Energy-Aware Code Cache Management for Memory-Constrained Java Devices

G. L. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irvin
CSE Department
The Pennsylvania State University
Introduction

- Just-In-Time compiler
- Adaptive compilation system
- Limited memory & cache size in embedded system
- Different compilation cost, execution cost, and interpretation for different Java methods
Previous Work

M. Chen and K. Olukuton. Targeting Dynamic Compilation for Embedded Environments. JVM 2002


N. Shaylor, A Just-in-Time Compiler for Memory Constrained Low-Power Devices. JVM 2002
Compile vs. Interpret

Cost
- Compile > Interpret > Execute (compiled code)

Execute when code is already in cache

Compile
- Compile cost can be amortized if the methods is invoked many times
- Compiled codes need to be put in cache

Interpret
- Higher cost than executing compiled code
LRU

When a method is invoked

- Execute if compiled code is already in code cache
- Interpret if method size is too large
- Compile & Execute

Kick-out strategy

- Kick out least recently used elements until there is enough space to put in cache
Problems Brought by LRU

- LRU always choose compile over interpret
- Not all methods will be invoked enough times to amortize their compilation cost
- LRU doesn’t consider invocation frequency
- It may happen that a frequently invoked method becomes a least recently used method and will be kicked out of code cache
Weight-based Bypassing

- Consider the relative importance (criticality) of different methods
- Combines recency and frequency
- Bypass (interpret) methods that have insufficient weight

Currently invoked method

```
method m1()
```

```
method m2()
```

>`?

```
weight_{m1}
```

```
weight_{m2}
```
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Threshold?</th>
<th>Threshold Reset?</th>
<th>Weight Reduction</th>
<th>Victim Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>LRU</td>
</tr>
<tr>
<td>Strategy 1</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>LRU</td>
</tr>
<tr>
<td>Strategy 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Local</td>
<td>LRU + Weight Comparison</td>
</tr>
<tr>
<td>Strategy 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Global</td>
<td>LRU + Weight Comparison</td>
</tr>
<tr>
<td>Strategy 4</td>
<td>Yes</td>
<td>Yes</td>
<td>Global</td>
<td>Min Weight + Weight Comparison</td>
</tr>
<tr>
<td>Strategy 5</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Threshold reset?

Strategy 1

...A...A...A...A...B...B...B...B...A...A...A...A...

Strategy 2

...A...A...A...A...A...B...B...B...B...B...A...A...A...A...

White: interpretation
Yellow: compile & execute
Red: execute
Weight Reduction

Currently invoked method

method m1() \[weight_{m1}\]

method m2() \[weight_{m2}\]

>?

Code cache

Local: reduce m2’s weight
Global: reduce the weight of all methods in code cache
Victim Selection

- LRU
- LRU + Weight Comparison
- Min Weight + Weight Comparison
Experiment Setup

- Benchmarks from SPEC JVM98 suite
- Java virtual machine is LaTTe
- Energy model from Simple Power
- Energy simulator based on Shade
- 8K code cache, 8K I-cache, 16K D-cache. Both I-cache and D-cache are 2-way associative with 32 bytes cache line size
- Threshold is 10
- Weight update policy that reduces the original weight by 25%
Normalized energy consumption with the base parameters
Impact of installation weight
Impact of Code Cache Size
Energy variance as a result of more aggressive compilation.
Conclusion

- The compilation decisions in constrained memory environments need to be careful.
- Traditional dynamic compilation strategy can be improved under in embedded systems.
- The choice of code cache replacement policies can influence the energy efficiency dramatically.