The Java HotSpot™ Virtual Machine Client Compiler: Technology and Application

Robert Griesemer
Srdjan Mitrovic
Senior Staff Engineers
Sun Microsystems, Inc.
Overall Presentation Goal

- Learn about “Java HotSpot” 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 and compiler version differences
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
- Try work-arounds in case of compiler issues
- Understand the impact of different versions
Speakers’ Qualifications

Robert Griesemer is a principal architect of the Java HotSpot VM and the Client Compiler
- Robert has more than a decade of experience with programming language implementation
- He has been with the Java HotSpot team since its inception in 1994

Srdjan Mitrovic is a principal architect of the Java HotSpot Client Compiler
- Srdjan has more than a decade of experience with compilers and run-time systems
- He has been with the Java HotSpot team since 1996
Presentation Agenda

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

- VM configurations
- Compilation steps
- On-stack replacement
- Deoptimization
- Quick summary
VM Configurations

Typical Java VM Software Stack

- libjvm.so / jvm.dll
- Core VM
- Client Compiler
  - java
  - java -hotspot
- Java
  - OS Libraries
  - Server Compiler
  - java -server
- OS
- Hardware

Client Compiler is the topic of this talk.
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)

Stack Before OSR

Stack After OSR

OSR

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

Deopt.

comp. m1 w/ inlined m2, m3 frame

m0 frame

m3 interpr.
m2 interpreted
m1 interpreted
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
Presentation Agenda

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

- Frontend
- Optimizations
- Backend
- Quick summary
Front-End

- Parsing
  - Reading and analyzing method bytecodes
- Intermediate Representation (IR)
  - Internal representation for a method
  - Control Flow Graph (CFG)
  - Retain as much bytecode info as possible
- Optimizations
- Code Order
  - Reorder CFG for code generation
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  ...

 CFG
Optimizations

- Constant folding
- Simple form of value numbering
- Load elimination
- Dead code elimination
- Block elimination
- Null check elimination
- Inlining
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
A a;
B b;
C c;
D d;
E e;
F f;
G g;
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
Backend

- Register Allocation
- Low-Level IR (LIR)
  - JDK™ 1.4 release only
  - Some additional optimizations
- Code Generation
Machine code:

```assembly
pushl %ebp
movl %esp,%ebp
subl 40,%esp
movl %eax,-4(%esp,1)
movl %eax,-8(%esp,1)
movl 8(%ebp),%esi
cmpl 24(%ebp),%esi
jge 0xaeeff5
movl 8(%ebp),%esi
cmpl 0,%esi
jge 0xaeeef03
movl 0x0,8(%ebp)
...  
```
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 (JDK™ 1.4 release only)
- Register allocation, low-level optimizations
- Code generation
Presentation Agenda

- Compilation in the Java HotSpot™ VM
- Structure of the Client Compiler
- Implications for Code written in the Java™ Programming Language
- Miscellaneous
- Summary, Demo, and Q&A
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
  
  ```java
  x_type get_x() { return _x; }
  void set_x(x_type 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
  - Stack trace

- 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
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
Presentation Agenda

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

- Flags
- Built-in profiler
- Version differences
- When to use the client compiler
- Quick summary
Flags (1)

- No flag tuning for compiler required
- Use standard command line flags
- Special situations
  - `-Xint`
  - `-XX:+PrintCompilation`
    - JDK™ 1.3.1, JDK™ 1.4 technology
  - `.hotspotrc`
  - `.hotspot_compiler`
Flags (2)

- Usage
  - -XX:+FlagName, -XX:-FlagName

- Flags and default setting
  - -BackgroundCompilation
    - Foreground/background compilation
  - +UseCompilerSafepoints
    - May help in presence of crashes
  - +StackTraceInThrowable
    - Disable to turn off stack traces in exceptions
Built-in Profiler

- Option: -Xprof
  - E.g.: java -Xprof -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 + native Method</th>
<th>Compiled + native Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun.java2d.loops.Blit.Blit</td>
<td>7.2% 0 + 90</td>
<td>9.2% 115 + 0</td>
</tr>
<tr>
<td>sun.awt.windows.Win32BlitLoops.Blit</td>
<td>0.7% 0 + 9</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total interpreted (including elided)</td>
<td>19.8% 72 + 174</td>
<td>15.0% 179 + 8</td>
</tr>
<tr>
<td>Total compiled (including elided)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Thread-local ticks:**

<table>
<thead>
<tr>
<th>Time spent</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocked (of total)</td>
<td>1330</td>
</tr>
<tr>
<td>Class loader</td>
<td>2</td>
</tr>
<tr>
<td>Interpreter</td>
<td>4</td>
</tr>
<tr>
<td>Compilation</td>
<td>124</td>
</tr>
<tr>
<td>Unknown: running frame</td>
<td>6</td>
</tr>
<tr>
<td>Unknown: thread_state</td>
<td>2</td>
</tr>
</tbody>
</table>
## Version Differences

<table>
<thead>
<tr>
<th>Corresponding JDK™</th>
<th>1.3</th>
<th>1.3.1</th>
<th>1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified source base</td>
<td>n/y*</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OSR</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Simple inlining</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Full inlining</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Deoptimization</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>More optimizations</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

* SPARC™ processor implementation only
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 -hotspot`
    - `java -server`
Quick Summary

• No flag tuning required
  - Help for special situations
• Profiler for program tuning
• Minor version differences only
  - Faster code with JDK™ 1.4 release
• Try both compilers for optimal solution
Presentation Agenda

- Compilation in the Java HotSpot™ VM
- Structure of the Client Compiler
- Implications for Code written in the Java™ Programming Language
- Miscellaneous
- Summary, Demo, and Q&A
Overall Summary

- Java HotSpot™ compilation
- Client compiler internals
- Programming and tuning hints
- More information at the BOFs
  - BOF-2697
  - BOF-2639
References


Java HotSpot™ Technology Documents

http://java.sun.com/products/hotspot/2.0/docs.html
Q&A