The Java™ HotSpot™ Performance Engine - A Technical Tour

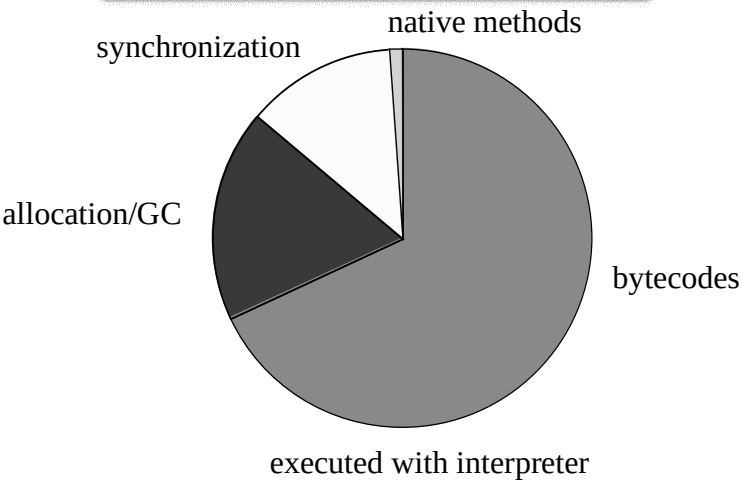
Robert Griesemer
Staff Engineer
Sun Microsystems, Inc.

The Java Programming Language

- platform-independent
- object-oriented
- multi-threaded
- dynamic class loading
- safe

© Sun Microsystems, Inc., 1999

Example: javac



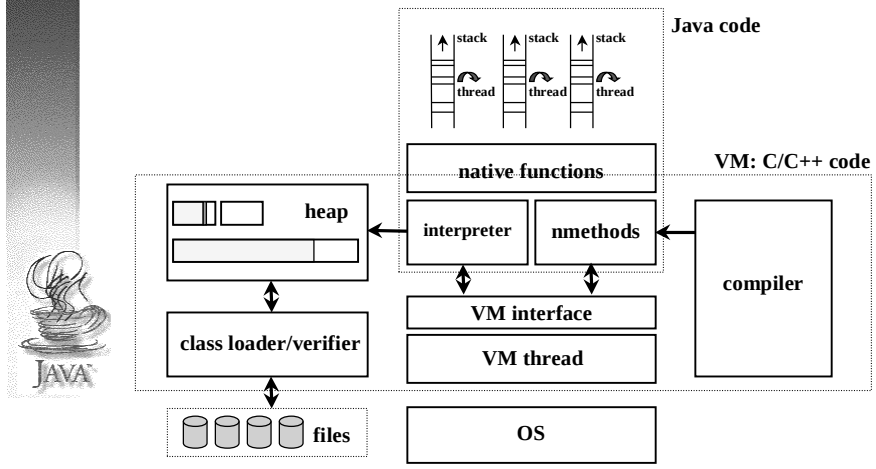
© Sun Microsystems, Inc., 1999

Talk Overview

- dynamic code generation
- CHA & inline caches
- fast allocation/GC
- synchronization
- backpatching
- deoptimization

© Sun Microsystems, Inc., 1999

VM Overview



© Sun Microsystems, Inc., 1999

Dynamic Code Generation

- overcome speed limitations of interpreters
- static compilation is not an option
  - dynamic class loading!
- more information available at run-time than statically

© Sun Microsystems, Inc., 1999

Just-In-Time Compilers

- translate bytecodes to machine code
- happens at runtime
- standard JITs: compile method when first invoked
- proven technology

© Sun Microsystems, Inc., 1999

The Trouble with JITs

- problem: consume execution time
- dilemma: either good code or fast compiler
- root of problem: compilation is too eager
  - balance compile & execution time
- can compilation cost be amortized?

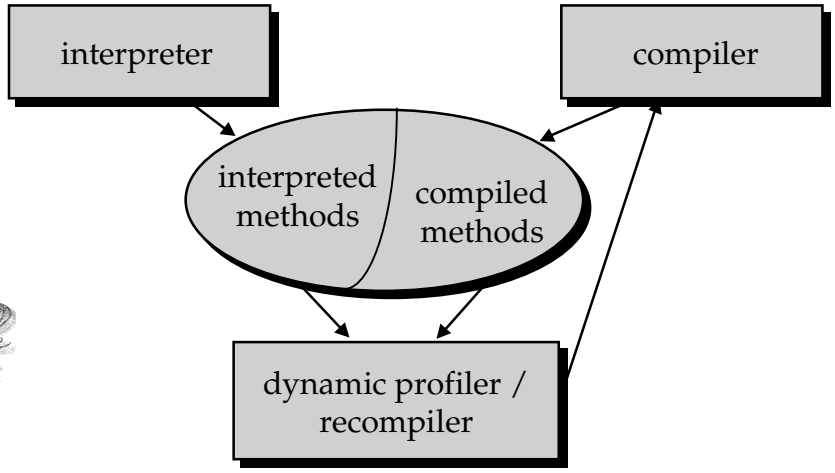
© Sun Microsystems, Inc., 1999

A Lazy Approach

- only compile/optimize the hotspots
- combine compiler with interpreter
- seamless transition between interpreted and compiled code
- predict the future by looking at the past

© Sun Microsystems, Inc., 1999

HotSpot Compilation Architecture



© Sun Microsystems, Inc., 1999

Advantages Over Standard JIT

- less to compile
  - faster startup
  - smaller code space
  - more time left for optimizations
- better code quality
  - use dynamic run-time information
- more flexibility
  - speed/space tradeoffs

© Sun Microsystems, Inc., 1999

Why Have an Interpreter?

- space
- startup time
- “source-level” debugging
- portability
- compiler can be simpler
  - deoptimization

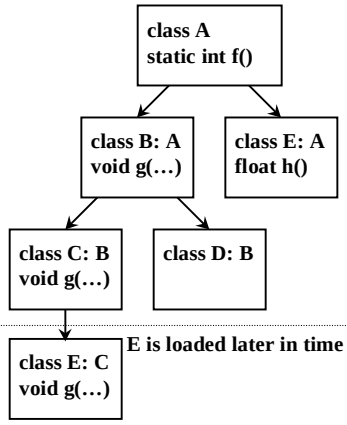
© Sun Microsystems, Inc., 1999

More Opportunities for Dynamic Code Generation

- interpreter
  - fine-tuning on machine code level for better performance
  - generated at startup time for flexibility and robustness
- runtime interface stubs
- dynamic stub-routines

© Sun Microsystems, Inc., 1999

Class Hierarchy Analysis

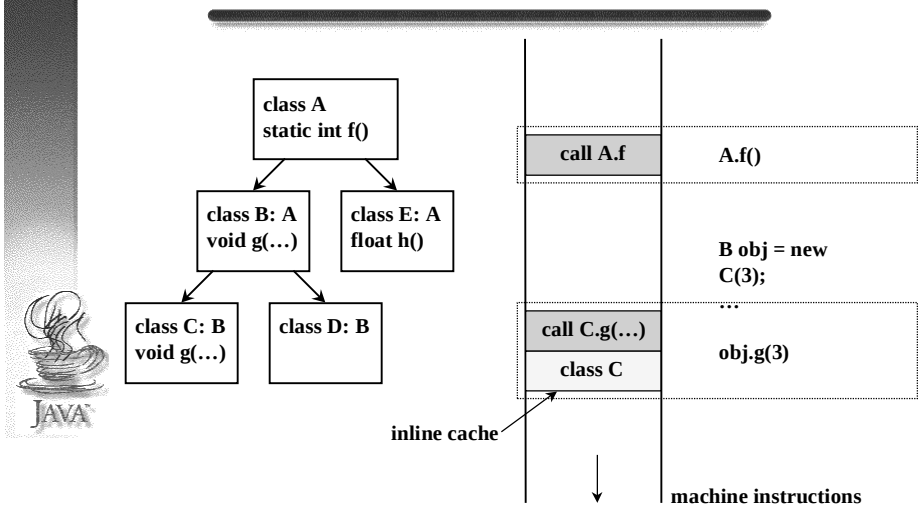


- f can be statically bound or inlined
- g must be dynamically dispatched for recv of **static type B**
- g can be statically bound or inlined for recv of **static type C or D**
- h can be statically bound or inlined

Note: these observations may not be true anymore after class loading!

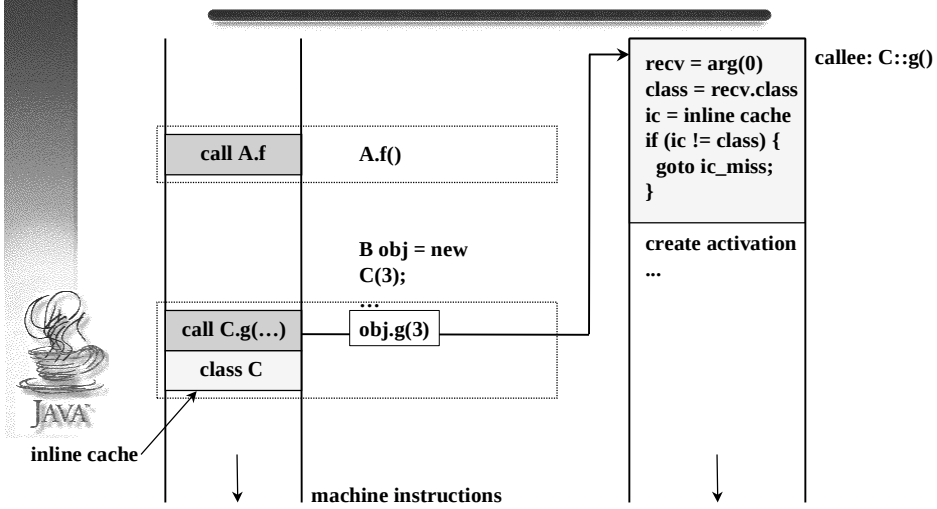
© Sun Microsystems, Inc., 1999

Inline Caches: Caller



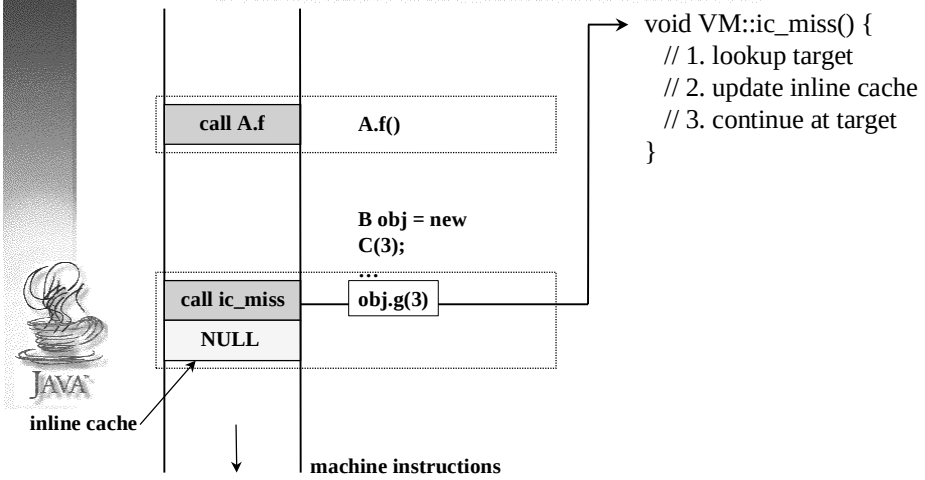
© Sun Microsystems, Inc., 1999

Inline Caches: Callee



© Sun Microsystems, Inc., 1999

Inline Caches: First Time



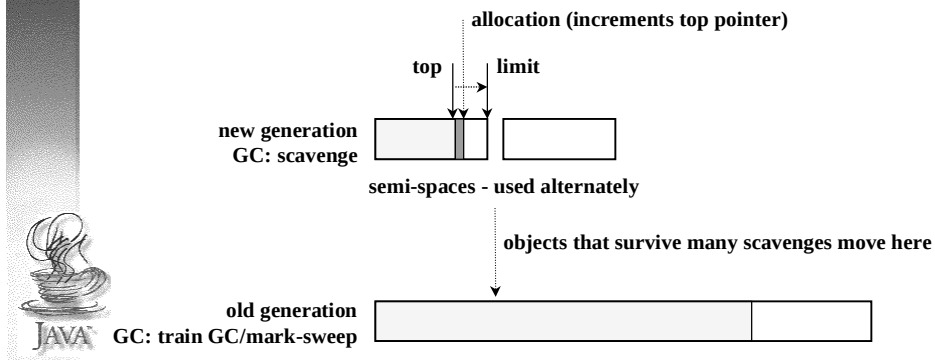
© Sun Microsystems, Inc., 1999

Fast Allocation

- smallest object is 2 words
- no handles in Java code
- accurate GC
  - compaction possible & required for fast allocation
- allocation = increment 1 pointer
- allocation code inlined

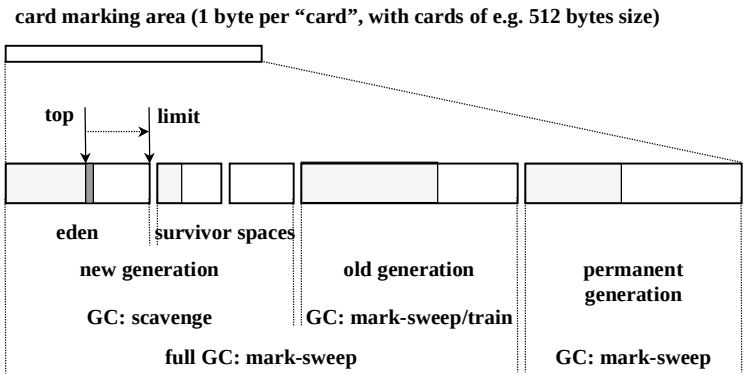
© Sun Microsystems, Inc., 1999

Heap: The Big Picture



© Sun Microsystems, Inc., 1999

Heap: (Almost) the Real Story



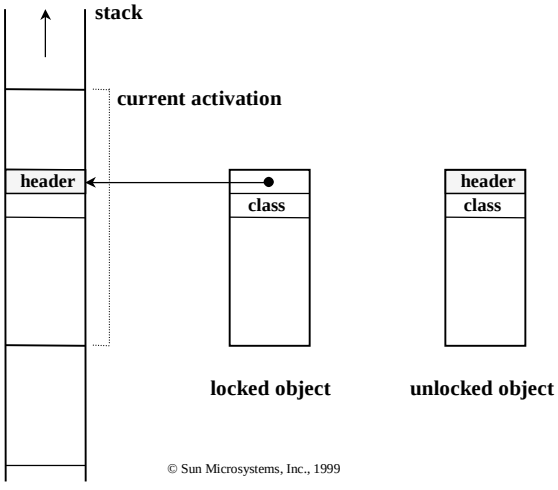
© Sun Microsystems, Inc., 1999

Fast Locking

- lock usage is “method” structured
  - locks can be stack-allocated
- software only
- virtually no per-object space overhead
  - 2 bits per object
- supports native threads, SMP

© Sun Microsystems, Inc., 1999

Fast Locking



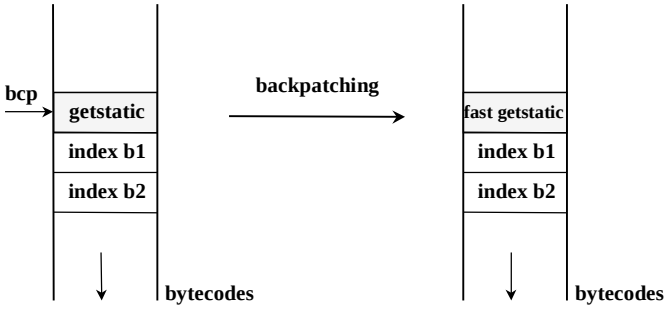
© Sun Microsystems, Inc., 1999

Backpatching

- code changed during execution
  - used by interpreter (“quickening”)
  - used by compiler (e.g., inline caches)
- can avoid repeated checks
- must be very careful with MT/MP
- “undo” possible

© Sun Microsystems, Inc., 1999

Backpatching in the Interpreter



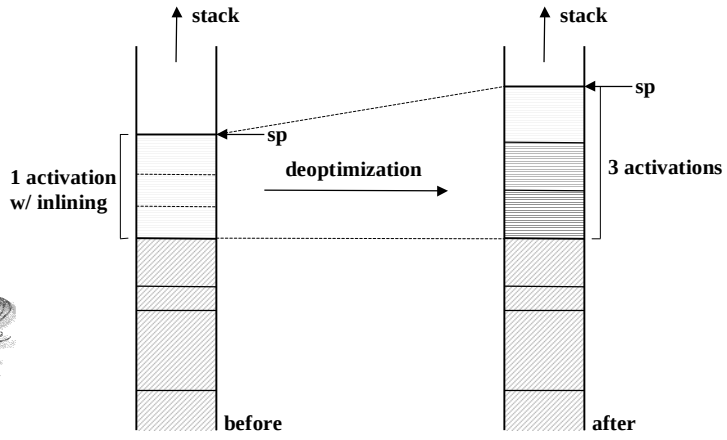
© Sun Microsystems, Inc., 1999

Deoptimization

- optimizations may need to be undone
- reflective information in compiled code
- deoptimization: compiled activations => interpreter activations
- required to do optimistic optimizations
- inverse: on-stack replacement

© Sun Microsystems, Inc., 1999

Deoptimization of a Stack Frame



© Sun Microsystems, Inc., 1999

Observations

- there is no silver bullet
  - must optimize many parts of a system
- careful balance of optimizations
  - optimizing A may slow down B
- complexity requires rigorous programming style

© Sun Microsystems, Inc., 1999

Conclusions

- HotSpot Technology is a reality & solves Java's challenges
- dynamic code generation has become standard technology
- Java performance has improved dramatically

© Sun Microsystems, Inc., 1999

References

- The Java™ HotSpot™ Performance Engine
  - http://java.sun.com/products/hotspot/
  - JavaOne talks and White Paper
- The Self project
  - http://self.sunlabs.com/
  - links to many Self-related papers

© Sun Microsystems, Inc., 1999