JavaOne℠

Sun's 2001 Worldwide Java Developer Conference

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

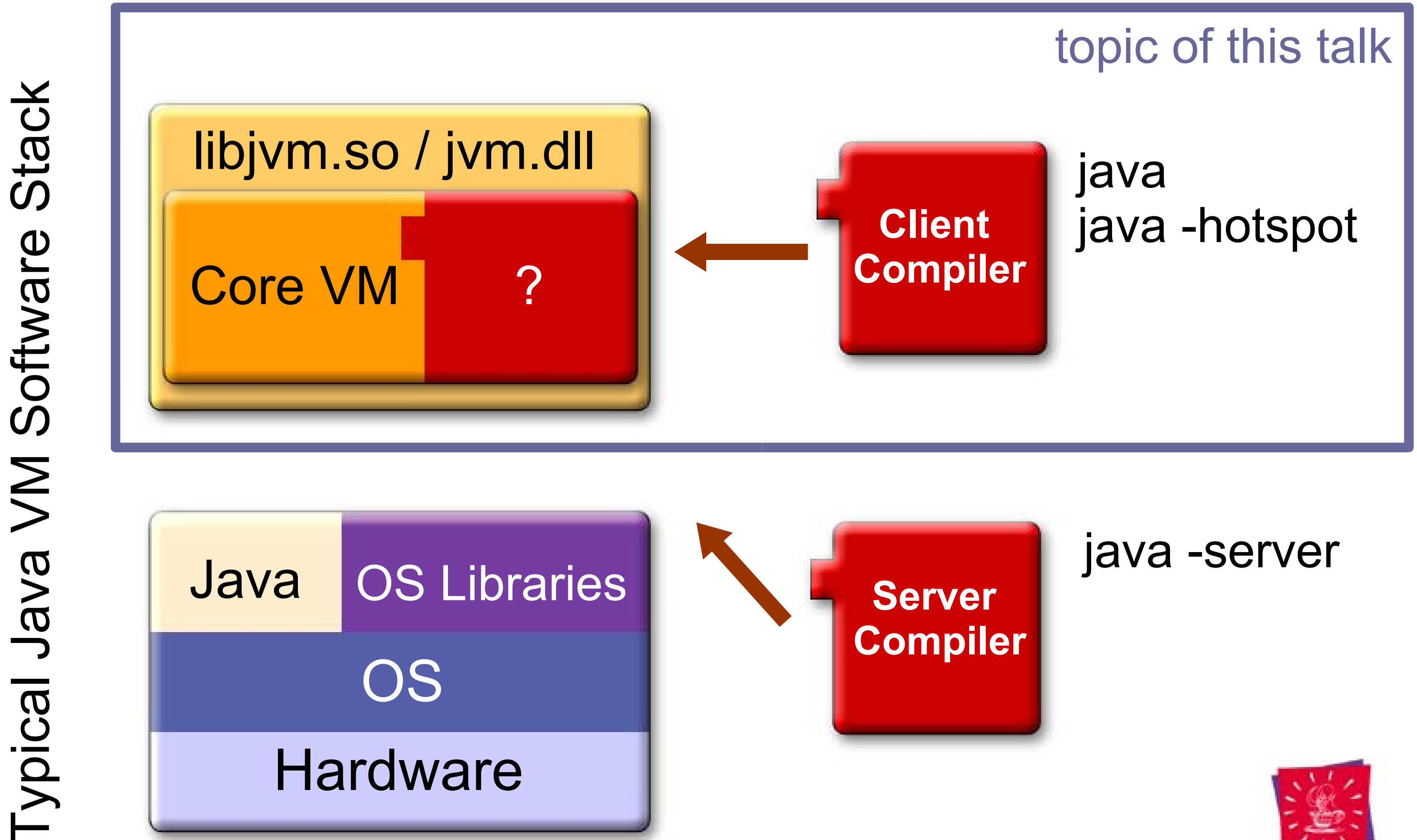
Compilation steps

On-stack replacement

Deoptimization

Quick summary

VM Configurations



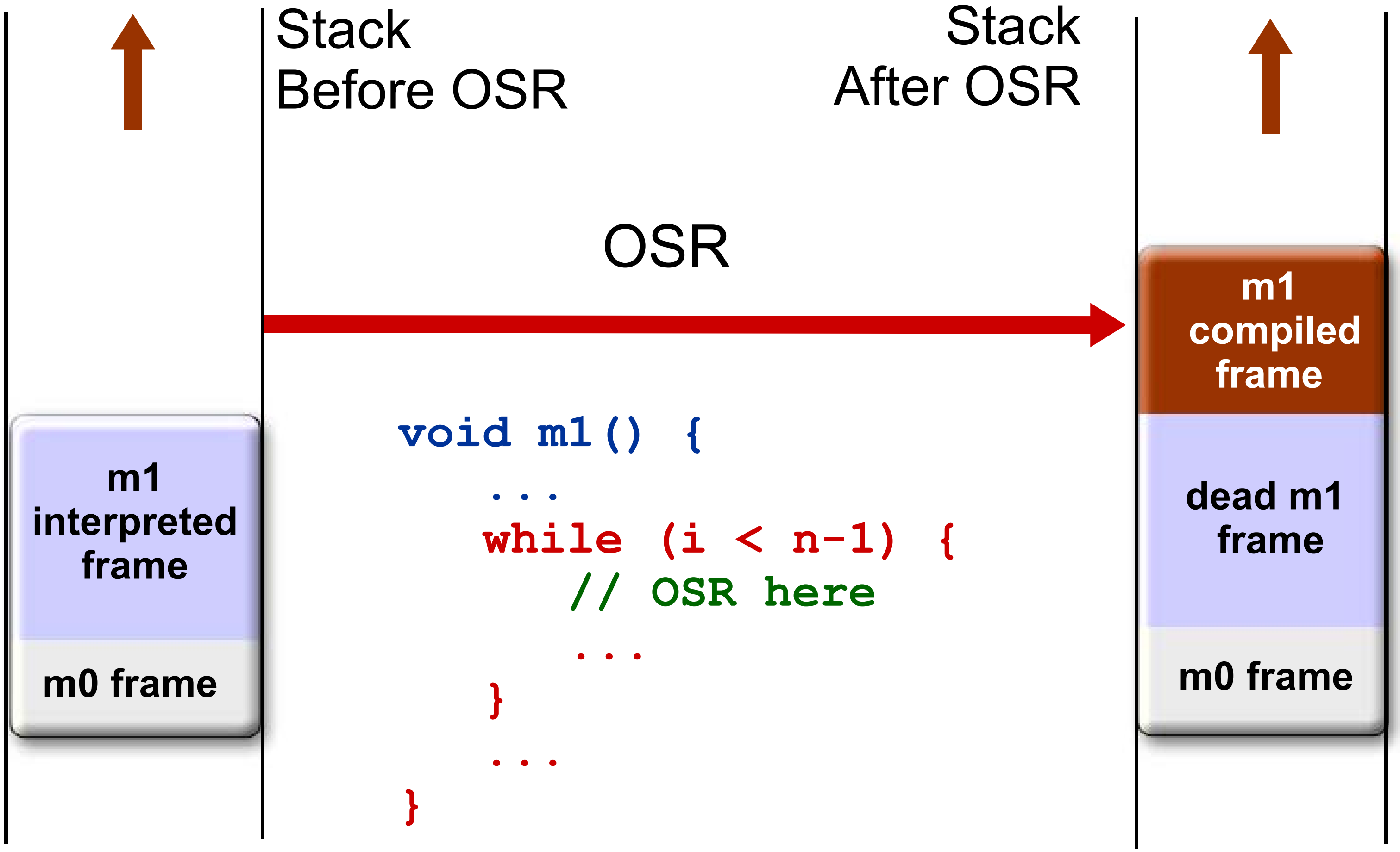
Compilation Steps

- 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

m1
interpreted
frame

m0 frame

m1
compiled
frame

dead m1
frame

m0 frame

```
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

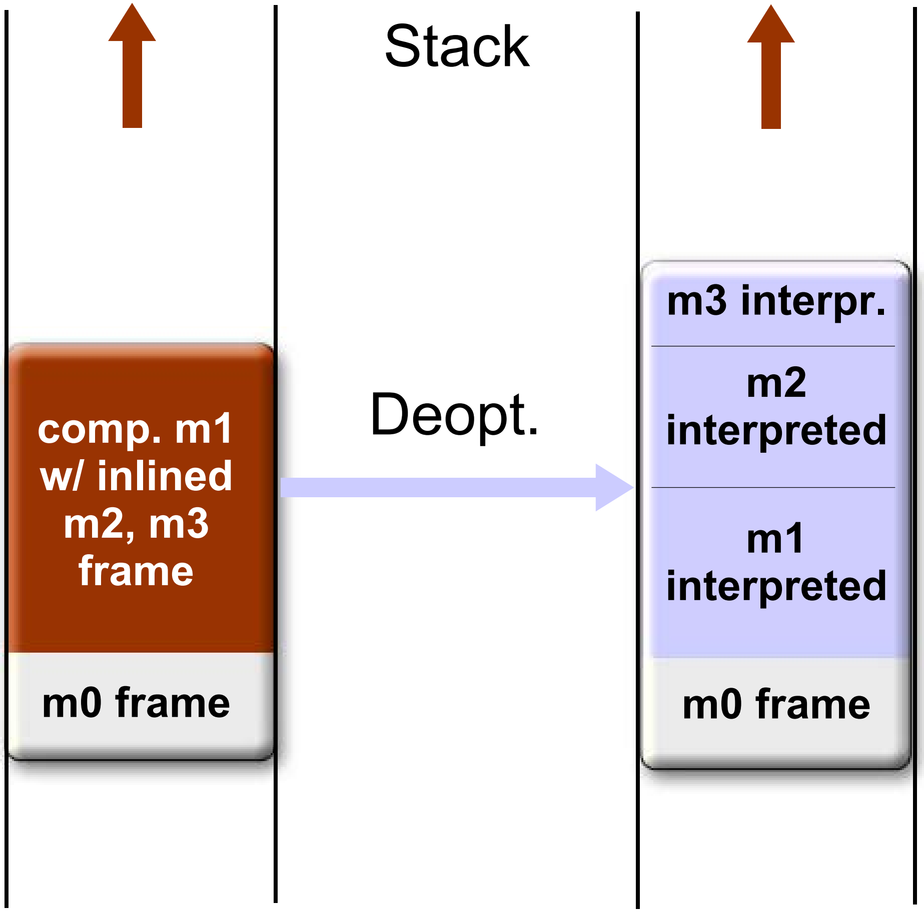
- Deoptimization

Deoptimization (2)

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

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

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



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

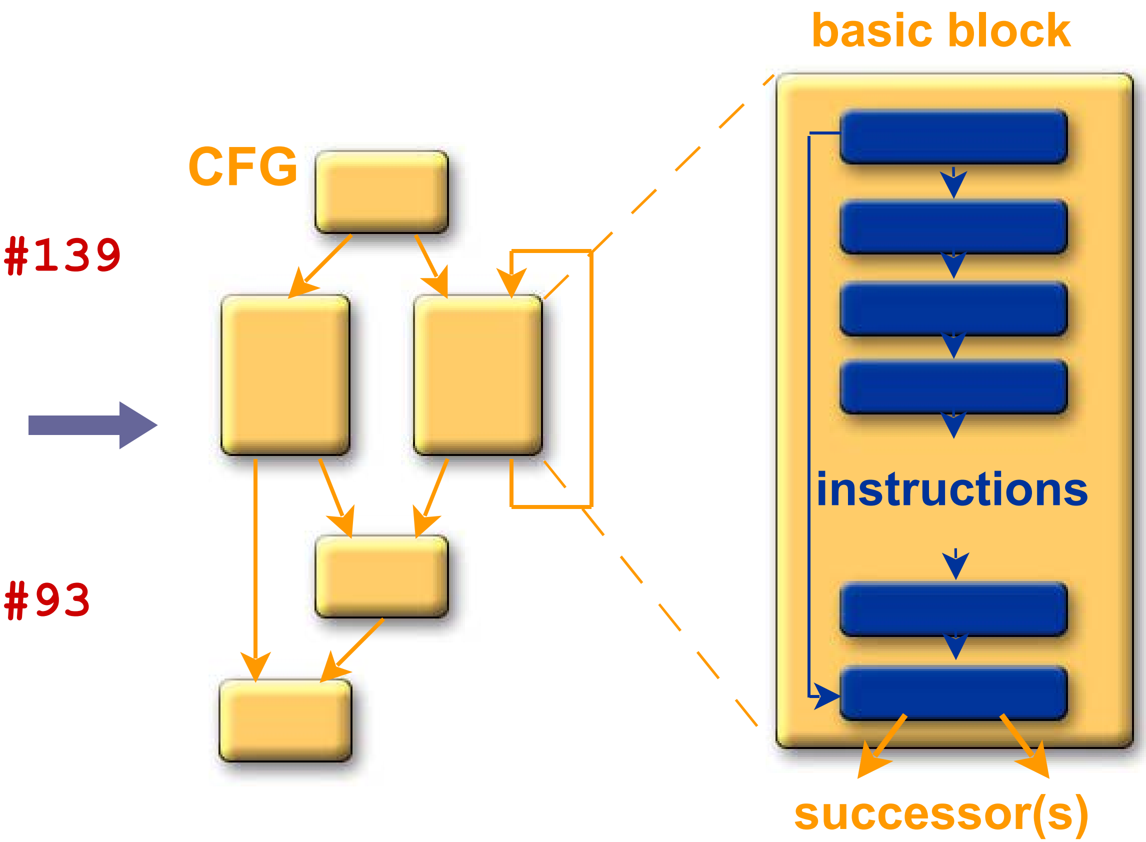
Front-End

- 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 aload_0
12 getfield #2
15 invokevirtual #93
18 aload_1
19 iconst_0
20 ...
```



CFG

basic block

instructions

successor(s)

Optimizations

Constant folding

Simple form of value numbering

Load elimination

Dead code elimination

Block elimination

Null check elimination

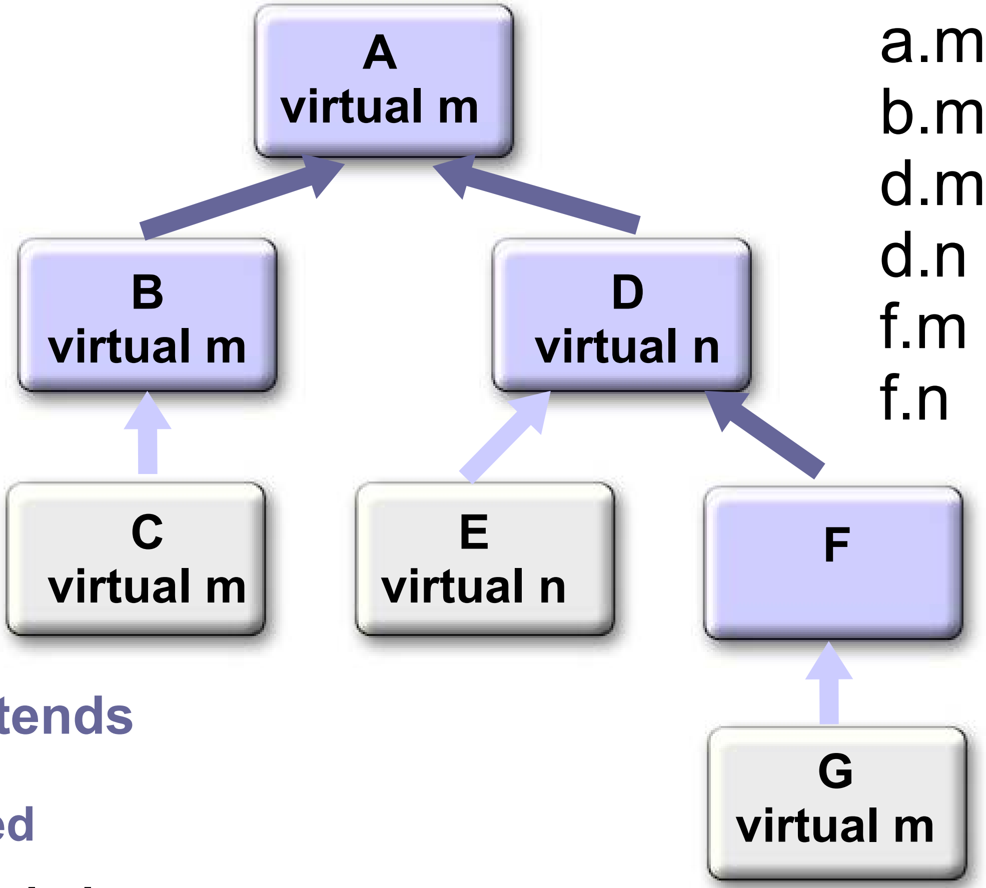
Inlining

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

Class Hierarchy Analysis (2)

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

→ extends

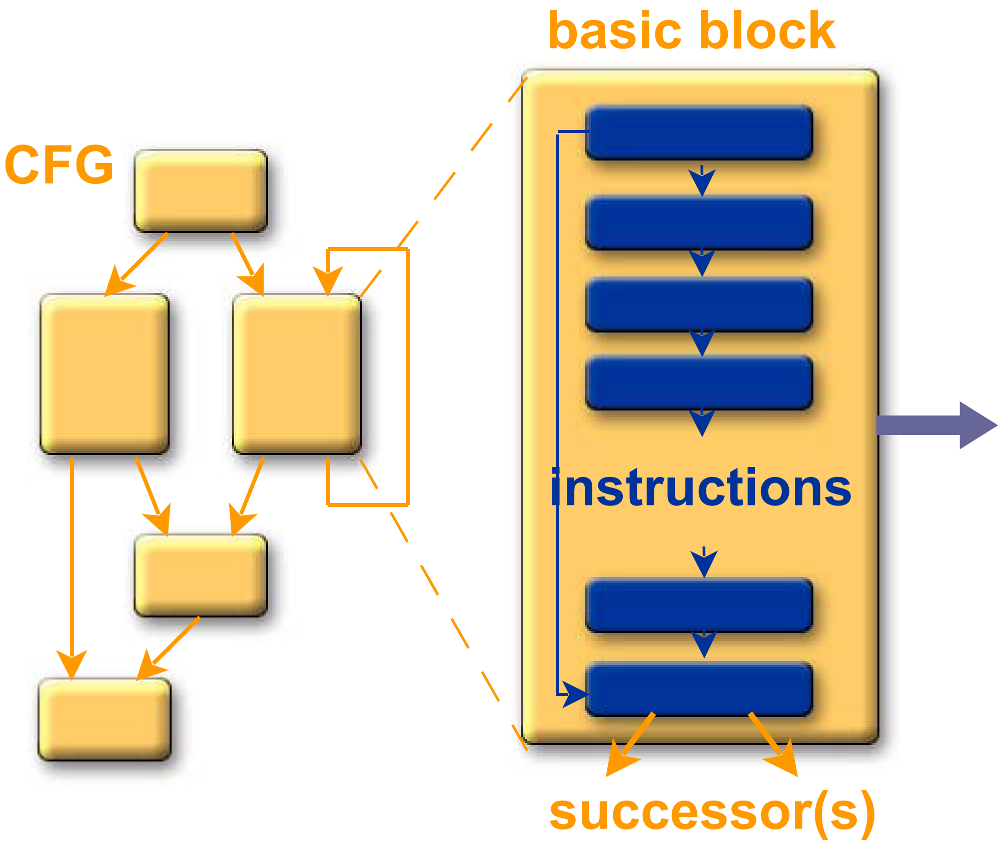
🟦 loaded

⬜ unloaded

Backend

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

Code Generation



```
Machine code:

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    0xaeef03
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

```
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

  Interpreted + native  Method
  7.2%      0  +    90  sun.java2d.loops.Blit.Blit
  0.7%      0  +     9  sun.awt.windows.Win32BlitLoops.Blit
  ...
 19.8%     72  +   174  Total interpreted (including elided)

     Compiled + native  Method
  9.2%    115  +     0  java.awt.GradientPaintContext.clip...
  ...
 15.0%    179  +     8  Total compiled (including elided)

  Thread-local ticks:
 51.7%   1330           Blocked (of total)
  0.2%      2           Class loader
  0.3%      4           Interpreter
 10.0%    124           Compilation
  0.5%      6           Unknown: running frame
  0.2%      2           Unknown: thread_state
```

Version Differences

Corresponding JDK™	1.3	1.3.1	1.4
Unified source base	n/y*	yes	yes
OSR	yes	yes	yes
Simple inlining	yes	yes	no
Full inlining	no	no	yes
Deoptimization	no	no	yes
More optimizations	no	no	yes

* 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

Robert Griesemer, Srdjan Mitrovic:
“A Compiler for the Java HotSpot™ Virtual Machine”, in *The School of Niklaus Wirth—The Art of Simplicity*, Morgan Kaufmann Publishers, 2000, ISBN 1-55860-723-4

Java HotSpot™ Technology Documents

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

Q&A

JavaOne℠

Sun's 2001 Worldwide Java Developer Conference