DMTCP
Transparent Checkpointing for Cluster Computations and the Desktop

Jason Ansel\textsuperscript{1}  Kapil Arya\textsuperscript{2}  Gene Cooperman\textsuperscript{2}

\textsuperscript{1}MIT
\textsuperscript{2}Northeastern University

May 26, 2009
Outline

1. Introduction
   - Background
   - Motivation
   - Related work
   - Short Demo

2. Design and Implementation
   - How it works
   - Distributed checkpointing algorithm
   - Other features

3. Results
   - Performance trends
   - Benchmarks

4. Conclusions
   - Final remarks
   - Questions
What is DMTCP / checkpointing?

- We present DMTCP: Distributed MultiThreaded CheckPointing
What is DMTCP / checkpointing?

- We present DMTCP: Distributed MultiThreaded Checkpointing
- Checkpointing is taking a snapshot of an applications state that can later be restarted
What is DMTCP / checkpointing?

- We present DMTCP: **Distributed MultiThreaded CheckPointing**
- Checkpointing is taking a snapshot of an applications state that can later be restarted
- DMTCP is
  - **distributed** - can checkpoint a network of programs connected by sockets
What is DMTCP / checkpointing?

- We present DMTCP: Distributed MultiThreaded CheckPointing
- Checkpointing is taking a snapshot of an applications state that can later be restarted
- DMTCP is
  - distributed - can checkpoint a network of programs connected by sockets
  - multithreaded - each program can have many threads
What is DMTCP / checkpointing?

- We present DMTCP: **Distributed MultiThreaded Checkpointing**
- Checkpointing is taking a snapshot of an applications state that can later be restarted
- DMTCP is
  - **distributed** - can checkpoint a network of programs connected by sockets
  - **multithreaded** - each program can have many threads
  - **transparent** - works on unmodified binaries
What is DMTCP / checkpointing?

- We present DMTCP: Distributed MultiThreaded Checkpointing
- Checkpointing is taking a snapshot of an applications state that can later be restarted
- DMTCP is
  - **distributed** - can checkpoint a network of programs connected by sockets
  - **multithreaded** - each program can have many threads
  - **transparent** - works on unmodified binaries
  - **user-level** - kernel is not modified
The traditional motivation for checkpointing

- Long running computation on a large cluster
- Computation takes 30 days
- On day 29...
The traditional motivation for checkpointing

- Long running computation on a large cluster
- Computation takes 30 days
- On day 29... a node crashes. Disaster!!!
- Must restart from the beginning
The traditional motivation for checkpointing

- Long running computation on a large cluster
- Computation takes 30 days
- On day 29... a node crashes. Disaster!!!
- Must restart from the beginning
- Restart from the last checkpoint
The traditional motivation for checkpointing

- Long running computation on a large cluster
- Computation takes 30 days
- On day 29... a node crashes. Disaster!!!
- Must restart from the beginning
- Restart from the last checkpoint
- Gives fault tolerance with no programmer support
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released
- Difficulty in checkpointing is robustness
- Going from checkpointing **one** application to **most**:
  - A four year effort
  - Now about 10 developers
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released
- Difficulty in checkpointing is robustness
- Going from checkpointing one application to most:
  - A four year effort
  - Now about 10 developers
- Exception: BLCR
  - Also works for most applications (though fails on many of our benchmarks)
  - Kernel level
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released

- Difficulty in checkpointing is robustness

- Going from checkpointing **one** application to **most**:
  - A four year effort
  - Now about 10 developers

- Exception: BLCR
  - Also works for most applications (though fails on many of our benchmarks)
  - Kernel level
    - Can’t bundle with application
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released
- Difficulty in checkpointing is robustness
- Going from checkpointing one application to most:
  - A four year effort
  - Now about 10 developers
- Exception: BLCR
  - Also works for most applications (though fails on many of our benchmarks)
  - Kernel level
    - Can’t bundle with application
    - Harder to maintain
Haven’t we heard of checkpointing before?

- Surveying existing checkpointing systems:
  - Most don’t work
  - Others have never been released
- Difficulty in checkpointing is robustness
- Going from checkpointing **one** application to **most**:
  - A four year effort
  - Now about 10 developers
- Exception: BLCR
  - Also works for most applications (though fails on many of our benchmarks)
  - Kernel level
    - Can’t bundle with application
    - Harder to maintain
  - Doesn’t support sockets
  - Distributed support (with customized MPI libraries) less robust
Related work

- **Kernel level**
  - Berkeley Lab Checkpoint/Restart (BLCR)
    - Doesn’t support sockets
    - Open source

- **User level**
  
  - DMTCP (our system)
    - Distributed/multithreaded
    - Open Source
Related work

- Kernel level
  - Berkeley Lab Checkpoint/Restart (BLCR)
    - Doesn’t support sockets
    - Open source
  - Zap (from Columbia University)
    - Distributed/multithreaded
    - Closed source, not publicly available

- User level
  - Deja Vu (from Virginia Tech)
    - Distributed/multithreaded
    - Closed source, not publicly available
  
  - DMTCP (our system)
    - Distributed/multithreaded
    - Open Source
Related work

- **Kernel level**
  - Berkeley Lab Checkpoint/Restart (BLCR)
    - Doesn’t support sockets
    - Open source
  - Zap (from Columbia University)
    - Distributed/multithreaded
    - Closed source, not publicly available

- **User level**
  - Deja Vu (from Virginia Tech)
    - Distributed/multithreaded
    - Closed source, not publicly available
    - Reported overheads 97x slower for a benchmark of similar scale
  - DMTCP (our system)
    - Distributed/multithreaded
    - Open Source
Other uses for checkpointing

- Fault tolerance
- Process migration
- Replacement for save/restore workspace
- Skip past long startup times
- Debugging
- Ultimate bug report
- Speculative execution
Short Demo
Outline

1. Introduction
   - Background
   - Motivation
   - Related work
   - Short Demo

2. Design and Implementation
   - How it works
   - Distributed checkpointing algorithm
   - Other features

3. Results
   - Performance trends
   - Benchmarks

4. Conclusions
   - Final remarks
   - Questions
Gaining initial control

- Dynamic library injection (LD_PRELOAD) to force the user application to load dmtcphijack.so
Gaining initial control

- Dynamic library injection (LD_PRELOAD) to force the user application to load dmtcp Hijack.so
- A **checkpointing manager thread** is spawned in each process
Gaining initial control

- Dynamic library injection (LD_PRELOAD) to force the user application to load dmtcphijack.so
- A **checkpointing manager thread** is spawned in each process
- Additional forked processes are hijacked recursively
- Remote process (spawned with ssh) are detected and hijacked
Gaining initial control

- Dynamic library injection (LD_PRELOAD) to force the user application to load dmtcp_hijack.so
- A **checkpointing manager thread** is spawned in each process
- Additional forked processes are hijacked recursively
- Remote process (spawned with `ssh`) are detected and hijacked
- **The result**: our library and checkpoint manger thread in every user process
Saving program state

1. User space memory
2. Processor state
3. Data in network
4. Kernel state
Saving program state

1. User space memory - read from checkpoint management thread
2. Processor state
3. Data in network
4. Kernel state
Saving program state

1. **User space memory** - read from checkpoint management thread
2. **Processor state** - hijack user threads and copy to memory
3. **Data in network**
4. **Kernel state**
Saving program state

1. **User space memory** - read from checkpoint management thread
2. **Processor state** - hijack user threads and copy to memory
3. **Data in network** - drained to process memory
4. **Kernel state**
Saving program state

1. **User space memory** - read from checkpoint management thread
2. **Processor state** - hijack user threads and copy to memory
3. **Data in network** - drained to process memory
4. **Kernel state** - probing at checkpoint time
Saving program state

1. **User space memory** - read from checkpoint management thread
2. **Processor state** - hijack user threads and copy to memory
3. **Data in network** - drained to process memory
4. **Kernel state** - probing at checkpoint time
   - Memory Maps – `/proc` filesystem
   - File descriptors (files) – `/proc` filesystem, `fstat`, etc
   - File descriptors (sockets, pipes, pts, etc) – `/proc` filesystem, `getsockopt`, wrappers around creation functions
   - Other information (signal handlers, etc) – POSIX API
Our checkpointing algorithm

- Distributed algorithm
- Only global communication is a barrier
- Coordinated / “stop the world” style checkpointing
Checkpointing algorithm, by example

Running normally, wait for checkpoint to begin
Checkpointing algorithm, by example

Suspend user threads, barrier
Checkpointing algorithm, by example

Suspend user threads, barrier
Checkpointing algorithm, by example

Elect shared resource leaders, barrier
Checkpointing algorithm, by example

Elect shared resource leaders, barrier
Checkpointing algorithm, by example

Drain socket data, barrier
Checkpointing algorithm, by example

Drain socket data, barrier
Checkpointing algorithm, by example

Perform single process checkpointing, barrier
Checkpointing algorithm, by example

Perform single process checkpointing, barrier

Node 1
- Process A

Node 2
- Process B
- Process C

Node 3
- Process D

Socket

DMTCP Control

User Control

Socket Data
Checkpointing algorithm, by example

Refill socket data, barrier

Node 1
Process A

Socket

Node 2
Process B
Process C

Shared Socket

Node 3
Process D

User Control
DMTCP Control
Socket Data

DMTCP
May 26, 2009 13 / 39
Checkpointing algorithm, by example

Refill socket data, barrier
Checkpointing algorithm, by example

Refill socket data, barrier
Checkpointing algorithm, by example

Resume user threads
Checkpointering algorithm, by example

Running normally

Node 1
- Process A
- Socket

Node 2
- Process B
- Process C
- Shared Socket

Node 3
- Process D
- Shared Socket

User Control
DMTCP Control
Socket Data

Jason Ansel (MIT)
DMTCP
May 26, 2009
13 / 39
Restart algorithm, by example

Start with nothing (possibly different nodes)
Restart algorithm, by example

Restart process on each node

Node 1
- Restart

Node 2
- Restart

Node 3
- Restart

Legend:
- User Control
- DMTCP Control
- Socket Data
Restart algorithm, by example

Recreate files, sockets, etc

Node 1

Restart

Node 2

Restart

Node 3

Restart

User Control  DMTCP Control  Socket Data
Restart algorithm, by example

Recreate files, sockets, etc

Node 1
Restart

Node 2
Restart

Node 3
Restart

User Control
DMTCP Control
Socket Data
Restart algorithm, by example

Fork user processes

Node 1
Restart

Node 2
Restart

Node 3
Restart

User Control
DMTCP Control
Socket Data
Restart algorithm, by example

Fork user processes

Node 1
Restart

Node 2
Restart
Restart

Node 3
Restart

User Control
DMTCP Control
Socket Data
Restart algorithm, by example

Rearrange FDs to match each user process
Restart algorithm, by example

Rearrange FDs to match each user process

- Node 1
- Node 2
- Node 3

DMTCP Control
User Control
Socket Data
Restart
Restart
Restart
Restart

Jason Ansel (MIT)
Restart algorithm, by example

Restore memory/threads
Restart algorithm, by example

Restore memory/threads

Node 1
- Process A

Node 2
- Process B
- Process C

Node 3
- Process D

- User Control
- DMTCP Control
- Socket Data
Restart algorithm, by example

Continue as if after a checkpoint
Restart algorithm, by example

Continue as if after a checkpoint
Restart algorithm, by example

Continue as if after a checkpoint

Node 1
- Process A

Node 2
- Process B
- Process C

Node 3
- Process D

Socket

DMTCP Control
User Control
Socket Data
Other features supported by DMTCP

- Threads, mutexes/semaphores, fork, exec, ssh
- Shared memory (between processes)
- TCP/IP sockets, UNIX domain sockets, pipes
- Pseudo terminals, terminal modes, ownership of controlling terminals
- Signals and signal handlers
- I/O (including the readline library), shared fds
- Parent-child process relationships, process id & thread id virtualization, session and process group ids
- Syslogd, vdso
- Address space randomization, exec shield
- Checkpoint image compression, forked checkpointing
- ...

Jason Ansel (MIT)
DMTCP
May 26, 2009 15 / 39
Other features supported by DMTCP

- Threads, mutexes/semaphores, fork, exec, ssh
- Shared memory (between processes)
- TCP/IP sockets, UNIX domain sockets, pipes
- Pseudo terminals, terminal modes, ownership of controlling terminals
- Signals and signal handlers
- I/O (including the readline library), shared fds
- Parent-child process relationships, process id & thread id
  virtualization, session and process group ids
- Syslogd, vdso
- Address space randomization, exec shield
- Checkpoint image compression, forked checkpointing
- ...
Other features supported by DMTCP

- Threads, mutexes/semaphores, fork, exec, ssh
- Shared memory (between processes)
- TCP/IP sockets, UNIX domain sockets, pipes
- Pseudo terminals, terminal modes, ownership of controlling terminals
- Signals and signal handlers
- I/O (including the readline library), shared fds
- Parent-child process relationships, process id & thread id
  virtualization, session and process group ids
- Syslogd, vdso
- Address space randomization, exec shield
- Checkpoint image compression, forked checkpointing
- ...

Jason Ansel (MIT)
Other features supported by DMTCP

- Threads, mutexes/semaphores, fork, exec, ssh
- Shared memory (between processes)
- TCP/IP sockets, UNIX domain sockets, pipes
- Pseudo terminals, terminal modes, ownership of controlling terminals
- Signals and signal handlers
- I/O (including the readline library), shared fds
- Parent-child process relationships, process id & thread id
- Virtualization, session and process group ids
- Syslogd, vdso
- Address space randomization, exec shield
- Checkpoint image compression, forked checkpointing
- ...

Jason Ansel (MIT)
Pseudo terminals

- Example execution:

  Process 1 opens /dev/ptmx
  Process 1 calls ptsname() on the FD
  Returns the string "/dev/pts/7"
  String copied and shared...
  At restart time /dev/pts/7 is in use!!!

Problem: we can't change the string hidden in user memory

Solution: virtualize in a sneaky way
ptsname() returns /tmp/unique
/tmp/unique is a symlink to /dev/pts/7
At restart time we can redirect /tmp/unique to an available device
Example execution:
- Process 1 opens /dev/ptmx
- Process 1 calls ptsname() on the FD
  - Returns the string "/dev/pts/7"
- String copied and shared
Example execution:
- Process 1 opens /dev/ptmx
- Process 1 calls ptsname() on the FD
  - Returns the string "/dev/pts/7"
- String copied and shared
- ...
- At restart time /dev/pts/7 is in use!!!
- Problem: we can’t change the string hidden in user memory
Pseudo terminals

- Example execution:
  - Process 1 opens `/dev/ptmx`
  - Process 1 calls `ptsname()` on the FD
    - Returns the string "/dev/pts/7"
  - String copied and shared
  - ...
  - At restart time /dev/pts/7 is in use!!!
  - Problem: we can’t change the string hidden in user memory
- Solution: virtualize in a sneaky way
Pseudo terminals

- Example execution:
  - Process 1 opens /dev/ptmx
  - Process 1 calls ptsname() on the FD
    - Returns the string "/dev/pts/7"
  - String copied and shared
  - ...
  - At restart time /dev/pts/7 is in use!!!
  - Problem: we can’t change the string hidden in user memory

- Solution: virtualize in a sneaky way
  - ptsname() returns /tmp/unique
Pseudo terminals

- **Example execution:**
  - Process 1 opens `/dev/ptmx`
  - Process 1 calls `ptsname()` on the FD
    - Returns the string "/dev/pts/7"
  - String copied and shared
  - ...
  - At restart time `/dev/pts/7` is in use!!!
  - Problem: we can’t change the string hidden in user memory

- **Solution: virtualize in a sneaky way**
  - `ptsname()` returns `/tmp/unique`
  - `/tmp/unique` is a symlink to `/dev/pts/7`
Pseudo terminals

- **Example execution:**
  - Process 1 opens `/dev/ptmx`
  - Process 1 calls `ptsname()` on the FD
    - Returns the string `"/dev/pts/7"
  - String copied and shared
  - ...
  - At restart time `/dev/pts/7` is in use!!!
  - Problem: we can’t change the string hidden in user memory

- **Solution: virtualize in a sneaky way**
  - `ptsname()` returns `/tmp/unique`
  - `/tmp/unique` is a symlink to `/dev/pts/7`
  - At restart time we can redirect `/tmp/unique` to an available device
Checkpoint image compression

Three checkpointing modes:

1. Uncompressed (normal) checkpoints

- Faster
- Normal
- Forked checkpointing (completed in parallel to user application)

- Smaller
- Calls "gzip –fast" as a filter

On our distributed benchmarks:
- 2.1x to 28.0x (mean 7.3x) compression
Checkpoint image compression

Three checkpointing modes:

1. Uncompressed (normal) checkpoints
2. Compressed checkpoints
   - Calls "gzip –fast" as a filter
   - On our distributed benchmarks:
     2.1x to 28.0x (mean 7.3x) compression
Checkpoint image compression

Three checkpointing modes:

1. Uncompressed (normal) checkpoints
2. Compressed checkpoints
   - Calls “gzip –fast” as a filter
   - On our distributed benchmarks: 2.1x to 28.0x (mean 7.3x) compression
3. Forked checkpointing
   - Completed in parallel to user application
Outline

1. Introduction
   - Background
   - Motivation
   - Related work
   - Short Demo

2. Design and Implementation
   - How it works
   - Distributed checkpointing algorithm
   - Other features

3. Results
   - Performance trends
   - Benchmarks

4. Conclusions
   - Final remarks
   - Questions
Compression enabled. ParGeant4 benchmark.
4 nodes through 32 nodes × 4 cores per node.
What controls checkpoint time?

- With compression:
  - $\text{time(checkpoint)} \approx \text{time(gzip memory)}$
  - In parallel across cluster
What controls checkpoint time?

- With compression:
  - \( \text{time}(\text{checkpoint}) \approx \text{time}(\text{gzip memory}) \)
  - In parallel across cluster

<table>
<thead>
<tr>
<th>Stage</th>
<th>Compressed</th>
<th>Uncompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend user threads</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>Elect FD leaders</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Drain kernel buffers</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>Write checkpoint</td>
<td>3.94</td>
<td>0.63</td>
</tr>
<tr>
<td>Refill kernel buffers</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>4.07</td>
<td></td>
</tr>
</tbody>
</table>

NAS/MG benchmark with 32 compute processes on 8 nodes
What controls checkpoint time?

- With compression:
  - \( \text{time(checkpoint)} \approx \text{time(gzip memory)} \)
  - In parallel across cluster

- Without compression, dominated by writing to disk

<table>
<thead>
<tr>
<th>Stage</th>
<th>Compressed</th>
<th>Uncompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend user threads</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>Elect FD leaders</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Drain kernel buffers</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>Write checkpoint</td>
<td>3.94</td>
<td>0.63</td>
</tr>
<tr>
<td>Refill kernel buffers</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>4.07</td>
<td>0.76</td>
</tr>
</tbody>
</table>

NAS/MG benchmark with 32 compute processes on 8 nodes
Benchmarks Overview

- Distributed benchmarks (10 benchmarks)
  - Run on a 32 node (128 core) cluster
Benchmarks Overview

- Distributed benchmarks (10 benchmarks)
  - Run on a 32 node (128 core) cluster
- Single node benchmarks (20 benchmarks)
  - Run on an 8 core machine
  - Some, not all, are multithreaded/multiprocess
Distributed benchmarks

- Based on sockets directly:

- Run using MPICH2:

- Run using OpenMPI:
Distributed benchmarks

- Based on sockets directly:
  - iPython/Shell and iPython/Demo: parallel/distributed python shell
- Run using MPICH2:

- Run using OpenMPI:
Distributed benchmarks

- **Based on sockets directly:**
  - iPython/Shell and iPython/Demo: parallel/distributed python shell

- **Run using MPICH2:**
  - Baseline

- **Run using OpenMPI:**
  - Baseline
Distributed benchmarks

- **Based on sockets directly:**
  - iPython/Shell and iPython/Demo: parallel/distributed python shell

- **Run using MPICH2:**
  - Baseline
  - ParGeant4: a million-line C++ toolkit for simulating particle-matter interaction.

- **Run using OpenMPI:**
  - Baseline
Distributed benchmarks

- **Based on sockets directly:**
  - iPython/Shell and iPython/Demo: parallel/distributed python shell

- **Run using MPICH2:**
  - Baseline
  - ParGeant4: a million-line C++ toolkit for simulating particle-matter interaction.
  - NAS NPB2.4: CG (Conjugate Gradient)

- **Run using OpenMPI:**
  - Baseline
  - NAS NPB2.4: BT (Block Tridiagonal), SP (Scalar Pentadiagonal), EP (Embarrassingly Parallel), LU (Lower-Upper Symmetric Gauss-Seidel), MG (Multi Grid), and IS (Integer Sort).
Single node benchmarks

- **Scripting languages:**
  - **BC** – an arbitrary precision calculator language
  - **GHCi** – the Glasgow Haskell Compiler
  - **Ghostscript** – PostScript and PDF language interpreter
  - **GNUPlot** – an interactive plotting program
  - **GST** – the GNU Smalltalk virtual machine
  - **Macaulay2** – a system supporting research in algebraic geometry and commutative algebra
  - **MATLAB** – a high-level language and interactive environment for technical computing
  - **MZScheme** – the PLT Scheme implementation
  - **OCaml** – the Objective Caml interactive shell
  - **Octave** – a high-level interactive language for numerical computations
  - **PERL** – Practical Extraction and Report Language interpreter
Single node benchmarks (continued)

- Scripting languages (continued):
  - **PHP** – an HTML-embedded scripting language
  - **Python** – an interpreted, interactive, object-oriented programming language
  - **Ruby** – an interpreted object-oriented scripting language
  - **SLSH** – an interpreter for S-Lang scripts
  - **tclsh** – a simple shell containing the Tcl interpreter
Scripting languages (continued):

- **PHP** – an HTML-embedded scripting language
- **Python** – an interpreted, interactive, object-oriented programming language
- **Ruby** – an interpreted object-oriented scripting language
- **SLSH** – an interpreter for S-Lang scripts
- **tclsh** – a simple shell containing the Tcl interpreter

Other programs:

- **Emacs** – a well known text editor
- **vim/cscope** – interactively examine a C program.
Single node benchmarks (continued)

- Scripting languages (continued):
  - **PHP** – an HTML-embedded scripting language
  - **Python** – an interpreted, interactive, object-oriented programming language
  - **Ruby** – an interpreted object-oriented scripting language
  - **SLSH** – an interpreter for S-Lang scripts
  - **tclsh** – a simple shell containing the Tcl interpreter

- Other programs:
  - **Emacs** – a well known text editor
  - **vim/cscope** – interactively examine a C program.
  - **Lynx** – a command line web browser
  - **SQLite** – a command line interface for the SQLite database
  - **tightvnc/twm** – headless X server and window manager
Scripting languages (continued):
- **PHP** – an HTML-embedded scripting language
- **Python** – an interpreted, interactive, object-oriented programming language
- **Ruby** – an interpreted object-oriented scripting language
- **SLSH** – an interpreter for S-Lang scripts
- **tclsh** – a simple shell containing the Tcl interpreter

Other programs:
- **Emacs** – a well known text editor
- **vim/cscope** – interactively examine a C program.
- **Lynx** – a command line web browser
- **SQLite** – a command line interface for the SQLite database
- **tightvnc/twm** – headless X server and window manager
- **RunCMS** – Simulation of the CMS experiment at LHC/CERN
Single node benchmarks (continued)

- **Scripting languages (continued):**
  - **PHP** – an HTML-embedded scripting language
  - **Python** – an interpreted, interactive, object-oriented programming language
  - **Ruby** – an interpreted object-oriented scripting language
  - **SLSH** – an interpreter for S-Lang scripts
  - **tclsh** – a simple shell containing the Tcl interpreter

- **Other programs:**
  - **Emacs** – a well known text editor
  - **vim/cscope** – interactively examine a C program.
  - **Lynx** – a command line web browser
  - **SQLite** – a command line interface for the SQLite database
  - **tightvnc/twm** – headless X server and window manager
  - **RunCMS** – Simulation of the CMS experiment at LHC/CERN
RunCMS Benchmark

- RunCMS benchmark
  - Developed at CERN
  - Simulates the CMS experiment of the large hadron collider (LHC)
  - 2 million lines of code
  - 700 dynamic libraries
  - 12 minute startup time

- Checkpoint time (with compression) is 25.2 seconds
- Restart time is 18.4 seconds
- 680MB memory image, compressed to 225MB
Outline

1. Introduction
   - Background
   - Motivation
   - Related work
   - Short Demo

2. Design and Implementation
   - How it works
   - Distributed checkpointing algorithm
   - Other features

3. Results
   - Performance trends
   - Benchmarks

4. Conclusions
   - Final remarks
   - Questions

Jason Ansel (MIT)
Future work

- Integration with Condor
  - Condor is a ground breaking process migration system
  - Based on its own single-process checkpointeer
    - Requires relinking.
    - Doesn’t support: threads, multiple processes, mmap, etc.
Future work

- Integration with Condor
  - Condor is a ground breaking process migration system
  - Based on its own single-process checkpointer
    - Requires relinking.
    - Doesn’t support: threads, multiple processes, mmap, etc.
  - DMTCP will remove these limitations
Future work

- Integration with Condor
  - Condor is a ground breaking process migration system
  - Based on its own single-process checkpointing
    - Requires relinking.
    - Doesn’t support: threads, multiple processes, mmap, etc.
  - DMTCP will remove these limitations
  - Hope to release an experimental beta version by end of summer
Future work

- Integration with Condor
  - Condor is a ground breaking process migration system
  - Based on its own single-process checkpointer
    - Requires relinking.
    - Doesn’t support: threads, multiple processes, mmap, etc.
  - DMTCP will remove these limitations
  - Hope to release an experimental beta version by end of summer

- DMTCP as a save/restore workspace feature in SCIRun
  - Computational workbench
  - Visual programming
  - For modelling, simulation and visualization
  - Millions of lines of code

- Improving support for X windows applications
MTCP (our single-process component):
  - Michael Rieker

Colleagues at U Wisconsin (integration with Condor):
  - Peter Keller and others

Colleagues at CERN (help with runCMS, ParGeant4):
  - John Apostolakis, Giulio Eulisse, Lassi Tuura, and others

Other DMTCP developers / contributors:
  - Alex Brick, Tyler Deniseton Xin Dong, Daniel Kunkle Artem Polyakov, Praveen Solanki, and Ana-Maria Visan
For more information

- Source code (LGPL), documentation, other publications:
- http://dmtcp.sourceforge.net/

- Questions?
Thank you
Usage

1. Start your program under DMTCP:
   
   ```
   dmtcpCheckpoint [options] <program>
   ```
   
   For example:
   
   ```
   dmtcpCheckpoint mpdboot -n 32
   dmtcpCheckpoint mpirun -np 32 hellompi
   ```
Usage

1. Start your program under DMTCP:
   
   ```
   dmtcp_checkpoint [options] <program>
   ```
   
   For example:
   
   ```
   dmtcp_checkpoint mpdboot -n 32
   dmtcp_checkpoint mpirun -np 32 hellompi
   ```

2. Request a checkpoint
   
   ```
   dmtcp_command --checkpoint
   ```
Usage

1. Start your program under DMTCP:
   
   ```
dmtcp_checkpoint [options] <program>
   
   For example:
   
dmtcp_checkpoint mpdboot -n 32
   dmtcp_checkpoint mpirun -np 32 hellompi
   ```

2. Request a checkpoint
   
   ```
dmtcp_command --checkpoint
   ```

3. Restart:
   
   ```
./dmtcp_restart_script.sh
   ```
MultiThreaded CheckPointing (MTCP)

- MTCP is our single process checkpointing component
- Separate/modular so that it can be swapped out (when porting)
- Requires its own talk to properly describe

See our past publication:
Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux.
Michael Rieker, Jason Ansel, and Gene Cooperman.
Distributed benchmark timings

![Graph showing checkpoint time for different benchmarks and compression types.]

- Uncompressed vs. Compressed
- Benchmarks: iPython/Shell, iPython/Demo, Baseline, ParGeant4, NAS/CG, NAS/LU, NAS/SP, NAS/MG, NAS/IS, NAS/BT

![Graph showing checkpoint size for different benchmarks and compression types.]

- Uncompressed vs. Compressed
- Benchmarks: iPython/Shell, iPython/Demo, Baseline, ParGeant4, NAS/CG, NAS/EP, NAS/LU, NAS/SP, NAS/MG, NAS/IS, NAS/BT
Single node benchmark performance
Experimental Setup

- Distributed (cluster) tests:
  - 32 node cluster
  - 4 cores per node (128 total cores)
  - dual-socket, dual-core Xeon 5130
  - 8 or 16 GB ram/node
  - 64-bit Red Hat Enterprise 4
  - Linux 2.6.9

- Single node tests:
  - 8 cores
  - dual-socket, quad core Xeon E5320
  - 8 GB ram
  - 64-bit Debian “sid”
  - Linux 2.6.28

- DMTCP has been tested on:
  - Ubuntu, Debian, OpenSuse, Fedora, RHEL, ...
  - Linux 2.6.9 and up
  - x86, x86_64
Our checkpoint algorithm

- The checkpoint management thread, in each user process, performs the following:
  1. Wait for the checkpoint to begin
  2. Hijack and suspend user threads
  3. Node-local elections for shared resources
  4. Drain sockets to process memory
  5. Single-process checkpointing
  6. Refill sockets
  7. Resume user threads
  8. Go to step 1

“______” is a cluster-wide barrier
Our restart algorithm

Initially, one restart process per node, in each restart process:
1. Restore files, ptys, other single process FDs
2. Reconnect sockets using a cluster wide discovery service
3. Fork into user processes
4. Rearrange FDs for each process
5. Restore each process memory / threads
6. Continue with step 9 in the checkpoint algorithm
   - Refill kernel buffers
   - Resume user threads
Checkpoint time is dominated by writing checkpoints to disk. Compression disabled. A synthetic program on 32 nodes.