Java™ On Steroids: Sun’s High-Performance Java Implementation

Urs Hölzle
Lars Bak   Steffen Grarup
Robert Griesemer   Srdjan Mitrovic

Sun Microsystems

History

- First Java implementations: interpreters
  - compact and portable but slow
- Second Generation: JITs
  - still too slow
  - long startup pauses (compilation)
- Third Generation: Beyond JITs
  - improve both compile & execution time

“HotSpot” Project Goals

Build world’s fastest Java system:

- novel compilation techniques
- high-performance garbage collection
- fast synchronization
- tunable for different environments (e.g., low-memory)

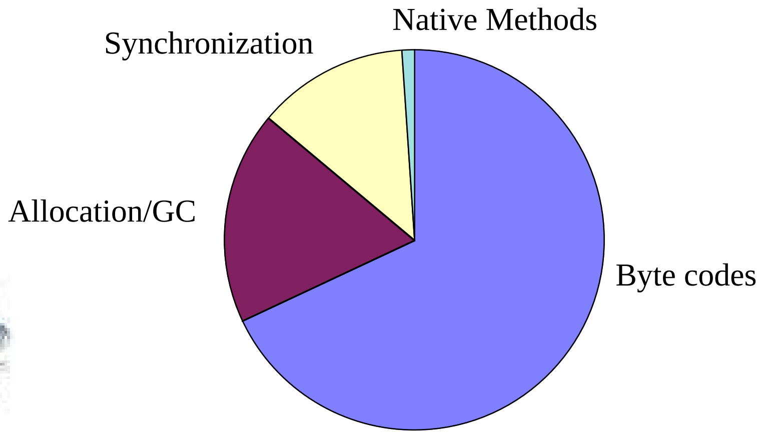
Overview

- Why Java is different
- Why Just-In-Time is too early
- How HotSpot works
- Performance evaluation
- Outlook: The future of Java performance

Why Java Is Different

- more frequent calls, smaller methods
  - slower calls (dynamic dispatch overhead)
  - no static call graph
  - standard compiler analysis fails
- sophisticated run-time system
  - allocation, garbage collection
  - threads, synchronization
- distributed in portable bytecode format

Example: javac



(executed with JDK interpreter)

Just-In-Time Compilers

- translate portable bytecodes to machine code
- happens at runtime (on the fly)
- standard JITs: compile on method-by-method basis when method is first invoked
- proven technology (used 10 years ago in commercial Smalltalk systems)

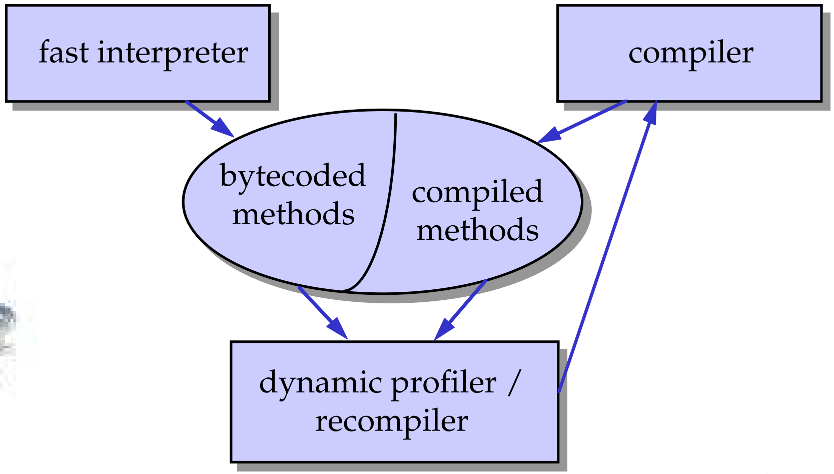
Why Just-In-Time Is Too Early

- problem: JITs consume execution time
- dilemma: either good code or fast compiler
  - gains of better optimizer may not justify extra compile time
- root of problem: compilation is too eager
  - need to balance compile & execution time

Solution: HotSpot Compilation

- lazy compilation: only compile/optimize the parts that matter
- combine compiler with interpreter
- seamlessly transition between interpreted and compiled code as necessary

HotSpot Architecture



HotSpot Advantages

- shorter compile time
- smaller code space
- better code quality
  - can exploit dynamic run-time information
- more flexibility (speed/space tradeoffs)

HotSpot Optimizing Compiler

- supports full Java language
  - all checks and exceptions, correct FP precision, dynamic loading, ...
- profile-driven inlining
- dispatch elimination
- many dynamic optimizations
- based on 10 years of research (Sun, Stanford, UCSB)

Garbage Collector

- accurate garbage collector
- fast allocation
- scalable to large heaps
  - generational GC
- incremental collection
  - typical GC pauses are less than 10 ms

Fast Synchronization

- software only
- extremely fast
  - up to 50x faster than others
- virtually no per-object space overhead
  - only 2 bits per object
- supports native threads, SMP

Performance Evaluation

- no microbenchmarks
  - but: limited set of benchmarks because HotSpot VM needs modified JDK
- all times are elapsed times
  - 200MHz Pentium Pro™ PC
  - warm file cache, best of three runs
- *preliminary data / prerelease software*

JVM Implementations

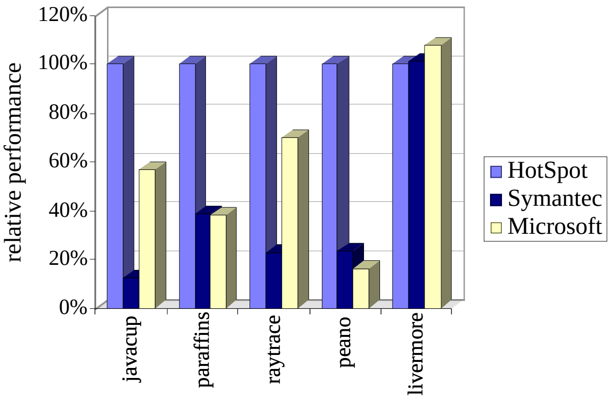
Systems measured:

- Pre-release “HotSpot” with next JDK
- Microsoft SDK 2.0 beta 2 (MS JDK 1.1)
- Symantec 1.5.3 JIT (JDK 1.1)

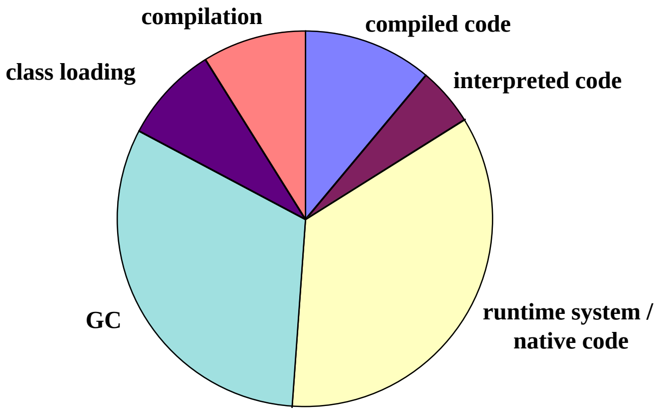
Caveats

- pre-release compiler & VM
  - functionally correct but untuned
  - but: implements full Java, no shortcuts for performance
- pre-release JDK libraries
  - VM needs new JDK
- other systems use different libraries
  - some are tuned; no JNI

Performance



Execution Profile (javacup)



CaffeineMarks: Just Say No

- small, artificial, C-like microbenchmarks
- no correlation to real Java programs
  - (almost) no calls, no dispatch, no allocation, no synchronization, no runtime system calls, ...
- easy target for compiler tricks
- prediction: we’ll soon see “infinite” CaffeineMarks

Hardware Wish List (Preliminary!)

- standard RISC is just fine, thanks
  - don’t penalize C code!!! (runtime system)
- large caches (esp. I-cache)
  - #1 performance booster
- reasonably cheap and selective I-cache flushing
- maybe some others (1-2% each)
- interpreters could use more support

Future of Java Performance

- performance will continue to improve
  - max. “typical” overhead 10-20% over C/C++
  - object-oriented Java programs will be faster than C++ equivalents
- JITs will be competitive with static compilers for most non-numerical apps
- next challenge: high-end SMP performance

Conclusions

- Java performance has improved dramatically in the past two years and will continue to improve further
- even performance-sensitive applications can use Java today
- Java does not need heavy architectural support to run efficiently
  - except in low-power, low-memory systems

Kudos

- David Ungar and the Self project
  - http://self.sunlabs.com
- David Griswold, Tim Lindholm, Peter Kessler, John Rose
- JavaSoft’s JVM & JDK teams