Implement Checkpointing for Android
(to speed up boot time and development process)

Jim Huang (黃敬群) <jserv@0xlab.org>
Kito Cheng (程皇嘉) <kito@0xlab.org>
Developer, 0xlab

Nov 5, 2012! Embedded Linux Conference Europe
Rights to copy

© Copyright 2012 0xlab
http://0xlab.org/
contact@0xlab.org

Attrition – ShareAlike 3.0
You are free
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

- **Attribution.** You must give the original author credit.
- **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

**Your fair use and other rights are in no way affected by the above.**

License text: [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode)
Basic Idea: Process Migration

- Process Migration (in past applications)
  - Distributed Load Balancing
  - Efficient Resource Utilization

- Crash Recovery and Rollback Transaction
  - Useful to system admin
Checkpointing

• From Wikipedia:
  – ... is a technique for inserting fault tolerance into computing systems. It basically consists of storing a snapshot of the current application state, and later on, use it for restarting the execution in case of failure.

• Used in distributed shared memory systems
• Even used in reversible debugger
• Different from virtual machine level snapshot/resume mechanisms
  – Checkpointing emphasizes on process level.
Ideas about Checkpointing for Android

• Resume to stored state for faster Android boot time
• Better product field trial experience due to regular checkpointing
• Deploy problematic states for engineering analysis and debugging transparently
• Q&A stress test purpose
Expectations of Checkpointing

• Application-transparent
  – supports applications without modifications or recompilation

• Supports a broad class of applications
  – Databases
  – parallel / MPI apps
  – desktop apps

• Comprehensive support for user-level state, kernel-level state, and distributed computation and communication state

• Supported on unmodified Linux kernel
  – checkpoint-restart should be integrated by addons
Challenges in checkpoint and restore

- Network stack will continue to execute even after application processes are stopped
- No system call interface to read or write control state
- No system call interface to read send socket buffers
- No system call interface to write receive socket buffers
- Consistency of control state and socket buffer state
Communication state checkpoint

- Acquire network stack locks to freeze TCP processing
- Save receive buffers using socket receive system call in peek mode
- Save send buffers by walking kernel structures
- Copy control state from kernel structures
- Modify two sequence numbers in saved state to reflect empty socket buffers
Communication state restart

- Create a new socket
- Copy control state in checkpoint to socket structure
- Restore checkpointed send buffer data using the socket write call
- Deliver checkpointed receive buffer data to application on demand
Existing Checkpointing mechanisms

- **CryoPID**
  - http://cryopid.berlios.de/

- **BLCR** (Berkeley Lab Checkpoint/Restart)
  - https://ftg.lbl.gov/projects/CheckpointRestart/

- **DMTCP**
  - http://dmtcp.sourceforge.net/
Implementation Considerations

- Checkpointing can be implemented in
  - kernel modifications + helpers in userspace
  - pure userspace
- Introduce a virtualization layer groups processes into specific states with private virtual name space
  - Intercepts system calls to expose only virtual identifiers (e.g., vpid)
  - Preserves resource names and dependencies across migration
- Mechanism to checkpoint and restart states
  - User and kernel-level state
  - Primarily uses system call handlers
  - File system not saved or restored
DMTCP

- **Distributed Multi-Threaded CheckPointing.**
- Works with Linux Kernel 2.6.9 and later.
- Supports sequential and multi-threaded computations across single/multiple hosts.
- Entirely in user space (no kernel modules or root privilege).
  - Transparent (no recompiling, no re-linking).
- Written in Northeastern University and MIT and under active development since 2006.
- License: GNU LGPL (allows freely using and linking)
CT = DMTCP checkpoint thread
T = User Thread

Coordinator

Signal (USR2)
DMTCP

Process 1
CT
T1

Network Socket

Process N
CT
T1
T2

```
dmtcp_checkpoint <EXE>  # starts coordinator
dmtcp_command -c       # talks to coordinator
dmtcp_restart ckpt_<EXE>-*.dmtcp
```

- Coordinator: a stateless synchronization server for the distributed checkpointing algorithm.
- Checkpoint/Restart performance related to size of memory, disk write speed, and synchronization.
How DMTCP works (1/4)

- MTCP: component for checkpoint single-process
- SIGUSR2: Used internally from checkpoint thread to user threads.

1. CT send SIGUSR2 to each threads for suspend
2. Write checkpoint image to disk
3. Exit SIGUSR2 handler, and resume.
How DMTCP works (2/4)

- **LD_PRELOAD**: Transparently preloads checkpoint libraries `dmtcphijack.so` which installs libc wrappers and checkpointing code.

- **Wrappers**: Only on less heavily used calls to libc
  - open, fork, exec, system, pipe, bind, listen, setsockopt, connect, accept, clone, close, ptsname, openlog, closelog, signal, sigaction, sigvec, sigblock, sigsetmask, sigprocmask, rt_sigprocmask, pthread_sigmask
  
  - Overhead is negligible.

```c
int open(const char *path, int flags){
    ... 
    func[(open](path, flags);
    ...
}
```

```c
int open(const char *path, int flags){
    ...
}
```

User Program  dmtcphijack.so  libc.so
How DMTCP works (3/4)

- Additional wrappers when process id & thread id virtualization is enabled
  - getpid, getppid, gettid, tcgetpgrp, tcsetpgrp, getgrp, setpgrp, getsid, setsid, kill, tkill, tgkill, wait, waitpid, waitid, wait3, wait4

```c
... pid = getpid(); ...
int getpid(){
    ...
    real_pid = funcs[_getpid]();
    return pid_table[real_pid];
}
dmtcphijack.so
...}
```
• Checkpoint image compression on-the-fly (default).
• Currently only supports dynamically linking to libc.so. Support for static libc.a is feasible, but not implemented.
Checkpoint under DMTCP

- `dmtcphijack.so` and `libmttcp.so` present in executable’s memory.
  - `dmtcp_checkpoint <EXE>`

Diagram:

- Coordinator
- User program
  - Connect to exist coordinator or create a new coordinator
  - `CT`
  - `T1` `T2`
Checkpoint under DMTCP

• Ask coordinator process for checkpoint via dmtcp_command.
  – dmtcp_command -c

• DMTCP also provides API to send command or query status

![Diagram showing Coordinator, User programs, and command flow]
Checkpoint under DMTCP

- Suspend user threads with SIGUSR2.

1. Coordinator sends command for prepare checkpoint.
2. CT sends SIGUSR2 to each thread for suspend.
Pre-checkpoint stage
Synchronize every node and elect shared file descriptor leaders.
Drain kernel buffers and do network handshake with peers.

1. Report all thread except CT are suspended
2. Drain buffers
Wait until all node are ready.
Checkpoint under DMTCP

- Write checkpoint to disk
  - One checkpoint file per process
  - ckpt_<EXE>_<uid>.dmtcp

Coordinator

User program

CT
T1 T2

User program

CT
T1 T2

Wait until all node checkpoint are done

Write checkpoint to disk separately
• Post-Checkpoint stage
• Refill kernel buffers

Wait until all node post-checkpoint are done

Refill buffer and re-handshake with peers.
Checkpoint under DMTCP

- Resume user threads.

User program

Coordinator

Send resume command

Resume!
Restart Process loads in memory.

- `dmtcp_restart ckpt_<EXE>_<uid>.dmtcp`
Restart under DMTCP

- Fork user program
Restart under DMTCP

- Reopen files and recreate ptys
- Recreate and reconnect sockets
- Rearrange file descriptors to initial layout

Wail all node recreate socket and reopen file

1. Reopen file/ptys/sockets
2. Rearrange FD via dup2
• Restore memory content.
• Restore stack status for checkpoint thread.

FDs will preserved across execve!
Restart under DMTCP (5/6)

- Restore other threads.
  - Recreate thread and restore stack and context.
  - Restore back to the post-checkpoint stage

- Refill kernel buffer

**Diagram:**
- Coordinator
  - CT
    - T1
    - T2
  - User program
- Wail all node restore done
  - 1. Memory and thread come back now!
  - 2. Refill buffers
• Resume user threads

```c
while (mtcp_state_value(&thread -> state) == ST_SUSPENDED) {
    mtcp_state_futex (&(thread -> state),
                     FUTEX_WAIT, ST_SUSPENDED, NULL);
}
```
DMTCP Workflow

Start with dmtcp wrapper

Run

Pre-Checkpoint

Checkpoint

Post-Checkpoint

Provide hook point!

Get checkpoint command

Restart

Re-open FDs

Restore memory

Restore threads
OS Features supported by DMTCP

• Threads, mutexes/semaphores, fork, exec
• Shared memory (via mmap), TCP/IP sockets, UNIX domain sockets, pipes, ptys, terminal modes, ownership of controlling terminals, signal handlers, open and/or shared fds, I/O (including the readline library), parent-child process relationships, process id & thread id virtualization, session and process group ids, and more...
DMTCP/Android: Additional Features

(LGPL; separated from Android)

- ARM Architecture support
  - Verified on Samsung Galaxy S2 + Android 4.0

- Binder IPC
  - Client: supported
  - Server: partially supported

- Ashmem: supported

- Logger: supported

- Properties: supported

- Wakelocks: Not supported

Source code is available

https://github.com/0xlab/dmtcp-android
https://github.com/0xlab/android-checkpoint
Support new FD type in DMTCP

- In DMTCP, every FD has an associated `Connection`:
  - TcpConnection, FileConnection, PtyConnection

- Implement a new subclass of Connection if you want to support a new FD type for Android:
  - AshmemConnection, BinderConnection, LoggerConnection, PropertyConnection

- Also, implement the `preCheckpoint`, `postCheckpoint`, and any others if needed.
class Connection {
public:
    virtual void preCheckpoint (const dmtcp::vector<int>&, KernelBufferDrainer&);
    virtual void postCheckpoint (const dmtcp::vector<int>&, bool);
    virtual void restore (const dmtcp::vector<int>&, ConnectionRewirer *);
    virtual bool isDupConnection (const Connection&, dmtcp::ConnectionToFds&);
    virtual void doLocking (const dmtcp::vector<int>&);
    virtual void saveOptions (const dmtcp::vector<int>&);
    virtual void restoreOptions (const dmtcp::vector<int>&);
    virtual void doSendHandshakes (const dmtcp::vector<int>&, const dmtcp::UniquePid&);
    virtual void doRecvHandshakes (const dmtcp::vector<int>&, const dmtcp::UniquePid&);
    virtual void restartDup2 (int, int);
protected:
    virtual void serializeSubClass (jalib::JBinarySerializer&);
};
Android Binder support for DMTCP

- **BinderConnection**
  - Reopen `/dev/binder` and reset ioctl parameters
  - Restore the mmap region

- **Hijack the whole libbinder**
  - Prevent libbinder from interpreting data twice
  - Implement necessary DMTCP hooks: `preCheckpoint`, `postCheckpoint`, `postRestart`
    - Re-initialize libbinder in `postRestart`

- The server part is partially supported because binder server is calling a blocked ioctl and blocking the whole checkpoint process.
  - We implement an early checkpoint stage to suspend such kind of threads.
More extensions in DMTCP/Android

- Improve the hook system in DMTCP
  - Original design only allows one set hook function.
  - Allow more than one set hook function in DMTCP/Android.

- Implement per thread callback hook
  - Restore the DVM internal thread info

- Add barrier and synchronize mechanisms to DMTCP
  - In order to make precise program checkpointing.
Android specific modifications

• Reorder code in framework
  – registerZygoteSocket()
    • The socket is inherited from the parent process `init`, which implies we cannot handle it in DMTCP.
  – Move few initializations later than the checkpoint process since the current binder support is incomplete.

• Reserve the ashmem's file descriptor
  – Original behavior is to close the fd after mmap
  – DMTCP binds connection to one fd, so the connection will be destroyed if that fd is closed.

• Implement the missing PThread function in bionic libc
  – pthread_tryjoin_np is required by DMTCP, but it is not implemented in original bionic.
Technical Issues when modifying DMTCP

- ARM Architecture support is incomplete.
  - We are going to contribute back to upstream.

- Different TLS implementation semantics between glibc and bionic libc
  - DMTCP/Android follows the techniques used in Android’s OpenGL ES package which links and defers to the slot of TLS in bionic libc. Not elegant, but it works

- PThread implementation expectation is quite different
  - AOSP master branch is merging libc from NetBSD, so it should be better for compatibility.

- Behavior of dynamic linker differs a lot in bionic libc.
- Flags in dlopen() is not really functional.
- The way to find symbol in bionic libc differs: weak symbol
## Checkpoint for Zygote

- **Experiment environment:**
  - Android-x86 ICS emulated by VirtualBox
  - Host: Thinkpad x200 (Intel Core 2 Due @ 2.4 GHz)

<table>
<thead>
<tr>
<th></th>
<th>with gzip</th>
<th>without gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Checkpoint time</strong></td>
<td>~10s</td>
<td>~5.5s</td>
</tr>
<tr>
<td><strong>Restart time</strong></td>
<td>~0.5s</td>
<td>~0.2s</td>
</tr>
<tr>
<td><strong>Image size</strong></td>
<td>~3M</td>
<td>~17M</td>
</tr>
</tbody>
</table>
Observations from logcat

-------- beginning of /dev/log/system
I/Vold ( 1270): Vold 2.1 (the revenge) firing up
D/Vold ( 1270): Volume usb state changing -1 (Initializing) -> 0 (No-Media)
I/Netd ( 1271): Netd 1.0 starting
I/ ( 1275): ServiceManager: 0x8062b50
I/ ( 1276): ServiceManager: 0x804fb98
I/AudioFlinger( 1276): Loaded primary audio interface from LEGACY Audio HW HAL (audio)
I/AudioFlinger( 1276): Using 'LEGACY Audio HW HAL' (audio.primary) as the primary audio interface
...
D/AudioHardware( 1276): ### setVoiceVolume: 1.000000
I/AudioPolicyService( 1276): [1276]Loaded audio policy from LEGACY Audio Policy HAL (audio_policy)

-------- beginning of /dev/log/system
I/Vold ( 1270): Vold 2.1 (the revenge) firing up
D/Vold ( 1270): Volume usb state changing -1 (Initializing) -> 0 (No-Media)
I/Netd ( 1271): Netd 1.0 starting
I/ ( 1275): ServiceManager: 0x8062b50
I/ ( 1276): ServiceManager: 0x804fb98
I/AudioFlinger( 1276): Loaded primary audio interface from LEGACY Audio HW HAL (audio)
I/AudioFlinger( 1276): Using 'LEGACY Audio HW HAL' (audio.primary) as the primary audio interface
...
D/AudioHardware( 1276): ### setVoiceVolume: 1.000000
I/AudioPolicyService( 1276): [1276]Loaded audio policy from LEGACY Audio Policy HAL (audio_policy)

Normal bootup log message

Bootup log message with restart
## Android Boottime with DMTCP

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boottime</td>
<td>27.96</td>
<td>27.95</td>
<td>32.89</td>
<td>26.59</td>
<td>32.33</td>
<td>32.36</td>
<td>33.22</td>
<td>32.99</td>
<td>36.47</td>
<td>32.85</td>
<td>31.56</td>
</tr>
<tr>
<td>Boottime with DMTCP (w/o gzip)</td>
<td>14.98</td>
<td>15.13</td>
<td>14.61</td>
<td>13.90</td>
<td>14.72</td>
<td>14.84</td>
<td>15.46</td>
<td>15.06</td>
<td>15.32</td>
<td>15.39</td>
<td>14.94</td>
</tr>
</tbody>
</table>

Measured by uptime in `onStart()` of Launcher2
Reversible Debugger based on DMTCP

- URDB is a universal reversible debugger that can add reversibility gained through the use of DMTCP.
- A user debugging with gdb would then ask URDB to go back to a point in time to when:
  – the expression had a correct value
  – the next statement would cause the expression to take on an incorrect value.
- Project page: http://sourceforge.net/projects/urdb/
Reversible Debugger using Checkpointing

Before:

After:

DMTCP:
- ckpt_gdb.1
- ckpt_a.out.1
- ckpt_gdb.2
- ckpt_a.out.2

pseudo-tty (pty):

gdb
a.out

tty:

gdb
a.out

URDB

ckpt/restart

ckpt/restart
• “DMTCP: An New Linux Checkpointing Mechanism for Vanilla Universe Job”, Condor Project, University of Wisconsin-Madison
• “Checkpointing using DMTCP, Condor, Matlab and FReD”, Gene Cooperman, Northeastern University, Boston
• URDB: Universal Reversible Debugger
  http://arxiv.org/abs/0910.5046
• Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems, G. (John) Janakiraman, Jose Renato Santos, Dinesh Subhraveti, Yoshio Turner, HP Labs
http://0xlab.org