Instant OS Updates via Userspace Checkpoint-and-Restart

Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, Pavel Emelyanov
OS updates are prevalent
And OS updates are unavoidable

- Prevent known, state-of-the-art attacks
  - Security patches
- Adopt new features
  - New I/O scheduler features
- Improve performance
  - Performance patches
Please do not power off or unplug your machine.
Installing update 11 of 208 ..
Unfortunately, system updates come at a cost

- Unavoidable downtime
- Potential risk of system failure
Unfortunately, system updates come at a cost

- Unavoidable downtime
- Potential risk of system failure

THE FINANCIAL AND OTHER COSTS OF DATA CENTER DOWNTIME
Posted on March 30, 2014 by Mary Hiers

... Amazon had 49 minutes of downtime in January 2013, which cost them an estimated $4 million in lost sales, or $80k per minute of the outage. When Google went down for a short period in 2013, it cost an estimated $545,000 in lost sales per minute. Obviously, downtime costs big companies far more in losses than small companies, but regardless, it’s an expense nobody wants to face. Here are some other important facts and figures.

$109k per minute
Hidden costs (losing customers)
Example: memcached

- Facebook's memcached servers incur a downtime of 2-3 hours per machine
  - Warming cache (e.g., 120 GB) over the network
Example: memcached

- Facebook's memcached servers incur a downtime of 2-3 hours per machine
  - Warming cache (e.g., 120 GB) over the network

Our approach updates OS in 3 secs for 32GB of data from v3.18 to v3.19 for Ubuntu / Fedora releases
Existing practices for OS updates

- Dynamic Kernel Patching (e.g., kpatch, ksplice)
  - **Problem:** only support minor patches

- Rolling Update (e.g., Google, Facebook, etc)
  - **Problem:** inevitable downtime and requires careful planning
Existing practices for OS updates

Losing application state is inevitable
→ Restoring memcached takes 2-3 hours

• Rolling Update (e.g., Google, Facebook, etc)
  – **Problem:** inevitable downtime and requires careful planning
Existing practices for OS updates

- Dynamic Kernel Patching (e.g., kpatch, ksplice)
  - Problem: only support minor patches
- Rolling Update (e.g., Google, Facebook, etc)
  - Problem: inevitable downtime and requires careful planning

Losing application state is inevitable
→ Restoring memcached takes 2-3 hours

Goals of this work:
- Support all types of patches
- Least downtime to update new OS
- No kernel source modification
Problems of typical OS update

OS

Stop service

Memcached

OS
Problems of typical OS update

OS → Memcached → Stop service → OS

Soft reboot → New OS
Problems of typical OS update

OS

Memcached

Stop service

OS

Soft reboot

Start service

New OS

Memcached

New OS

New OS
Problems of typical OS update

OS

2-3 hours of downtime

Stop service

Soft reboot

Start service

New OS

New OS
Problems of typical OS update

OS

Stop service

Soft reboot

Start service

Memcached

2-3 hours of downtime

2-10 minutes of downtime

New OS

New OS
Problems of typical OS update

2-3 hours of downtime

2-10 minutes of downtime

Is it possible to keep the application state?
KUP: Kernel update with application checkpoint-and-restore (C/R)
KUP: Kernel update with application checkpoint-and-restore (C/R)
KUP: Kernel update with application checkpoint-and-restore (C/R)
KUP: Kernel update with application checkpoint-and-restore (C/R)
KUP: Kernel update with application checkpoint-and-restore (C/R)

KUP's life cycle

- Checkpoint
- In-kernel switch
- Restore
KUP: Kernel update with application checkpoint-and-restore (C/R)

**KUP's life cycle**

- Checkpoint
- In-kernel switch
- Restore

1-10 minutes of downtime
KUP: Kernel update with application checkpoint-and-restore (C/R)

KUP's life cycle

Checkpoint
In-kernel switch
Restore

1-10 minutes of downtime

Challenge: how to further decrease the potential downtime?
Techniques to decrease the downtime

1) Incremental checkpoint

Checkpoint

In-kernel switch

Restore
Techniques to decrease the downtime

1) Incremental checkpoint
   - Checkpoint
   - In-kernel switch
   - Restore

2) On-demand restore
Techniques to decrease the downtime

1) Incremental checkpoint

3) FOAM: a snapshot abstraction

2) On-demand restore
Techniques to decrease the downtime

1) Incremental checkpoint
2) On-demand restore
3) FOAM: a snapshot abstraction
4) PPP: reuse memory without an explicit dump
Techniques to decrease the downtime

1) Incremental checkpoint
2) On-demand restore
3) FOAM: a snapshot abstraction
4) PPP: reuse memory without an explicit dump
Incremental checkpoint

- Reduces downtime (up to 83.5%)
- **Problem**: Multiple snapshots increase the restore time
Incremental checkpoint

- Reduces downtime (up to 83.5%)
- **Problem:** Multiple snapshots increase the restore time

![Diagram showing Naive and Incremental checkpoint with downtime highlighted]

- $S_i \rightarrow$ Snapshot instance
Incremental checkpoint

- Reduces downtime (up to 83.5%)
- **Problem**: Multiple snapshots increase the restore time

**Timeline**

- **Naive checkpoint**
  - \( S_1 \)
  - Downtime

- **Incremental checkpoint**
  - \( S_1 \)
  - \( S_2 \)
  - \( S_i \rightarrow \text{Snapshot instance} \)
Incremental checkpoint

- Reduces downtime (up to 83.5%)
- **Problem**: Multiple snapshots increase the restore time

**Timeline**

- Naive checkpoint: $S_1$
- Incremental checkpoint:
  - $S_1$
  - $S_2$
  - $S_3$

$S_i \rightarrow$ Snapshot instance

- downtime
Incremental checkpoint

- Reduces downtime (up to 83.5%)
- **Problem**: Multiple snapshots increase the restore time

---

**Timeline**

- **Naive checkpoint**: 
  - \(S_1\)
  - Downtime

- **Incremental checkpoint**: 
  - \(S_1\)
  - \(S_2\)
  - \(S_3\)
  - \(S_4\)
  - Downtime

\(S_i \rightarrow \text{Snapshot instance}\)
On-demand restore

- Rebind the memory once the application accesses it
  - Only map the memory region with snapshot and restart the application
- Decreases the downtime (up to 99.6%)
- **Problem**: Incompatible with incremental checkpoint
Problem: both techniques together result in inefficient application C/R

- During restore, need to map each pages individually
  - Individual lookups to find the relevant pages
  - Individual page mapping to enable on-demand restore

- An application has 4 pages as its working set size

- Incremental checkpoint has 2 iterations
  - 1\textsuperscript{st} iteration → all 4 pages (1, 2, 3, 4) are dumped
  - 2\textsuperscript{nd} iteration → 2 pages (2, 4) are dirtied

- Increases the restoration downtime (42.5%)
Problem: both techniques together result in inefficient application C/R

- During restore, need to map each pages individually
  - Individual lookups to find the relevant pages
  - Individual page mapping to enable on-demand restore

- An application has 4 pages as its working set size

- Incremental checkpoint has 2 iterations
  - 1\textsuperscript{st} iteration \(\rightarrow\) all 4 pages (1, 2, 3, 4) are dumped
  - 2\textsuperscript{nd} iteration \(\rightarrow\) 2 pages (2, 4) are dirtied

- Increases the restoration downtime (42.5%)
New abstraction: file-offset based address mapping (FOAM)

- Flat address space representation for the snapshot
  - One-to-one mapping between the address space and the snapshot
  - No explicit lookups for the pages across the snapshots
  - A few map operations to map the entire snapshot with address space
- Use sparse file representation
  - Rely on the concept of holes supported by modern file systems
- Simplifies incremental checkpoint and on-demand restore
Techniques to decrease the downtime

1) Incremental checkpoint
2) On-demand restore
3) FOAM: a snapshot abstraction
4) PPP: reuse memory without an explicit dump
Redundant data copy

• Application C/R copies data back and forth
• Not a good fit for applications with huge memory
Redundant data copy

• Application C/R copies data back and forth

• Not a good fit for applications with huge memory
Redundant data copy

• Application C/R copies data back and forth
• Not a good fit for applications with huge memory
Redundant data copy

• Application C/R copies data back and forth
• Not a good fit for applications with huge memory
Redundant data copy

- Application C/R copies data back and forth
- Not a good fit for applications with huge memory
Redundant data copy

- Application C/R copies data back and forth
- Not a good fit for applications with huge memory

Flow diagram:
- Memcached
- OS
- New OS
- RAM
  - Dump data
  - Read data
- Snapshot
  - 1 2 3 4
- Running
- Checkpoint
- In-kernel switch
- Restore
- Running
Redundant data copy

- Application C/R copies data back and forth

Is it possible to avoid memory copy?

![Diagram showing data flow between RAM, Snapshot, Dump data, and Read data.]
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy

**Diagram:**
- **Memcached**
- **OS**
- **RAM**
  - 1
  - 2
  - 3
  - 4
- **Snapshot**
- **Running** → **Checkpoint** → **In-kernel switch** → **Restore** → **Running**

**Legend:**
- Reserve the memory in the OS
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy

MEMCACHED

OLD OS

Implicitly map the memory region

RAM

1 2 3 4

SNAPSHOT

NEW OS

Running → Checkpoint → In-kernel switch → Restore → Running
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy

![Diagram showing memory management across OS transitions]
Avoid redundant data copy across reboot

- Reserve the application's memory across reboot
- Inherently rebind the memory without any copy

Challenge: how to notify the newer OS without modifying its source?
Persist physical pages (PPP) without OS modification

- Reserve virtual-to-physical mapping information
  - Static instrumentation of the OS binary
  - Inject our own memory reservation function, then further boot the OS

- Handle page-faults for the restored application
  - Dynamic kernel instrumentation
  - Inject our own page fault handler function for memory binding
Persist physical pages (PPP) without OS modification

- Reserve virtual-to-physical mapping information
  - Static instrumentation of the OS binary

- No explicit memory copy

- Does not require any kernel source modification
  - Dynamic kernel instrumentation
  - Inject our own page fault handler function for memory binding
Implementation

- Application C/R → criu
  - Works at the namespace level

- In-kernel switch → kexec system call
  - A mini boot loader that bypasses BIOS while booting

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>criu / on-demand restore</td>
<td>810 lines of C</td>
</tr>
<tr>
<td>criu / FOAM</td>
<td>950 lines of C</td>
</tr>
<tr>
<td>criu / PPP</td>
<td>600 lines of C</td>
</tr>
<tr>
<td>KUP systemctl, init</td>
<td>1040 lines of Python/Bash</td>
</tr>
<tr>
<td>criu / others, kexec(), etc.</td>
<td>150 lines of C</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,550 lines of code</strong></td>
</tr>
</tbody>
</table>
Evaluation

- How effective is KUP's approach compared to the in-kernel hot patching?

- What is the effective performance of each technique during the update?
KUP can support major and minor updates in Ubuntu

- KUP supports 23 minor/4 major updates (v3.17–v4.1)
- However, kpatch can only update 2 versions
  - e.g., layout change in data structure

![kpatch failure scenarios](image)
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance

![Diagram showing bandwidth and timeline with different storage options like Basic - SSD, Incremental checkpoint - SSD, On-demand restore - SSD, FOAM - SSD, Basic - RP-RAMFS, Incremental checkpoint - RP-RAMFS, On-demand restore - RP-RAMFS, FOAM - RP-RAMFS, and PPP. The timeline ranges from 190 to 250 seconds, and the bandwidth ranges from 0 to 150 MB.]
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance

Bar chart showing bandwidth (MB) over time (sec) for different configurations:
- Basic – SSD
- Incremental checkpoint – SSD
- On-demand restore – SSD
- FOAM – SSD
- Basic – RP-RAMFS
- Incremental checkpoint – RP-RAMFS
- On-demand restore – RP-RAMFS
- FOAM – RP-RAMFS
- PPP

Timeline (sec) ranges from 190 to 250, and Bandwidth (MB) ranges from 0 to 150.
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance

Bandwidth (MB)

Timeline (sec)

Downtime (sec)
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Updating OS with memcached

- PPP has the least degradation
- Storage also affects the performance
Limitations

• KUP does not support checkpoint and restore all socket implementations
  – TCP, UDP and netlink are supported

• Failure during restoration
  – System call is removal or interface modification
Demo
Summary

• KUP: a simple update mechanism with application checkpoint-and-restore (C/R)

• Employs various techniques:
  – New data abstraction for application C/R
  – Fast in-kernel switching technique
  – A simple mechanism to persist the memory
Summary

- KUP: a simple update mechanism with application checkpoint-and-restore (C/R)
- Employs various techniques:
  - New data abstraction for application C/R
  - Fast in-kernel switching technique
  - A simple mechanism to persist the memory

Thank you!
Backup Slides
Handling in-kernel states

- Handles namespace and cgroups
- `ptrace()` syscall to handle the blocking system calls, timers, registers etc.
- Parasite code to fetch / put the application's states
- `/proc` file system exposes the required information for application C/R
- A new mode (TCP_REPAIR) allows handling the TCP connections
What cannot be checkpointed

- X11 applications
- Tasks with debugger attached
- Tasks running in compat mode (32 bit)
Possible changes after application C/R

- Per-task statistics
- Namespace IDs
- Process start time
- Mount point IDs
- Socket IDs (st_ino)
- VDSO
Suitable applications

- Suitable for all kinds of applications
- PPP approach supports all types of applications
  - May fail to restore on the previous kernel
- FOAM is not a good candidate for write-intensive applications
  - More confidence in safely restoring the application on the previous kernel
PPP works effectively

- FOAM on SSD → slow
- FOAM on RP-RAMFS → space inefficient
PPP works effectively

- FOAM on SSD → slow
- FOAM on RP-RAMFS → space inefficient
PPP works effectively

- FOAM on SSD → slow
- FOAM on RP-RAMFS → space inefficient

![Graph showing performance and out of memory error with WSS and write percentage]