted, interactive, object-

oriented programming language; Ruby (1.8.7) – an interpreted object-oriented scripting language; SLSH (0.8.2) – an interpreter for S-Lang scripts; SQLite (2.8.17) – a command line interface for the SQLite database; tclsh (8.4.19) – a simple shell containing the Tcl interpreter; TightVNC+TWM (1.3.9, 1.0.4) – a headless X11 server running Tab Window Manager underneath it; and vim/cscope (15.6) – interactively examine a C program.

Of particular interest is the checkpointing of TightVNC, a headless X11 server. We checkpoint the vncserver, the window manager, and all graphical applications. Between checkpoints, clients can connect with (uncheckpointed) vncviewers to interact with the graphical applications. Using this technique, we can checkpoint graphical applications without the need to checkpoint interactions with graphics hardware.

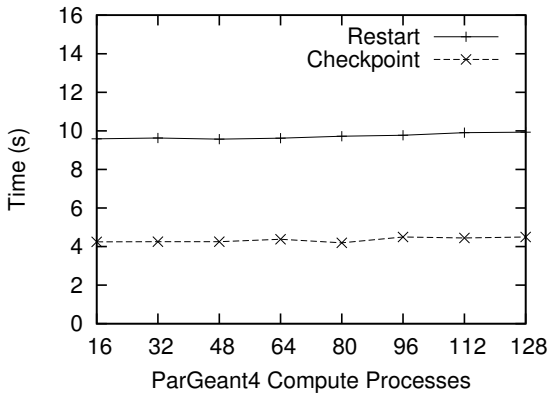
Additionally (not included in the graphs, because of differences in scale) we have demonstrated checkpointing of RunCMS. RunCMS checkpoints in 25.2 seconds and restarts in 18.4 seconds. RunCMS is of especially timely interest, with the recent startup of the large hadron collider at CERN. We are collaborating with the CMS experiment at CERN to checkpoint and restart their CMS software of up to two million lines of code and up to 700 dynamic libraries. We test on a configuration which grows to 680 MB of data after running for 12 minutes. At that time, it had loaded 540 dynamic libraries, as measured by the maps file of the proc filesystem. The checkpointed image file on disk was 225 MB, after gzip compression. (DMTCP invokes gzip compression by default.)

5.2. Distributed Applications

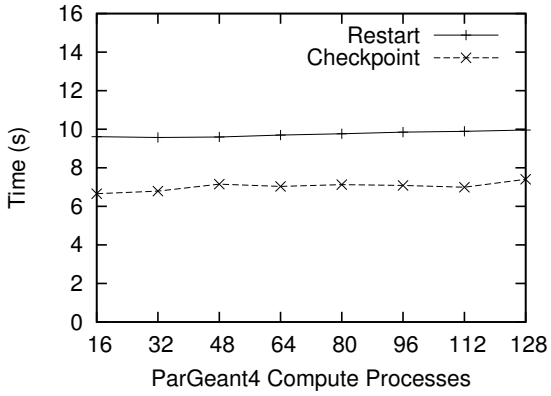
Distributed tests (Section 5.2) were conducted on a 32 node cluster with 4 cores per node (128 total cores). Each node was configured with dual-socket,dual-core Xeon 5130 processors and 8 GB or 16 GB of RAM. Each node was running 64-bit Red Hat Enterprise GNU/Linux release 4 with kernel version 2.6.9-34. The cluster was connected with Gigabit Ethernet.

In Figure 5b, checkpoints were written to a centralized EMC CX300 SAN storage device over a 4 Gbps Fibre Channel Switch. (SAN stands for storage area network.) On our cluster, only 8 of the 32 nodes were connected to the SAN. The remaining 24 nodes wrote indirectly to the storage device via NFS. In all other tests, checkpoints were written to local disk of each node.

We report checkpoint times, restart times, and checkpoint file sizes for a suite of distributed applications. These results are contained in Figures 4a, 4b and 4c, respectively. In each case, we report the timings and file sizes both with and without compression. The following applications are shown:



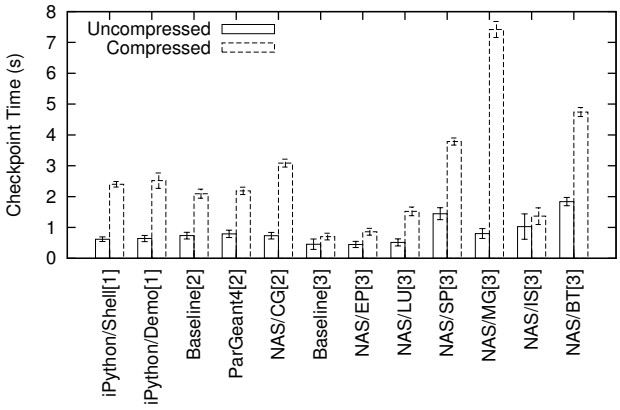
(a) Checkpoints stored to local disk of each node.



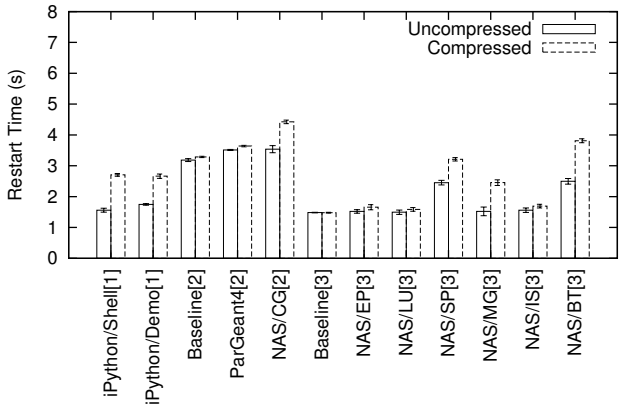
(b) Checkpoints stored to centralized RAID storage via SAN and NFS.

Figure 5: Timings as the number of processes and nodes changes. Application is ParGeant4 running under MPICH2. Compression is enabled. Compute processes per core and per node are held constant at 1 and 4, while the number of nodes is varied. (Note: An additional 21 to 161 MPICH2 resource management processes are also checkpointed.)

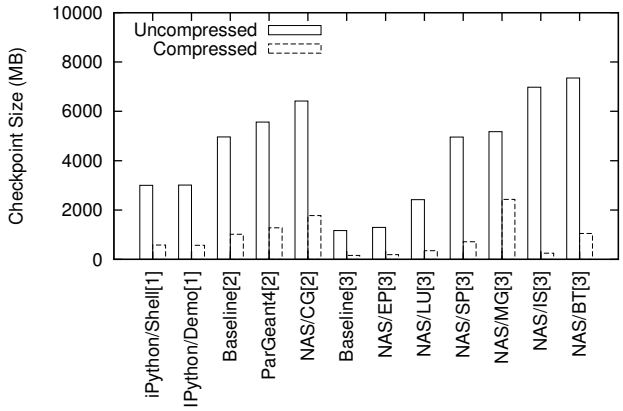
- **Based on sockets directly:**
  - iPython: [23] an enhanced Python shell with support for parallel and distributed computations. Used in scientific computations such as SciPy/NumPy. `iPython/Shell`: is the interactive iPython interpreter, idle at time of checkpoint. `iPython/Demo`: is the "parallel computing" demo included with the iPython tutorial.
- **Run using MPICH2 (1.0.5):**
  - `Baseline` is a "hello world" type application included to show the cost of checkpointing MPICH2 and its resource manger, MPD.
  - `ParGeant4`: Geant4 [11] is a million-line C++ toolkit for simulating particle-mattter interaction. It is based at CERN, where the largest particle collider in the world has been built. ParGeant4 [3]



(a) Checkpoint timings.



(b) Restart timings.



(c) Aggregate (cluster-wide) checkpoint size.

Figure 4: **Distributed Applications.** Timings on 32 nodes. Applications marked [1] use sockets directly. Applications marked [2] are run using MPICH2. Applications marked [3] are run using OpenMPI. Timing tests repeated 10 times, and the mean value is shown. Error bars in timings indicate plus or minus one standard deviation.

is a parallelization based on TOP-C, that is distributed with the Geant4 distribution. TOP-C (Task Oriented Parallel C/C++) was in turn built on top of MPICH2 for this demonstration.

- `NAS NPB2.4`: CG (Conjugate Gradient, level C) from the well-known benchmark suite NPB. NPB 2.4-MPI was used.

- **Run using OpenMPI (1.2.4):**
  - `Baseline` is a "hello world" type application included to show the cost of checkpointing OpenMPI and its resource manger, OpenRTE.
  - `NAS NPB2.4`: a series of well-known MPI benchmarks. NPB 2.4-MPI was used. The benchmarks run under OpenMPI are: BT (Block Tridiagonal, level C: 36 processes since the software requires a square number), SP (Scalar Pentadiagonal, level C: 36 processes since the software requires a square number), EP (Embarrassingly Parallel, level C), LU (Lower-Upper Symmetric Gauss-Seidel, level C), MG (Multi Grid, level C), and IS (Integer Sort, Level C).

In Figure 5a we use ParGeant4 as a test case to report on scalability with respect to the number of nodes. When resource management processes are included, we are checkpointing a total of 289 processes in the largest example. Figure 5b repeats this tests with checkpoints written to a centralized storage device.

Figure 6 illustrates the time as memory usage grows, while holding fixed the number of participating nodes at 32. The implied bandwidth is well beyond the typical 100 MB/s

of disk, and is presumably indicating the use of secondary storage cache in the Linux kernel. Restart times also indicate the use of cache and page table optimizations in the kernel.

An optional feature in DMTCP is to issue a `sync` after checkpointing to wait for kernel write buffers to empty before resuming the user threads. Results shown do not issue a call to `sync`. This is consistent with timing methodology most prevalent in related work. The cost of issuing a `sync` can be easily estimated based on checkpoint size and disk speed. As an example, if a `sync` is issued for ParGeant4 (compression enabled) a mean additional cost of 0.79 seconds (with a standard deviation of 0.24) is incurred. An alternate strategy is to sync the *previous* checkpoint instead. This has the benifits of still guaranteeing the consistency of all except the last checkpoint without having to wait for disk in most cases.

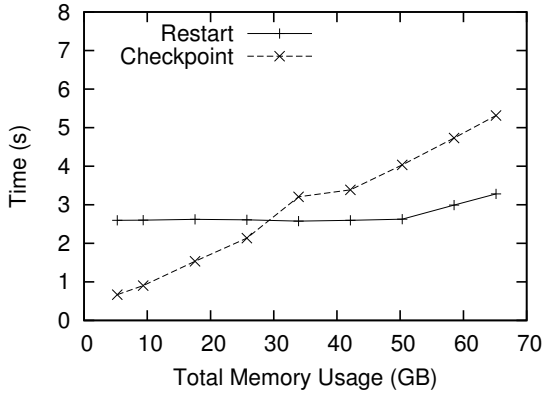


Figure 6: Timings as memory usage grows. A synthetic OpenMPI program allocating random data on 32 nodes. Compression is disabled. Checkpoints written to local disk.

5.3. Breakdown of Checkpoint/Restart Times

Table 1 shows the times for the different stages of checkpointing and restart. Checkpoint time is dominated by writing the checkpoint image to disk. This timing breakdown is typical for all other applications we examined. The time for writing the checkpoint image to disk is almost entirely eliminated by using the technique of forked checkpointing [20], [21]. In forked checkpointing, a child process is forked, the child process writes the checkpoint image, and the parent process continues to execute, taking advantage of UNIX copy-on-write-semantics. Forked checkpointing has the disadvantage that compression runs in parallel and may slow down the user process and take longer. The forked checkpointing times presented here are from an experimental version of DMTCP. Forked checkpointing was also supported in a forerunner of the current work [7].

Stage	Uncompressed	Compressed	Fork Compr.
Suspend user threads	0.0251	0.0217	0.0250
Elect FD leaders	0.0014	0.0013	0.0013
Drain kernel buffers	0.1019	0.1020	0.1017
Write checkpoint	0.6333	3.9403	0.0618
Refill kernel buffers	0.0006	0.0008	0.0016
Total	0.7630	4.0669	0.1922

(a) Checkpoint

Stage	Uncompressed	Compressed
Restore files and ptys	0.0056	0.0088
Reconnect sockets	0.0400	0.0214
Restore memory/threads	0.8139	2.1167
Refill kernel buffers	0.0009	0.0018
Total	0.8604	2.1487

(b) Restart

Table 1: Time (in seconds) for different stages of checkpoint and restart for NAS/MG under OpenMPI, using 8 nodes. Forked is the same as compressed, except that compression and writing are delegated to a child process and allowed to run in parallel.

The stages in Table 1a correspond to steps 2 through 6 in Figure 1. The stages in Table 1b correspond to steps 1 through 6 in Figure 2, except that steps 3 and 4, which take negligible time, are lumped in with step 5. Since the first 3 reported times for restart occur in parallel on each node, the reported times are an average across all 8 nodes. All other times are the durations between the global barriers.

5.4. Experimental Analysis

In principle, the time for checkpointing is dominated by: (i) compression (when enabled); (ii) checkpointing memory to disk; and (iii) (to a much lesser extent) flushing network data in transit and re-sending. When compression is enabled, that time dominates. The cost of flushing and re-sending is bounded above by the size of the corresponding kernel buffers and the capacity of the network switches, which tend to be on the order of tens of kilobytes. Restart tends to be faster than checkpoint, because `gunzip` operates more quickly than `gzip`.

The graphs in Figure 4 show that the time to checkpoint using compression tends to be slowest when the uncompressed checkpoint image is largest. An exception occurs for NAS/IS. NAS/IS is a parallel integer bucket-sort. The bucket sort code has allocated large buckets to guard against overflow. Presumably, the unwritten portion of the bucket is likely to be mostly zeroes, and it compresses both quickly and efficiently.

Figure 5 shows the time for checkpoint and restart to be insensitive to the number of nodes being used. This is to be expected since checkpointing on each node occurs asynchronously. It also demonstrates that the single checkpoint coordinator, which implements barriers, is not a bottleneck.

In the event that it were a bottleneck, we would replace it by a distributed coordinator in our implementation.

6. Conclusions and Future Work

A scalable approach to transparent distributed checkpointing has been demonstrated that does not depend on a specific message passing library. Nor does it depend on kernel modification. The approach achieves broad application coverage across a wide array both of scientific and common desktop applications. On 128 distributed cores (32 nodes), a typical checkpoint time is 2 seconds, or 0.2 seconds by using forked checkpointing, along with negligible run-time overhead. This makes DMTCP attractive both for frequent checkpointing and for minimal application interruptions during checkpointing of interactive applications. Experimental results have shown that the approach is scalable and that timings remain nearly constant as nodes are added to a computation within a medium-size cluster. The centralized checkpoint coordinator, which implements barriers, has minimal overhead in these experiments. As the approach is scaled to ever larger clusters, the single coordinator can be replaced by a distributed coordinator using well-known algorithms for distributed global barriers and distributed discovery services.

In the future, it is hoped to support new communication models such as multicast and RDMA (remote direct memory access) as used in networks such as InfiniBand. Future work will fully support the ptrace system call, and therefore checkpointing of gdb sessions. Future work will also extend the ability to checkpoint X-Windows applications, as currently demonstrated by the simple checkpointing of TightVNC. This will further enhance the attractiveness of user-level checkpointing.

7. Acknowledgements

We thank our colleagues at CERN who have discussed, helped debug, and tested the use of DMTCP on runCMS and on ParGeant4. In particular, we thank John Apostolakis, Giulio Eulisse and Lassi Tuura. We thank Xin Dong for his help in installing and testing ParGeant4 in a variety of operating circumstances. We acknowledge the gracious help of Michael Rieker in numerous discussions, and in fixing more than one bug in MTCP for us. We also acknowledge the help of Daniel Kunkle in testing checkpointing of TightVNC. Finally, we thank Peter Keller, David Wentzlaff, and Marek Olszewski for helpful comments on draft manuscripts.

The computational facilities for this work were partially supported by the National Science Foundation under Grant CNS-06-19616.

References

[1] Hazim Abdel-Shafi, Evan Speight, and John K. Bennett. Efficient user-level thread migration and checkpointing on Windows NT clusters. In *Usenix 1999 (3rd Windows NT Symposium)*, pages 1–10, 1999.

[2] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM Press.

[3] G. Alverson, L. Anchordoqui, G. Cooperman, V. Grinberg, T. McCauley, S. Reucroft, and J. Swain. Using TOP-C for commodity parallel computing in cosmic ray physics simulations. *Nuclear Physics B (Proc. Suppl.)*, 97:193–195, 2001.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *ACM/IEEE 2002 Conference on Supercomputing*. IEEE Press, 2002.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[6] P. Emerald Chung, Woei-Jyh Lee, Yennun Huang, Deron Liang, and Chung-Yih Wang. Winckp: A transparent checkpointing and rollback recovery tool for Windows NT applications. In *Proc. of 29th Annual International Symposium on Fault-Tolerant Computing*, pages 220–223, 1999.

[7] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, pages 283–291, Singapore, 2006. IEEE Press.

[8] William R. Dieter and James E. Lumpp Jr. User-level checkpointing for LinuxThreads programs. In *USENIX Annual Technical Conference (FREENIX Track)*, pages 81–92, 2001.

[9] DLL injection, Wikipedia. http://en.wikipedia.org/wiki/DLL_injection.

[10] Elmootazbellah Elnozahy and James Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[11] Geant4 Web page. http://wwwinfo.cern.ch/asd/geant4/geant4.html, 1999–.

[12] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 77–83, New York, NY, USA, 2002. ACM Press.

[13] Paul Hargrove and Jason Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics Conference Series*, 46:494–499, September 2006.

[14] Thomas Herault, Pierre Lemarinier, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault-tolerant MPI. In *Proceedings of International Symposium on High Performance Computing and Networking (SC2006)*, 2006.

[15] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdain. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS) / 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society, March 2007.

[16] G.J. Janakiraman, J.R. Santos, D. Subhraveti, and Y. Turner. Application-transparent distributed checkpoint-restart on standard operating systems. In *Dependable Systems and Networks (DSN-05)*, pages 260–269, 2005.

[17] Byoung-Jip Kim. Comparison of the existing checkpoint systems. Technical report, IBM Watson, October 2005.

[18] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes for commodity clusters. In *2007 USENIX Annual Technical Conference*, pages 323–336, 2007.

[19] Oren Laadan, Dan Phung, and Jason Nieh. Transparent networked checkpoint-restart for commodity clusters. In *2005 IEEE International Conference on Cluster Computing*. IEEE Press, 2005.

[20] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proc. of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[21] Kai Li, Jeffrey F. Naughton, and James S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5:874–879, August 1994.

[22] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical report 1346, University of Wisconsin, Madison, Wisconsin, April 1997.

[23] Fernando Perez and Brian E. Granger. iPython: A system for interactive scientific computing. *Computing in Science and Engineering*, pages 21–29, May/June 2007. (See, also, http://ipython.scipy.org/moin/Parallel_Computing.).

[24] Eduardo Pinheiro. EPCKPT — a checkpoint utility for the Linux kernel. http://www.research.rutgers.edu/~edpin/epckpt/.

[25] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[26] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proc. of the USENIX Winter 1995 Technical Conference*, pages 213–323, 1995.

[27] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the Native POSIX Thread Library for Linux. In *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA-06)*, pages 492–498, 2006.

[28] Eric Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, November 2003.

[29] Joseph Ruscio, Michael Heffner, and Srinidhi Varadarajan. DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[30] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19:479–493, 2005.

[31] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual International Symposium on Computer Architecture*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[32] Georg Stellner. Cocheck: Checkpointing and process migration for MPI. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

[33] O.O. Sudakov, I.S. Meshcheriakov, and Y.V. Boyko. CHPOX: Transparent checkpointing system for Linux clusters. In *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 159–164, 2007.

[34] Namyoon Woo, Soonho Choi, hyungsoo Jung, Jungwhan Moon, Heon Y. Yeom, Taesoon Park, and Hyungwoo Park. MPICH-GF: Providing fault tolerance on Grid environments. The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), the poster and research demo session May, 2003, Tokyo, Japan.

[35] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In 8^{th} *IEEE International Symposium on High Performance Distributed Computing*, pages 177–184, 1999.

[36] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level checkpoint and recovery for LAM/MPI. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 72 – 81, 2005.

[37] Gengbin Zheng, Lixia Shi, and L.V. Kale. FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing (Fault-Tolerant Session)*, pages 93–103, 2004.