Efficient Virtualization on Embedded Power Architecture Platforms

Aashish Mittal, Dushyant Bansal, Sorav Bansal
Indian Institute of Technology Delhi

Varun Sethi
Freescale Semiconductor
Embedded Virtualization: Motivation

- Resource partitioning (control-plane / data-plane)
- High availability (active/standby configuration)
- In-service upgrade
- Sandboxing (isolate untrusted software)
- App compatibility, consolidation, …
Our Contribution

Efficient Software virtualization on Embedded Power architecture

2-6x faster performance improvement
Embedded Power Architecture
Embedded Power Architecture

- Satisfies Popek-and-Goldberg virtualization requirements
- 4 byte fixed length word aligned instructions
- Software managed small TLB
  - 16 variable sized (4KB - 4GB) and 512 fixed size (4KB) entries
  - S-byte alignment constraint for a page of size S
- Orthogonal rwx page permission bits for user/kernel
- No segmentation support
- Branching
  - Direct branch restricted to 26 bit offset
  - Indirect branches through registers only
Trap-and-emulate
Trap-and-Emulate
# Performance overhead of Trap-and-Emulate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Bare-metal</th>
<th>Trap-and-Emul.</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Boot</td>
<td>Boots a Linux 3.0 guest</td>
<td>6.5</td>
<td>30.03</td>
<td>4.6x</td>
</tr>
<tr>
<td>Echo spawn</td>
<td>Spawns 1000 echo processes</td>
<td>1.4</td>
<td>21.34</td>
<td>15.2x</td>
</tr>
<tr>
<td>Find</td>
<td>Executes ‘find / -name temp’</td>
<td>0.39</td>
<td>1.89</td>
<td>4.8x</td>
</tr>
</tbody>
</table>
## Performance overhead of Trap-and-Emulate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Bare-metal</th>
<th>Trap-and-Emul.</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Boot</td>
<td>Boots a Linux 3.0 guest</td>
<td>6.5</td>
<td>30.03</td>
<td>4.6x</td>
</tr>
<tr>
<td>Echo spawn</td>
<td>Spawns 1000 echo processes</td>
<td>1.4</td>
<td>21.34</td>
<td>15.2x</td>
</tr>
<tr>
<td>Find</td>
<td>Executes ‘find / -name temp’</td>
<td>0.39</td>
<td>1.89</td>
<td>4.8x</td>
</tr>
</tbody>
</table>

4-16x slower than Bare-metal
Binary Translation
Full Binary Translation

x86 based solution: Full Binary Translation (Full BT) [VMware]
In-place Binary Translation

Our solution on Embedded Power: In-place Binary Translation (In-place BT)

Kernel Code

Guest Virtual Address Space

Privileged instruction

Translated instruction
Full BT vs. In-place BT: Architectural Implications

<table>
<thead>
<tr>
<th>Requirement</th>
<th>x86</th>
<th>Embedded Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Popek-and-Goldberg</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Fixed length word aligned instructions</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Advantages of In-place Binary Translation

<table>
<thead>
<tr>
<th></th>
<th>Full BT</th>
<th>In-place BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead on Indirect jump</td>
<td>High</td>
<td>Zero</td>
</tr>
<tr>
<td>Translation complexity</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
In-Place Binary Translation
In-Place Binary Translation

Kernel Code

Privileged instruction

Guest Virtual Address Space
In-Place Binary Translation

Guest Virtual Address Space

Kernel Code

Move from machine state register to r0

mfmsr r0
In-Place Binary Translation

Guest Virtual Address Space

Kernel Code

Hypervisor

mfmsr r0
In-Place Binary Translation

Guest Virtual Address Space

Kernel Code

Hypervisor

mfmsr r0
In-Place Binary Translation

Kernel Code

```
 mfmsr r0
```

Guest Virtual Address Space

```
 lwz r0, addr
```

Shared Space

```
 addr
```

Load msr from ‘shared space’ into r0

Privileged instruction

Translated instruction
In-Place Binary Translation

Kernel Code

```
mfmsr r0
```

```
lwz r0, addr
```

Shared Space

```
addr
```

Guest Virtual Address Space

Privileged instruction

Translated instruction

Mark as execute-only

Translated instruction

Translated instruction

Translated instruction
Translation for complex instruction

Guest Virtual Address Space

Kernel Code

mtmsr r0

Move r0 to *machine state register*

More than one instruction to emulate
Translation for complex instruction

Kernel Code

mtmsr r0

branch

Emulation Code

Translation Cache

Branch back
Translation for complex instruction

Kernel Code

mtmsr r0

branch

Emulation Code

Translation Cache

26

26

Branch back
Translation Cache - Constraint

Translation Cache

±32MB

Kernel Code

Guest Virtual Address Space
Placement Constraints

- Unused by Guest
- Placement constraints on translation cache
Placement Constraints

- Unused by Guest  easy
- Placement constraints on translation cache

+32MB

Shared Space

Translation Cache

Kernel Code

Guest Virtual Address Space
Placement Constraints

- Unused by Guest  
  - easy
- Placement constraints on translation cache  
  - harder

- Shared Space
- Translation Cache
- Kernel Code

Guest Virtual Address Space
Stealing space for Translation Cache

- Steal space within 32MB
- Choose some data section which lies within 32MB of kernel
Stealing space for Translation Cache

- Steal space within 32MB
  - Choose some data section which lies within 32MB of kernel

- Mark space as execute-only
  - All R/W accesses to this region trap into hypervisor
Read/Write Tracing
Read/Write tracing cost

Kernel Code

Guest Virtual Address Space

Translation cache

In-place patch

Privileged instruction

Translated instruction
Read/Write tracing cost

Kernel Code

Guest Virtual Address Space

Privileged instruction

Translated instruction

Traced region access

Translation cache

In-place patch

Read/Write access by Guest
Read/Write tracing cost

Guest Virtual Address Space

- Privileged instruction
- Translated instruction
- Traced region access

Kernel Code

In-place patch

Translation cache

R/W traced region

Read/Write access by Guest
Read/Write tracing cost

Kernel Code

Guest Virtual Address Space

<table>
<thead>
<tr>
<th>Privileged instruction</th>
<th>Translated instruction</th>
<th>Traced region access</th>
</tr>
</thead>
</table>

In-place patch

Translation cache

R/W traced region

Read/Write access by Guest
R/W tracing – False sharing

256MB

Page Faults

- TLB Misses
- R/W Tracing
R/W tracing – Tradeoff

256MB

4KB

TLB Misses
Tracing Exits

Page Faults
R/W tracing – Tradeoff

256MB

4MB

Page Faults

- TLB Misses
- Tracing Exits
R/W tracing – Adaptive page resizing

Page Faults

256MB

4KB

TLB Misses

Tracing Exits
Adaptive Page Resizing: Optimizing Tradeoff

• Fragmentation of large page. Based on workload type:
  – *Burst* - page is broken such that all patch-sites on that page belong to the “shortest” page
  – *Scan* - page is broken into two halves and the half with larger number of tracing page faults is untraced

• Remove patch and re-instate Read/Write privileges
  – High number of tracing page faults

• Opportunistic merging
  – Periodically merge neighboring pages to reduce TLB pressure
## Adaptive Page Resizing Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trap-and-Emulate</th>
<th>In-place BT + Adaptive-PR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Running Time in sec</td>
<td></td>
</tr>
<tr>
<td>Linux Boot</td>
<td>30.03</td>
<td>14.39</td>
</tr>
<tr>
<td>Echo spawn</td>
<td>21.34</td>
<td>8.9</td>
</tr>
<tr>
<td>Find</td>
<td>1.89</td>
<td>1.67</td>
</tr>
</tbody>
</table>

1.5-2.4x faster than trap-and-emulate
Overhead

Kernel Code

Guest Virtual Address Space

Translation cache

Privileged instruction

Translated instruction

Traced region access

Read/Write access by Guest
Overhead

Guest Virtual Address Space

Translation cache

Kernel Code

Read/Write access by Guest

Translation cache

Read/Write access by Guest to mirrored data

Privileged instruction

Translated instruction

Traced region access
Adaptive-Data Mirroring

Instructions accessing data on pages protected by R/W tracing

- Mirror the accessed data on *shared space*
- *In-place translation*

![Diagram showing adaptive-data mirroring](image)
## Adaptive Data Mirroring Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trap-and-Emulate</th>
<th>Adaptive-PR</th>
<th>+Adaptive-DM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Boot</td>
<td>30.03</td>
<td>14.39</td>
<td>12.39</td>
</tr>
<tr>
<td>Echo spawn</td>
<td>21.34</td>
<td>8.9</td>
<td>6.85</td>
</tr>
<tr>
<td>Find</td>
<td>1.89</td>
<td>1.67</td>
<td>0.83</td>
</tr>
</tbody>
</table>

Running Time in sec

2.2-3.1x faster than trap-and-emulate
## Comparison with Bare Metal

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bare-Metal</th>
<th>Our Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Boot</td>
<td>6.5</td>
<td>12.39</td>
</tr>
<tr>
<td>Echo spawn</td>
<td>1.4</td>
<td>6.85</td>
</tr>
<tr>
<td>Find</td>
<td>0.39</td>
<td>0.83</td>
</tr>
</tbody>
</table>

1.9-4.9x slower than bare metal
More results in paper

• Comparisons on Imbench and unixbench
• Comparing adaptive page resizing with statically configured page resizing
• Three-way tradeoff between
  – Privileged Instruction Exits
  – Read/Write Tracing Page Faults
  – TLB Miss Page Faults
Correlation between Architecture & Hypervisor Design

• RISC vs CISC architecture
  – In-place vs Full Binary Translation
• Orthogonal RWX bits vs RW/NX bits (x86)
• Software managed TLBs (with page size flexibility) vs Segmentation
Conclusions

• Improved virtualization performance of unmodified guests on embedded Power Architecture (2-6x)

• Insight into implications of subtle architectural design decisions on VMM design
Questions?
Adaptive Page Resizing
Adaptive Page Resizing
Adaptive Page Resizing
Adaptive Page Resizing
Adaptive Page Resizing
Adaptive Page Resizing
Adaptive Page Resizing