Client Compiler for the Java HotSpot™ Virtual Machine

Technology and Application

Tom Rodriguez, Ken Russell
Sun Microsystems, Inc.
Overall Presentation Goal

Learn about compilation in the Java HotSpot™ Virtual Machine, and the Client Compiler

Understand how the Client Compiler deals with specific Java™ programming language features

Get to know tuning and trouble-shooting techniques
Learning Objectives

• As a result of this presentation, you will be able to:
  – Understand “Java HotSpot compilation”
  – Write better code in the Java programming language
  – Improve the performance of your applications
  – See what’s new in 1.4 and what’s coming
Speaker’s Qualifications

- Tom Rodriguez is the technical lead for the Java HotSpot Client Compiler
- Ken Russell has worked on the Java HotSpot runtime and compiler for over two years
Presentation Agenda

• Compilation in the Java HotSpot™ VM
• Structure of the Client Compiler
• Implications for Code written in the Java™ Programming Language
• Miscellaneous
• Summary
• Q&A
Compilation in the Java HotSpot™ VM

- VM Configurations
- Compilation Steps
- On-Stack Replacement
- Class Hierarchy Analysis
- Deoptimization
- Quick Summary
VM Configurations

Typical Java VM Software Stack

- libjvm.so/jvm.dll
- Core VM
- Java
- OS Libraries
- OS
- Hardware

Topic of This Talk

- Client Compiler
- java
- java-client

- Server Compiler
- java-server

Session 3198
Compilation Steps

• Every method is interpreted first

• Hot methods are scheduled for compilation
  – Method invocations
  – Loops

• Compilation can be foreground/background
  – Foreground compilation default for Client VM
  – Background compilation in parallel
On-Stack Replacement (1)

- Choice between interpreted/compiled execution
- Problem with long-running interpreted methods
  - Loops!
- Need to switch to compiled method in the middle of interpreted method execution
  - On-Stack Replacement (OSR)
On-Stack Replacement (2)

void m1() {
    ...
    while (i < n-1) {
        // OSR here
        ...
    }
    ...
}
Class Hierarchy Analysis (1)

• Dynamic pruning of receiver class set
  – Static calls instead of virtual calls
  – Inlining across virtual calls
  – Faster type tests

• Class Hierarchy Analysis (CHA)
  – Analysis of loaded classes
  – Can change over time
  – Effective
Session 3198

Class Hierarchy Analysis (2)

A a;
B b;
C c;
D d;
E e;
F f;
G g;

A virtual m

B virtual m

C virtual m

D virtual n

E virtual n

F

G virtual m

a.m → A.m, B.m
b.m → B.m
d.m → A.m
d.n → D.n
f.m → A.m
f.n → D.n

extends

loaded

unloaded
Deoptimization (1)

- Compile-time assumptions may become invalid over time
  - Class loading
- Debugging of program desired
  - Single-stepping
- Active compiled methods become invalid
- Need to switch to interpreted method in the middle of compiled method execution
  - Deoptimization
Deoptimization (2)

T3  m3(...) {
    ...  
    // Deopt. here
    }

T2  m2(...) {
    m3(...);
    }

T1  m1() {
    ...  
    m2(...);
    ...  
    }

Stack

m0 frame

comp. m1 w/inlined m2, m3 frame

m1 interpreted

m2 interpreted

m3 interpreted

Deopt.

m0 frame
Quick Summary

• Client VM differs from Server VM in compiler

• Hotspots trigger compilation
  – Compiled method invocation
  – OSR

• Class loading, debugging changes compile-time assumptions
  – Deoptimization
What’s New in 1.4

- Low-level Intermediate Representation
- Deoptimization
- Inlining
- Improved Debugging and Profiling Support
Deoptimization and Inlining

• 1.3.x client compiler inlined simple accessors
• 1.4 uses deoptimization and C HA
  – More inlining possibilities
• 1.4.0 client compiler does not inline:
  – Methods with exception handlers
  – Synchronized methods
Improved Debugging and Profiling

- Full-speed debugging
  - Compiler no longer disabled when using debuggers (-Xdebug)
  - Application runs at full speed
    - Breakpoint setting causes deoptimization
  - Server compiler support coming in 1.4.1

- Profiling
  - Substantial work on JVM PI robustness
  - Existing tools work much better with 1.4
Structure of the Client Compiler

- Representation of bytecodes
- Frontend
- Backend
- Machine code generation
Representation of Code

- **High Level Intermediate (HIR)**
  - Closely mimics bytecode
  - Directed graph to enforce evaluation order

- **Low Level Intermediate (LIR)**
  - Close to machine instructions
  - Some high level instructions
    - Virtual calls
    - Type checks
    - Intrinsics
Frontend

- Parse bytecodes into HIR
  - Eliminate loads (copy propagation)
  - Local value numbering
  - Constant folding and identities
  - Inline
  - Use CHA to optimize calls

- Global Optimizations
  - Block merging
  - Eliminate null checks
Intermediate Representation

Bytecodes

0 aload_1
1  bipush 46
3 invokevirtual  #139
6  istore_2
7  iload_2
8  ifgt  18
11 aloadd_0
12 getfield  #2
15 invokevirtual  #93
18 aloadd_1
19 iconst_0
20 ...
Session 3198

Backend

- Generate LIR by walking HIR
  - Expression level register allocation
- Optimize LIR
  - Assign locals to registers
    - Loop focused
    - Peephole optimizer
- Emit machine code from LIR
  - Generate object maps for use by GC
  - Generate information for deoptimization
Code Generation

LIR:

label [label:0x8176838]
move [edi|I] [eax|I]
move [int:4|I] [esi|I]
idiv [eax|I] [esi|I]
[edx|I] [eax|I] [bci:8]
cmp [eax|I] [int:0|I]
move [int:0|I] [esi|I]
branch [NE] [label:0x8182d60]
move [int:1|I] [esi|I]
label [label:0x8182d60]
move [esi|I] [eax|I]
return [eax|I]

Machine code:

movl %edi,%eax
movl 0x4,%esi
cmpl 0x80000000,%eax
jne -0x252e6b7b
xorl %edx,%edx
cmpl -1,%esi
je -0x252e6b78
cltd
idivl %esi,%eax
cmpl 0,%eax
movl 0x0,%esi
jne -0x252e6b65
movl 0x1,%esi
movl %esi,%eax
movl %ebp,%esp
popl %ebp
ret
Quick Summary

• Frontend does
  – Parse and analyze bytecodes
  – Build and optimize IR
  – Use CHA for very effective OO optimizations
  – Reorder CFG for code generation

• Backend does
  – LIR
  – Register allocation, low-level optimizations
  – Code generation
Implications for Code Written in the Java™ Programming Language

- Accessors
- Usage of `final`
- Object Allocation (`new`)
- Exception Handling
- Other Issues
- Quick Summary
Accessors

• Use accessors
  – XType getX() { return _x; }
  – void setX(XType x) { _x = x; }
• Abstracts from implementation
• Easier to maintain
• No performance penalty
  – Inlining!
Usage of final

- Don’t use `final` for performance tuning
- CHA will do the work
  - Where CHA can’t do it, `final` doesn’t help either
- Keep software extensible
- No performance penalty
  - Static calls
  - Inlining
Object Allocation (new)

- Object allocation (new) inlined
  - Works in most cases
  - Extremely fast (~ 10–20 clock cycles)
- Do not manage memory yourself
  - GC will slow down
  - Larger memory footprint
- Keep software simple
Exception Handling

• Exception object creation is very expensive
• Exception handling is not optimized
  – Use it for exceptional situations
  – Don’t use it as programming paradigm
  – Don’t use instead of regular `return`
• Exception handling costs only when used
  – Safe to declare exceptions
• If you must use exceptions for control flow:
  – Preallocate exception object
Other Issues

- Client Compiler optimized for clean OO code
  - “Hand-tuning” often counterproductive
  - Generated code can be problematic
    - Obfuscators
- Do not optimize prematurely
  - Use profiling information
Quick Summary

• Write clean OO code
  – Use accessors
  – Use final by design only
  – Use new for object allocation
  – Use exception handling for exceptional cases

• Keep it simple, keep it clean
Miscellaneous

- Built-in Profiler
- When to use the Client Compiler
- What’s coming in 1.4.1
- Quick Summary
Built-In Profiler

- Option: -X prof
  - E.g.: java -X prof -jar Java2Demo.jar
- Statistical (sampling) flat profiler
  - Not hierarchical
- Per thread
  - Output when thread terminates
## Sample Profiler Output

Flat profile of 27.38 secs (2574 total ticks): AWT-EventQueue-0

<table>
<thead>
<tr>
<th>Method</th>
<th>Interpreted</th>
<th>Native</th>
<th>Total interpreted (including elided)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun.java2d.loops.Blit.Blit</td>
<td>7.2%</td>
<td>0 + 90</td>
<td></td>
</tr>
<tr>
<td>sun.awt.windows.Win32BlitLoops.Blit</td>
<td>0.7%</td>
<td>0 + 9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.8%</td>
<td>72 + 174</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Compiled</th>
<th>Native</th>
<th>Total compiled (including elided)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.GradientPaintContext.clip...</td>
<td>9.2%</td>
<td>115 + 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.0%</td>
<td>179 + 8</td>
<td></td>
</tr>
</tbody>
</table>

Thread-local ticks:

<table>
<thead>
<tr>
<th>Blocked (of total)</th>
<th>51.7%</th>
<th>1330</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class loader</td>
<td>0.2%</td>
<td>2</td>
</tr>
<tr>
<td>Interpreter</td>
<td>0.3%</td>
<td>4</td>
</tr>
<tr>
<td>Compilation</td>
<td>10.0%</td>
<td>124</td>
</tr>
<tr>
<td>Unknown: running frame</td>
<td>0.5%</td>
<td>6</td>
</tr>
<tr>
<td>Unknown: thread_state</td>
<td>0.2%</td>
<td>2</td>
</tr>
</tbody>
</table>
When to Use the Client Compiler

- **Client Compiler characteristics**
  - Fast compilation
    - Quick startup time
  - Small footprint

- **Use for apps with same expectations**

- **Recommendation**
  - Try client and server, choose best
    - `java -client`
    - `java -server`
What’s New in 1.4.1

- Better peephole optimization
- Reduced footprint for compiled code
- New fast subtype check
- Inlining of jsrs
- Improved exception handling
  - Better full-speed debugging
- Reduced footprint on the Solaris™ OE
Overall Summary

- Compilation with the Java HotSpot™ VM
- Client Compiler Internals
- Programming and Tuning Hints
References

