The Java HotSpot VM
Under the Hood

Tobias Hartmann
Zoltán Majó

HotSpot Compiler Team
Oracle Corporation
About us

• **Tobias Hartmann**
  – MSc ETH Zurich, Switzerland
  – Lives in Rheinfelden, Germany

• **Zoltán Majó**
  – PhD ETH Zurich, Switzerland
  – Grew up in Cluj, Romania

• **Both of us: @Oracle since 2014**
  – Compiler team for the Java HotSpot Virtual Machine
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
A typical computing platform

Application Software

User Applications
- Spring
- Apache Sling
- ...
- ...

System Software
- Java SE
- Java EE
- Java Virtual Machine
- Operating system
- Hardware
A typical computing platform

User Applications

Application Software

Spring
Apache Sling
...
...

System Software

Hardware
A typical computing platform

**Application Software**

- User Applications
  - Spring
  - Apache Sling
  - ...
  - ...

**System Software**

- Java SE
- Java EE
- Java Virtual Machine
- Operating system
- Hardware
Outline

• Why virtual machines?
• The Java HotSpot VM
  – Just-in-time compilation
  – Optimistic compiler optimizations
  – Tiered compilation
  – Recent projects: Segmented Code Cache, Compact Strings
  – Future: AOT, JVMCI
• Conclusions
Programming language implementation

Programming language

C

Compiler
Standard libraries
Debugger
Memory management

Language implementation

Compiler
Standard libraries
Debugger
Memory management

Operating system

Windows

Hardware

Intel x86

Solaris

Compiler
Standard libraries
Debugger
Memory management

SPARC
(Language) virtual machine

- **Programming language**: Java, JavaScript, Scala, Python
- **Virtual machine**: HotSpot VM
- **Operating system**: Windows, Linux, Mac OS X, Solaris
- **Hardware**: Intel x86, PPC, ARM, SPARC
The VM: An application developer’s view

Java source code

int i = 0;
do {
    i++;
} while (i < f());

Bytecodes

0: iconst_0
1: istore_1
2: iinc
5: iload_1
6: invokevirtual f
9: if_icmplt 2
12: return

 compile

execute

HotSpot Java VM

• Ahead-of-time
• Using javac

• Instructions for an abstract machine
• Stack-based machine (no registers)
The VM: A VM engineer’s view

**Bytecodes**

0: iconst_0
1: istore_1
2: iinc
5: iload_1
6: invokevirtual f
9: if_icmplt 2
12: return

**HotSpot Java VM**

- **Compilation system**
  - C1
  - C2

- **Compiled method**
  - Machine code
  - Debug info
  - Object maps

- **Garbage collector**

- **Heap**

- **Stack**

**Interpretation**

**Execution**
Major components of HotSpot

- **Runtime**
  - Interpreter
  - Thread management
  - Synchronization
  - Class loading

- **Heap management**
  - Garbage collectors

- **Compilation system**
Interpretation vs. compilation in HotSpot

• **Template-based interpreter**
  - Generated at VM startup (before program execution)
  - Maps a well-defined machine code sequence to every bytecode instruction
  - Optimization: cache top-of-stack value in a register to reduce # of memory accesses

<table>
<thead>
<tr>
<th>Bytecodes</th>
<th>Machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: iconst_0</td>
<td>mov -0x8(%r14), %eax</td>
</tr>
<tr>
<td>1: istore_1</td>
<td>movzbl 0x1(%r13), %ebx</td>
</tr>
<tr>
<td>2: iinc</td>
<td>inc %r13</td>
</tr>
<tr>
<td>5: iload_1</td>
<td>mov $0xfff40,%r10</td>
</tr>
<tr>
<td>6: invokevirtual + 9: if_icmplt 2</td>
<td>jmpq *(%r10,%rbx,8)</td>
</tr>
</tbody>
</table>

• **Compilation system**
  - Speedup relative to interpretation: ~100X
  - Two *just-in-time compilers* (C1, C2)
  - Aggressive optimistic optimizations
Ahead-of-time vs. just-in-time compilation

- **AOT**: *Before* program execution
- **JIT**: *During* program execution
- **Tradeoff**: *Resource usage vs. performance of generated code*

<table>
<thead>
<tr>
<th>Performance</th>
<th>Interpretation</th>
<th>Amount of compilation</th>
<th>Compile everything</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bad performance</strong></td>
<td>due to interpretation</td>
<td>Good performance</td>
<td>due to good selection of compiled methods and of applied optimizations</td>
</tr>
</tbody>
</table>
Balancing resource usage and performance

• Getting to the “sweet spot”
• Carefully selecting
  1. Methods to compile
  2. Applied compiler optimizations
1. Selecting method to compile

- **Hot** methods (frequently executed methods)
- Profile method execution
  - # of method invocations, # of backedges
- A method’s lifetime in the VM

Diagram:
- Interpreter
  - Gather profiling information
- Compiler (C1 or C2)
  - Compile bytecode to native code
- Code cache
  - Store machine code

Deoptimization

Compiler’s optimistic assumptions proven wrong

# method invocations > THRESHOLD₁
# of backedges > THRESHOLD₂
Virtual call inlining

Class hierarchy

```
class A {
    void bar() {
        S_1;
    }
}

class B extends A {
    void bar() {
        S_2;
    }
}
```

Method to be compiled

```
void foo() {
    A a = create(); // return A or B
    a.bar();
}
```

Compiler: Inline call? Yes.
Virtual call inlining

Class hierarchy

class A {
    void bar() {
        S1;
    }
}

class B extends A {
    void bar() {
        S2;
    }
}

Method to be compiled

```java
void foo() {
    A a = create(); // return A or B
    S1;
}
```

• Benefits of inlining
  – Virtual call avoided
  – Code locality

• Optimistic assumption: only A is loaded
  – Note dependence on class hierarchy
  – Deoptimize if hierarchy changes

Compiler: Inline call?
Yes.
Virtual call inlining

Class hierarchy

```java
class A {
    void bar() {
        S1;
    }
}

class B extends A {
    void bar() {
        S2;
    }
}
```

Method to be compiled

```java
void foo() {
    A a = create(); // return A or B
    S1;
}
```
Virtual call inlining

Class hierarchy

class A {
    void bar() {
        S1;
    }
}

class B extends A {
    void bar() {
        S2;
    }
}

Method to be compiled

void foo() {
    A a = create(); // return A or B
    S1;
}
Virtual call inlining

Class hierarchy

```java
class A {
    void bar() {
        S1;
    }
}

class B extends A {
    void bar() {
        S2;
    }
}
```

Method to be compiled

```java
void foo() {
    A a = create(); // return A or B
    a.bar();
}
```

Compiler: Inline call? No.
Hot path compilation

Control flow graph

```
S_1;
S_2;
S_3;
if (x > 3)
S_4;
S_5;
S_6;
S_7;
S_8;
S_9;
```

10'000
True
False
0

```java
guard(x > 3)
S_1;
S_2;
S_3;
S_4;
S_5;
```

Generated code

Uncommon trap
Deoptimization

• Compiler’s optimistic assumption proven wrong
  – Assumptions about class hierarchy
  – Profile information does not match method behavior

• Switch execution from compiled code to interpretation
  – Reconstruct state of interpreter at runtime
  – Complex implementation

• Compiled code
  – Possibly thrown away
  – Possibly reprofiled and recompiled
Performance effect of deoptimization

• Follow the variation of a single method’s performance
2. Selecting compiler optimizations

- **C1 compiler**
  - Limited set of optimizations
  - Fast compilation
  - Small footprint

- **C2 compiler**
  - Aggressive optimistic optimizations
  - High resource demands
  - High-performance code

- **Graal**
  - Experimental compiler
  - Not part of HotSpot

---

Client VM

Server VM

Tiered compilation (enabled since JDK 8)
Balancing resource usage and performance

1. Selecting methods to compile
   - “Hot” methods
   - Controlled by invocation and backedge threshold

2. Choosing compiler optimizations
   - C1: *moderately optimizing* and *fast* compiler
   - C2: *highly optimizing* and *slow* compiler
   - Limitation (before JDK 8): *Single compiler* in the VM (client or server)
   - Starting with JDK 8: *Both compilers enabled* at the same time (tiered compilation)
Outline

• Why virtual machines?

• The Java HotSpot VM
  – Just-in-time compilation
  – Optimistic compiler optimizations
  – Tiered compilation
  – Recent projects: Segmented Code Cache, Compact Strings
  – Future: AOT, JVMCI

• Conclusions
Tiered compilation

• Combine the benefits of
  – Interpreter: Fast startup
  – C1: Fast warmup
  – C2: High peak performance
  – Still within the sweet spot of resource usage/performance tradeoff
Benefits of tiered compilation (artist’s concept)

Client VM (C1 only)

Performance

Method warm-up time

Interpreted

C1-compiled

VM Startup

Time

VM Teardown

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.
Benefits of tiered compilation (artist’s concept)

Server VM (C2 only)

<table>
<thead>
<tr>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method warm-up time</td>
</tr>
<tr>
<td>Interpreted</td>
</tr>
<tr>
<td>C2-compiled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM Startup</td>
</tr>
<tr>
<td>VM Teardown</td>
</tr>
</tbody>
</table>
Benefits of tiered compilation (artist’s concept)

Tiered compilation

Performance

Method warm-up time

Interpreted

C1-compiled

C2-compiled

Time

VM Startup

VM Teardown
Tiered compilation

• Combined benefits of interpreter, C1, and C2
• Additional benefits
  – More accurate profiling information
More accurate profiling

w/ tiered compilation: 1'100 samples gathered
w/o tiered compilation: 300 samples gathered
Tiered compilation

• Combined benefits of interpreter, C1, and C2

• Additional benefits
  – More accurate profiling information

• Drawbacks
  – Complex implementation
  – Careful tuning of compilation thresholds needed
  – More pressure on code cache – Tobias will tell you more about that
A method’s lifetime (w/ tiered compilation)

- **Interpreter**
  - Collect profiling information

- **C1**
  - Generate code quickly
  - Continue collecting profiling information

- **C2**
  - Generate high-quality code
  - Use profiling information

- **Deoptimization**
- **Code cache**
Performance of a single method (w/ tiered compilation)
Compilation levels (detailed view)

Typical compilation sequence

Compilation level

4  C2
3  C1: full profiling
2  C1: limited profiling
1  C1: no profiling
0  Interpreter

Associated thresholds:
Tier4InvocationThreshold
Tier4MinInvocationThreshold
Tier4CompileThreshold
Tier4BackEdgeThreshold

Associated thresholds:
Tier3InvokeNotifyFreqLog
Tier3BackedgeNotifyFreqLog
Tier3InvocationThreshold
Tier3MinInvocationThreshold
Tier3BackEdgeThreshold
Tier3CompileThreshold
Outline

• Why virtual machines?

• The Java HotSpot VM
  – Just-in-time compilation
  – Optimistic compiler optimizations
  – Tiered compilation
  – Recent projects: Segmented Code Cache, Compact Strings
  – Future: AOT, JVMCI

• Conclusions
Part 1: Segmented Code Cache

Improving the layout of JIT generated code
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
What is a code cache?

- **Stores code** generated by JIT compilers
- **Continuous chunk of memory**
  - Fixed size -XX:ReservedCodeCacheSize
  - Bump pointer allocation with free list
- **Memory managed by sweeper**
  - Cold methods are evicted
  - Hot methods remain
- **Why should I care?**
  - Essential for **performance**
Code cache usage: JDK 6 and 7
Code cache usage: JDK 8 (Tiered Compilation)
Code cache usage: JDK 9

- VM internals
- C1 compiled (profiled)
- C2 compiled (non-profiled)
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
Challenges

• Tiered compilation increases amount of code by **2-4X**
• All code is stored in a single code cache
  – Different types with different characteristics
  – Different usage frequencies (hotness)
  – Access to specific code requires full iteration
• High **fragmentation and bad locality**
Properties of compiled code

- Optimization level
- Size
- Cost of compilation
- Lifetime
# Types of compiled code

<table>
<thead>
<tr>
<th></th>
<th>Optimized</th>
<th>Small</th>
<th>Cheap</th>
<th>Immortal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-method code</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Profiled code (C1)</strong></td>
<td>instrumented</td>
<td>medium</td>
<td>cheap</td>
<td>limited</td>
</tr>
<tr>
<td><strong>Non-profiled code (C2)</strong></td>
<td>highly optimized</td>
<td>large</td>
<td>expensive</td>
<td>long</td>
</tr>
</tbody>
</table>

---

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.
Code cache usage

- C1 compiled (profiled)
- C2 compiled (non-profiled)
- VM internals
- free space
Code cache usage: Reality

- free space
- profiled code
- non-profiled code
Code cache usage: Reality

- Free space
- Profiled code
- Non-profiled code
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
Design

- **Without Segmented Code Cache**

- **With Segmented Code Cache**
  - non-profiled methods
  - profiled methods
  - non-methods
Segmented Code Cache: Reality

profiléd methods

non-profiled methods

- free space
- profiléd code
- non-profiled code
Segmented Code Cache: Reality

profiled methods

non-profiled methods

hotness
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
Evaluation: Code locality

- **Instruction Cache (ICache)**
  - 14% less ICache misses

- **Instruction Translation Lookaside Buffer (ITLB)\(^1\)**
  - 44% less ITLB misses
  - 9% speedup with microbenchmark

\(^1\) caches virtual to physical address mappings to avoid slow page walks
Evaluation: Sweeper

- # full sweeps
- Cleanup pause time
- Sweep time
Evaluation: Runtime

![Bar chart showing improvement in runtime for different benchmarks and environments.](chart.png)
Program Agenda

1. Background
2. Challenges
3. Design
4. Evaluation
5. Conclusion
Conclusion

• **Code layout matters**
  – *Significant impact on performance*
  – Code locality reduces iTLB misses

• **Segmented Code Cache helps**
  – Less sweeper overhead
  – Reduced fragmentation

• **Base for future extensions**
  – New code types
  – Separation of code and metadata
Part 2: Compact Strings

Improve VM internal handling of Strings
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Java Strings

```java
public class HelloWorld {
    public static void main(String[] args) {
        String myString = "HELLO";
        System.out.println(myString);
    }
}
```
Java Strings

```java
public class HelloWorld {
    public static void main(String[] args) {
        String myString = "HELLO";
        System.out.println(myString);
    }
}
```

```java
public final class String {
    private final char value[];
    ...
}
```
Java Strings

```java
public class HelloWorld {
    public static void main(String[] args) {
        String myString = "HELLO";
        System.out.println(myString);
    }
}
```

```java
public final class String {
    private final char value[];
    ...
}
```

class `String` is final and the `value` array is also final.

The string "HELLO" is represented as a `char` array:

- `char` value[] = [0x0048, 0x0045, 0x004C, 0x004C, 0x004F]
- UTF-16 encoded
- 2 bytes
“Perfection is achieved, not when there is nothing more to add, but when there is nothing more to take away.”

— Antoine de Saint Exupéry
There is a lot to take away here..

- UTF-16 encoded Strings always occupy **two bytes** per char
- **Wasted memory** if only Latin-1 (one-byte) characters used:

```
char value[] = {0x0048, 0x0045, 0x004C, 0x004C, 0x004F};
```

2 bytes
There is a lot to take away here..

- UTF-16 encoded Strings always occupy **two bytes** per char
- **Wasted memory** if only Latin-1 (one-byte) characters used:

```
char value[] = {0x0048, 0x0045, 0x004C, 0x004C, 0x004F}
```

2 bytes

- But is this a problem in **real life**?
Real life analysis: char[] footprint

• 950 heap dumps from a variety of applications
  – char[] footprint makes up 10% - 45% of live data
  – Majority of characters are single byte
  – 75% of Strings are smaller than 35 characters
  – 75% of Characters are in Strings of length < 250

• Predicted footprint reduction of 5% - 10%
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Project Goals

• Memory footprint reduction by improving space efficiency of Strings
• Meet or beat throughput performance of baseline JDK 9
• Full compatibility with related Java and native interfaces
• Full platform support
  – x86/x64, SPARC, ARM
  – Linux, Solaris, Windows, Mac OS X
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Design

• String class now uses a byte[] instead of a char[]

```java
public final class String {
    private final byte value[];
    private final byte coder;
    ...
}
```

• Additional 'coder' field indicates which encoding is used

```java
byte value[] = {\x00, \x04, \x00, \x04, \x00, \x04, \x00, \x04, \x00};
UTF-16 encoded
```

```java
byte value[] = {\x48, \x45, \x4c, \x4c, \x4f};
Latin-1 encoded
```

```java
HELLO
```
Design

- If all characters have a zero upper byte
  → String is compressed to **Latin-1** by stripping off high order bytes

- If a character has a non-zero upper byte
  → String cannot be compressed and is stored **UTF-16** encoded

\[
\text{byte } \text{value[]} = \begin{bmatrix}
0x00 & 0x48 & 0x00 & 0x45 & 0x00 & 0x4C & 0x00 & 0x4F \\
\end{bmatrix} \quad \text{UTF-16 encoded}
\]

\[
\text{byte } \text{value[]} = \begin{bmatrix}
0x48 & 0x45 & 0x4C & 0x4C & 0x4F \\
\end{bmatrix} \quad \text{Latin-1 encoded}
\]
Design

- Compression / inflation needs to fast
- Requires HotSpot support in addition to Java class library changes
  - JIT compilers: Inintrinsics and String concatenation optimizations
  - Runtime: String object constructors, JNI, JVMTI
  - GC: String deduplication
- Kill switch to enforce UTF-16 encoding (-XX:-CompactStrings)
  - For applications that extensively use UTF-16 characters
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Evaluation

- New and existing unit tests
- Microbenchmarks at the String API level
- Large benchmarks to measure overall performance
public class LogLineBench {
    int size;

    String method = generateString(size);

    public String work() throws Exceptions {
        return "[" + System.nanoTime() + "] " +
            Thread.currentThread().getName() +
            "Calling an application method " + method +
            " without fear and prejudice.";
    }
}
LogLineBench results

<table>
<thead>
<tr>
<th></th>
<th>Performance ns/op</th>
<th>Allocated b/op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Baseline</td>
<td>149</td>
<td>153</td>
</tr>
<tr>
<td>CS disabled</td>
<td>152</td>
<td>150</td>
</tr>
<tr>
<td>CS enabled</td>
<td>142</td>
<td>139</td>
</tr>
</tbody>
</table>

- Kill switch works (no regression)
- 27% performance improvement and 46% footprint reduction
Large workloads

- **SPECjbb2005**
  - 21% footprint reduction
  - 27% less GCs
  - 5% throughput improvement

- **SPECjbb2015**
  - 7% footprint reduction
  - 11% critical-jOps improvement

- **Weblogic (startup)**
  - 10% footprint reduction
  - 5% startup time improvement
Program Agenda

1. Java Strings
2. Project Goals
3. Design
4. Evaluation
5. Conclusion
Conclusion

Compact Strings helps our applications a lot.

Ongoing effort: Indify String Concat, Fused Strings

Try out JDK 9 early access: jdk9.java.net/download/

.. and tell us how it performs with your applications!
Future

• **AOT: Ahead-of-time compilation**
  – Compile to native code (not to Java bytecodes)
  – More information: [https://www.youtube.com/watch?v=Xybzyv8qbOc](https://www.youtube.com/watch?v=Xybzyv8qbOc) (45-minute talk from JVMLS’15)

• **JVMCI: Java Virtual Machine Compiler Interface**
  – Current compilers written in C/C++
  – JVMCI: Interface to allow Java code to intercept JVM activity and plug-in native code
  – Experimental feature, Graal and SubstrateVM use it
Conclusions

• Java – a vibrant platform
  – New features: Segmented Code Cache, Compact Strings, JVMCI
  – … and many other features to be released with JDK 9
  – Stay tuned!

• The future of the Java platform
  “Our SaaS products are built on top of Java and the Oracle DB—that’s the platform.”

  Larry Ellison, Oracle CTO
Thank you for your attention!
Backup slides