Lazy Cache Invalidation for Self-Modifying Codes

Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, and Trevor Mudge

University of Michigan
Advanced Computer Architecture Laboratory
Mobile Devices Use High-Level SW

Worldwide Smartphone OS Market Share, 1Q 2012

- Android
- BlackBerry OS
- iOS
- Symbian
- Windows Phone 7/Windows Mobile
- Linux
- Other

Source: IDC

- Relies on the Dalvik VM
- Now supports JIT compilation/dynamic optimization

Android is the most popular smartphone OS
Mobile Development Tradeoffs

High-level languages
Programmer productivity

Portability

Native code
Power efficiency

Tradeoffs

Higher software overhead
More bugs
Longer Development Time

Performance
JIT Compilation Basics

- JIT compilation is often used with high-level languages
  - Enables portability
  - Allows for dynamic optimization
- JIT compilers cause additional SW overhead
  - Code profiling
  - Code cache management

Source: Hazelwood CGO 2004
JIT Compilation Basics

- JIT compilation is often used with high-level languages
  - Enables portability
  - Allows for dynamic optimization

- JIT compilers cause additional SW overhead
  - Code profiling
  - Code cache management

![Diagram showing JIT compilation process]

Memory Allocation
Page Permissions
D-Cache Cleaning
I-cache Invalidation

Source: Hazelwood CGO 2004
Outline

- Motivation
- Traditional I-Cache Invalidation
- Lazy Cache Invalidation
- Results
- Conclusion
JIT Must Manage I-Cache Coherence

I-caches are read only
– Use SW to manage coherence
I-caches are read only
  – Use SW to manage coherence
The jmp target is changed
  – Write new instruction to d-cache
I-caches are read only
  – Use SW to manage coherence
The jmp target is changed
  – Write new instruction to d-cache
Clean the d-cache line
  – Make visible to i-cache
I-caches are read only
  – Use SW to manage coherence

The jmp target is changed
  – Write new instruction to d-cache

Clean the d-cache line
  – Make visible to i-cache

Invalidate line in the i-cache
  – Force i-cache to get new line
I-caches are read only
  – Use SW to manage coherence
The jmp target is changed
  – Write new instruction to d-cache
Clean the d-cache line
  – Make visible to i-cache
Invalidate line in the i-cache
  – Force i-cache to get new line
Fetch the jmp instruction
  – Will miss in i-cache, hit in L2
Two Ways to Perform Invalidations

- Invalidate a single i-cache line
  - Invalidate a line based on its address
  - **Pro:** precise control over which lines get invalidated
  - **Con:** must invalidate each line sequentially

- Invalidate the entire i-cache
  - Invalidates every i-cache line at once
  - **Pro:** invalidates the entire i-cache with 1 instruction
  - **Con:** unnecessary invalidations increase i-cache miss rate

- We propose page-sized invalidation
  - Invalidate a page based on its address
Two Ways to Perform Invalidations

- Invalidate a single i-cache line
  - Invalidate a line based on its address
  - **Pro:** precise control over which lines get invalidated
  - **Con:** must invalidate each line sequentially

- Invalidate the entire i-cache
  - Invalidates every i-cache line at once
  - **Pro:** invalidates the entire i-cache with 1 instruction
  - **Con:** unnecessary invalidations increase i-cache miss rate

- We propose page-sized invalidation
  - Invalidate a page based on its address

  Provides a balance between code cache management and i-cache miss rate
Outline

- Motivation
- Traditional I-Cache Invalidation
- Lazy Cache Invalidation
- Results
- Conclusion
**Lazy Cache Invalidation**

Add a page version number to cache and TLB

<table>
<thead>
<tr>
<th>I-Cache</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page</td>
<td>Tag</td>
</tr>
<tr>
<td>Physical Page</td>
<td>Version</td>
</tr>
</tbody>
</table>

Add a page version number to cache and TLB

- **Cache Hit**
- **TLB**
Lazy Cache Invalidation

On a fetch, compare page versions
If they don’t match, it’s a miss

I-Cache

<table>
<thead>
<tr>
<th>Physical Page</th>
<th>Tag</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>

TLB

<table>
<thead>
<tr>
<th>Physical Page</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

Cache Hit
Lazy Cache Invalidation

On a fetch, compare page versions
If they don’t match, it’s a miss

Add a new instruction, `pginv`, which increments counter in TLB.
Example Invalidation

\textit{pginv} A executed by CPU

\begin{itemize}
\item I-Cache
\begin{tabular}{|c|c|c|}
\hline
Page A & Tag A & 2 \\
\hline
\end{tabular}
\item TLB
\begin{tabular}{|c|c|}
\hline
Page A & 2 \\
\hline
\end{tabular}
\end{itemize}
Example Invalidation

pginv $A$ executed by CPU

Increment version in TLB
Example Invalidation

$pginv$ A executed by CPU

Increment version in TLB

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>2</th>
</tr>
</thead>
</table>

TLB

| Page A | 3 |

I-cache performs fetch
Example Invalidation

pginv A executed by CPU

Increment version in TLB

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>2</th>
</tr>
</thead>
</table>

TLB

<table>
<thead>
<tr>
<th>Page A</th>
<th>3</th>
</tr>
</thead>
</table>

I-cache performs fetch

I-cache miss, versions do not match
What if the Counter Rolls Over?

$pginv \ A$ executed by CPU

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>0</th>
</tr>
</thead>
</table>

TLB

| Page A | 7 |

3 bits
What if the Counter Rolls Over?

\[ pginv A \text{ executed by CPU } \]

Increment version in TLB

**I-Cache**

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>0</th>
</tr>
</thead>
</table>

**TLB**

| Page A | 0 |

\[ 3 \text{ bits} \]
What if the Counter Rolls Over?

.pg.inv A executed by CPU

I-Cache line still at version 0

Increment version in TLB

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>0</th>
</tr>
</thead>
</table>

TLB

| Page A | 0 |

3 bits

CPU
What if the Counter Rolls Over?

$pginv$ A executed by CPU

I-Cache line still at version 0

Increment version in TLB

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>0</th>
</tr>
</thead>
</table>

TLB

| Page A | 0 |

3 bits

I-cache performs fetch
What if the Counter Rolls Over?

$pginv\ A$ executed by CPU

I-Cache line still at version 0

I-Cache hit – fetch incorrectly fetches old line

I-cache performs fetch

CPU

Increment version in TLB

I-Cache

<table>
<thead>
<tr>
<th>Page A</th>
<th>Tag A</th>
<th>0</th>
</tr>
</thead>
</table>

TLB

<table>
<thead>
<tr>
<th>Page A</th>
<th>0</th>
</tr>
</thead>
</table>

3 bits

pginv A executed by CPU
What if the Counter Rolls Over?

\( \text{pginv A executed by CPU} \)

I-Cache line still at version 0

Increment version in TLB

I-Cache

Page A

Invalidate entire i-cache on rollover

I-cache performs fetch

I-Cache hit, but i-cache has old line

University of Michigan
Methodology

- gem5 full-system simulator
  - ARM ISA
  - SimpleTiming CPU
- DaCapo Benchmarks
  - Real-world apps, written in Java
  - Sun/Oracle Java v1.6.0.30
  - Ubuntu 11.04
  - Linux kernel v2.6.38.8
- BBench
  - Web-page rendering benchmark
  - Ran on Android browser
  - Android v2.3 Gingerbread
  - Android kernel v2.6.35.8

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>32kB split I/D</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>4-way set associative</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1MB shared L2</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>8-way set associative</td>
</tr>
</tbody>
</table>
Methodology

- **gem5 full-system simulator**
  - ARM ISA
  - SimpleTiming CPU

- **DaCapo Benchmarks**
  - Real-world apps, written in Java
  - Sun/Oracle Java v1.6.0.30
  - Ubuntu 11.04
  - Linux kernel v2.6.38.8

- **BBench**
  - Web-page rendering benchmark
  - Ran on Android browser
  - Android v2.3 Gingerbread
  - Android kernel v2.6.35.8

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>32kB split I/D</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>4-way set associative</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1MB shared L2</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>8-way set associative</td>
</tr>
</tbody>
</table>

Modified kernel source to use our invalidation instruction
Kernel Source Modifications

- Modified `v7_coherent_user_range()` in Linux kernel
  - Added support for our `pginv` instruction
- Invalidations always occur in page-sized chunks
  - Support for unaligned page invalidations
  - 2 possibilities:

```
Spans multiple pages

Invalidate both pages
```

```
Page A
Page B
```

```
Page A
Page B
```

```
Force a page alignment
Invalidate
range
```

Invalidation range
Kernel Source Modifications

- Modified `v7_coherent_user_range()` in Linux kernel
  - Added support for our `pginv` instruction
- Invalidations always occur in page-sized chunks
  - Support for unaligned page invalidations
  - 2 possibilities:
Outline

- Motivation
- Traditional I-Cache Invalidation
- Lazy Cache Invalidation
- Results
- Conclusion
I-Cache Invalidations are Prevalent

Sampled every 100k instructions

tradebeans

jython
I-Cache Invalidations are Prevalent

Sampled every 100k instructions
I-Cache Invalidations are Prevalent

Sampled every 100k instructions
I-Cache Invalidations are Prevalent

Invalidations are frequent and happen continuously

Sampled every 100k instructions
I-Cache Invalidations are Prevalent

Web browser – BBench

Not as many invalidations, but high frequency

Sampled every 100k instructions
I-Cache Invalidations are Prevalent

Web browser – BBench

Not as many invalidations, but high frequency

Sampled every 100k instructions

Invalidations are frequent and happen continuously
Invalidations Happen in Bursts

Number of instructions between bursts of invalidations
Invalidations Happen in Bursts

Median number of instructions in-between bursts of invalidations is low
Invalidations Happen in Bursts

Number of instructions between bursts of invalidations

Web-browser – BBench

CDF
Invalidations Happen in Bursts

Bursts of invalidations are further apart, but still frequent

Number of instructions between bursts of invalidations
Rollovers are Not Frequent

Counter Size

Instructions Between Overflows

1.0E+10
1.0E+09
1.0E+08
1.0E+07
1.0E+06
1.0E+05
1.0E+04
1.0E+03
1.0E+02
1.0E+01
1.0E+00

aurora  batik  eclipse  fop  h2  jython  luindex  lusearch  pmd  sunflow  tomcat  tradesbeans  tradesoap  xalan  bbench

1 bit
2 bits
3 bits
4 bits
5 bits
6 bits
7 bits
8 bits
Even for modestly sized counters, rollovers are not frequent
Results – Invalidation Speedup

Sequential Invalidation

Regular Execution I-Cache Maintenance Regular Execution

Time
Results – Invalidation Speedup

Sequential Invalidation

Regular Execution \(\rightarrow\) I-Cache Maintenance \(\rightarrow\) Regular Execution

Lazy Cache Invalidation

Regular Execution \(\rightarrow\) Regular Execution

Return to regular execution quicker.
Results – Invalidation Speedup

For typical page sizes, page invalidation can be sped up by ~128X
Conclusion

- Modern mobile applications rely on high-level SW features
  - These features incur additional overhead
  - CPU architectures should provide for high-level SW
- In this work, we focus on speeding up i-cache invalidations
  - Provide a mechanism to invalidate page-sized chunks of cache lines
- Page invalidation is sped up by ~128X
- Overhead is low
  - Small number of bits in TLB/cache line for version number
  - 1 instruction added to the ISA
  - Version counter rollover not frequent
Acknowledgements

- Thanks to my co-authors
- Thanks to the reviewers for their feedback
- This work was support by a grant from ARM
Questions?