



# The HotSpot Virtual Machine

---

- Virtual machine for Java from Sun
- Overview of technology
- Benchmarking



# History of HotSpot

---

- The Self Project, 1989-1995
  - Research from Stanford University and Sun Microsystems Laboratories Inc.
- Longview Technologies, 1994-1997
  - Startup acquired by Sun 1997
- Java Software Division, Sun Microsystems
  - Integration into Java SDK

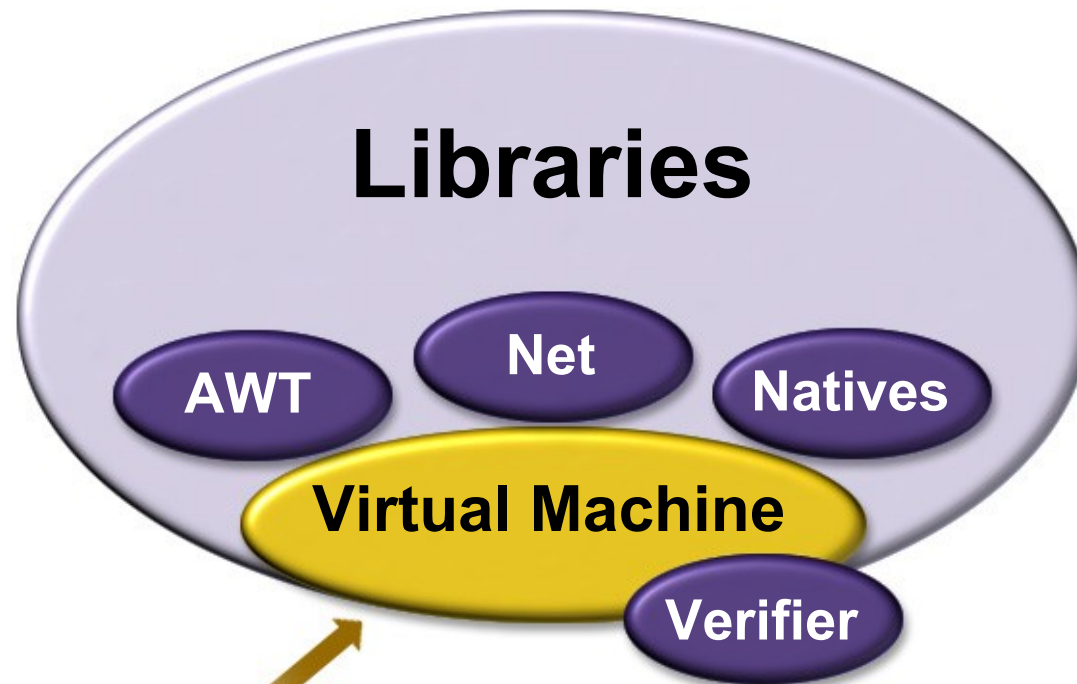


# What Is HotSpot?

---

- Java™ Virtual Machine (JVM)
  - Fast
  - Scalable
  - Robust
  - Compliant
  - Portable

# Java™ 2 Platform, SE



Java HotSpot™  
Performance Engine



# Performance

---

## Where does time go?

- Byte code execution
- Runtime system
  - Garbage collection
  - Thread management
- Native libraries
- Operating system

# Obstacles to Performance



---

- Platform independent byte codes
- Object oriented
  - Virtual calls are the default
  - High call frequency
  - High allocation rate
  - Garbage collection
- Synchronization in language
- Dynamic class loading



# Technology

---

- ➔ ■ Adaptive compilation
- Highly optimizing compilation
- Efficient memory management
- Fast synchronization

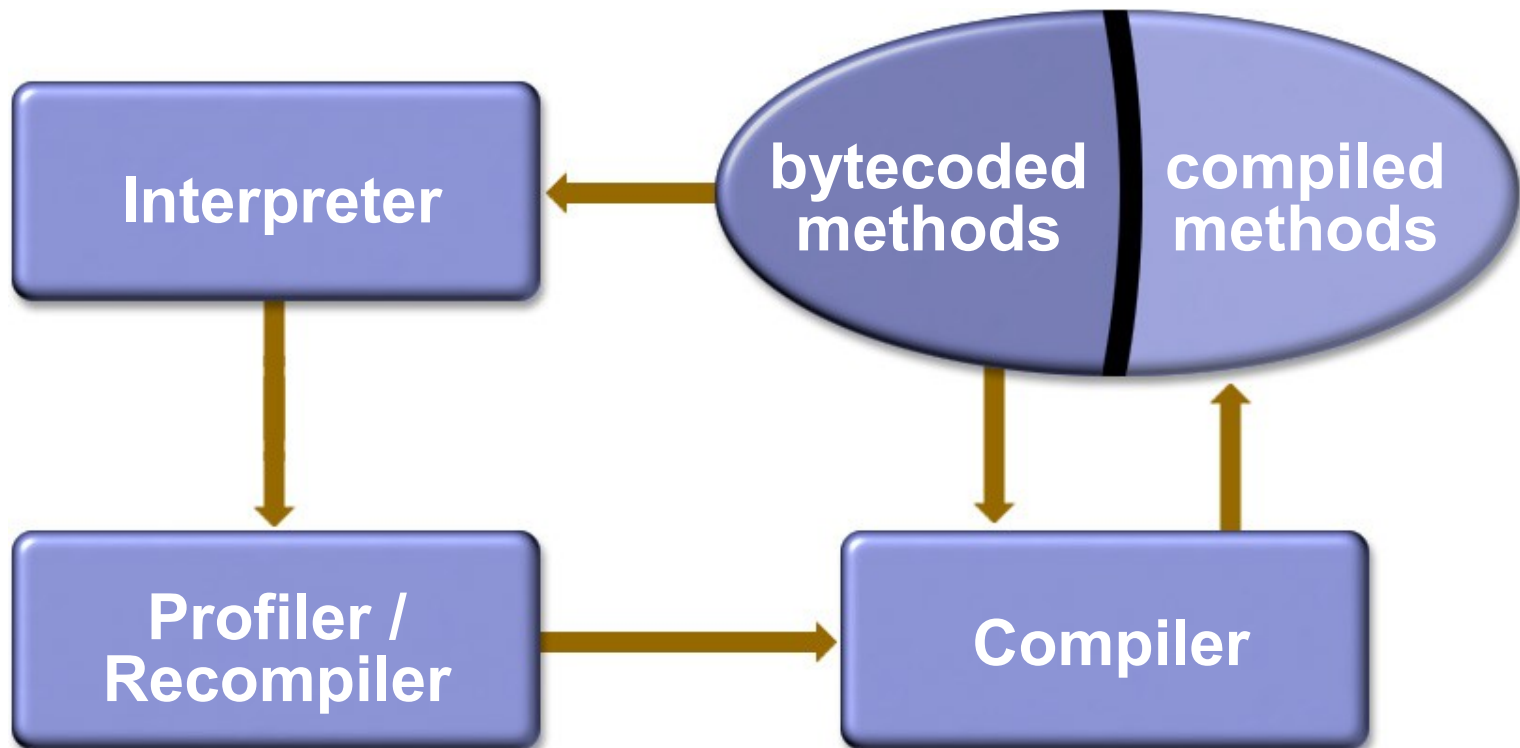


# Problems with JITs

---

- Compilation hurts:
  - Start-up time
  - Memory
- Good code vs. fast compiler
- Favors bad programming style
  - Use of final
  - Break down abstractions

# Adaptive Execution Engine





# Interpreter

---

- Fast interpreter is important
- Generated dynamically
  - Pipeline specific assembly
- High performance
  - “Top of Stack” caching



# Adaptive Analysis

---

- All methods have invocation counters
- Incremented on entry and backward branch
- Invocation overflow invokes compiler
- Call-stack analysis selects root of compile



# Adaptive Compilation

---

- Only compile hotspots
  - Dynamically profile
  - Still interpret when sensible
- Compile fewer methods
  - More aggressive optimizations
  - Spreads out compilation pauses
  - Preserves memory

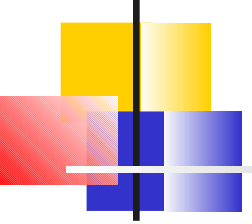


# Technology

---

- Adaptive compilation
- ➔ ■ Highly optimizing compilation
- Efficient memory management
- Fast synchronization

# Optimizing Compiler

- 
- 
- SSA intermediate representation
  - Global graph coloring register allocator
  - Precise debugging model
  - Support for exact garbage collection
  - Retargetable



# Aggressive Inlining

---

- Goal: bigger basic blocks
- Inlining of virtual methods
  - Class loading can invalidate code
  - Dependencies must be tracked

```
class A {  
    int foo() { return 3; }  
}
```

# Dynamic Deoptimization

- Permits backing out of optimizations

```
class B extends A {  
    int foo() { return 6; }  
}
```

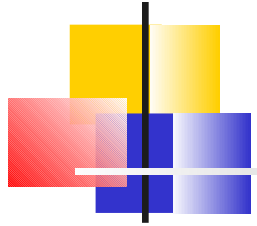
- Transforms compiled activations into interpreted activations



# On Stack Replacement

---

- Inverse of deoptimization
- Transforms interpreted activations into compiled activations
- Relevant only to microbenchmarks



# Technology

---

- Adaptive compilation
- Highly optimizing compilation
- ➔ ■ Efficient memory management
- Fast synchronization

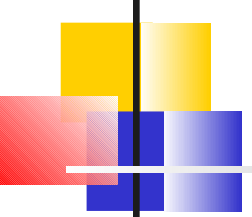


# Memory Management Problems

---

- Existing problems
  - Not scalable to large heaps
  - Long pauses
  - Object fragmentation
  - Slow allocation
  - Memory leaks
- Results in bad programming style
  - Use of free lists

# HotSpot Memory Management

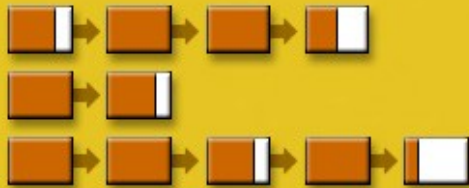
- 
- 
- Accurate garbage collection
    - Safe-points in runtime and compiled code
  - Fast allocation
  
  - Collection scales to large heaps
    - Generational
    - Incremental

# Three Collectors at Work



**New generation**

**Train generation**



**Permanent generation**

- Copying collector
  - Efficient since objects die young
- Train algorithm copying collector
  - Incremental collection of old objects
- Mark-compact collector
  - Relocate objects in place
- Typical pauses <10ms
- Software write barrier

# Technology

---

- Adaptive compilation
- Highly optimizing compilation
- Efficient memory management
- ➔ ■ Fast synchronization



# Fast Synchronization

---

- Performance problem
  - Many basic libraries are multi thread safe
  - Data structures are protected by synchronized operations
  - Locking and unlocking are very frequent
    - Most operations are uncontended

# Synchronization in Java



---

- Two ways to synchronize:

```
synchronized void foo() {
```

```
    ...
```

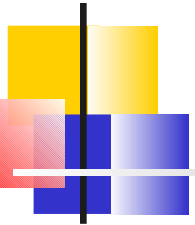
```
}
```

```
synchronized(obj) { ... }
```

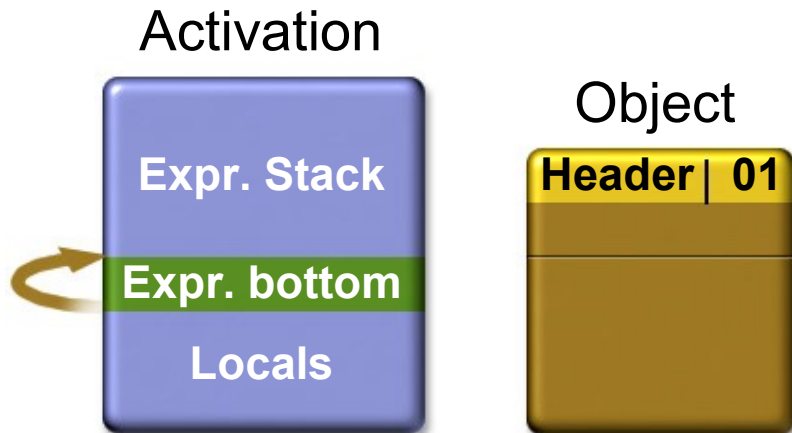
- Both ways are block structured

*Idea: allocate monitor on stack*

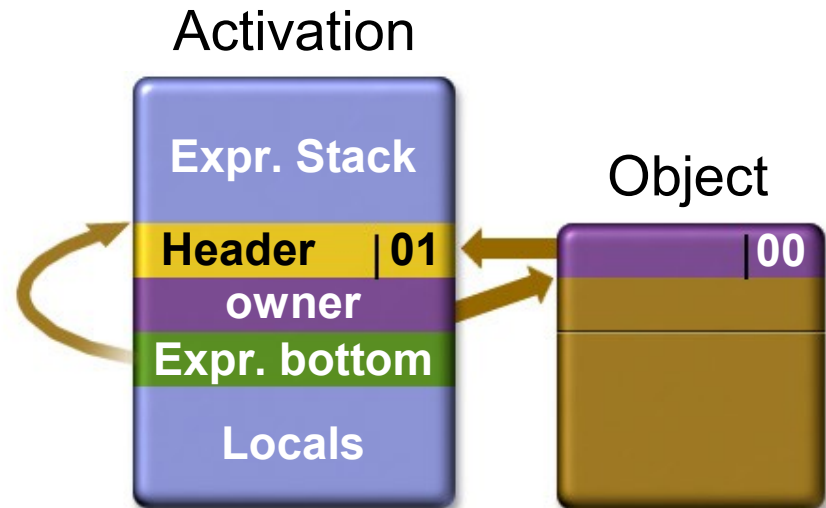
# Locking an Object

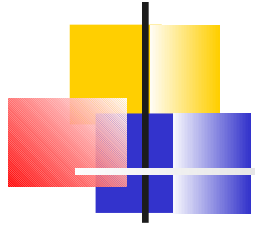


## Unlocked



## Locked





# Contended Monitor Case

---

- Header word of object is upgraded to refer a “heavyweight” monitor
- The “heavyweight” monitor contains OS-level mutex/condition pair



# Fast Synchronization

---

- Common cases fast
  - Uncontended lock
  - Re-acquire lock from same thread
  - Unlocking
- Minimal space overhead per object
  - 2 bits in the object header
- Use stack pointer as thread identifier

# Java HotSpot™ Conclusion



---

- Fast
- Scalable
- Reliable
- Compliant

*Raises the performance bar for  
Java Virtual Machines*



# Research Papers

---

- **Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming**, *by Urs Hölzle*  
Ph.D. thesis, Computer Science Department, Stanford University
- **Debugging Optimized Code with Dynamic Deoptimization**, *by Urs Hölzle, Craig Chambers, and David Ungar*  
Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, 32-43, San Francisco, June, 1992
- **Tenuring Policies for Generation-Based Storage Reclamation**, *by David Ungar and Frank Jackson*  
Proceedings of OOPSLA 1988: San Diego, California, 1-17
- **Incremental Collection of Mature Objects**, *by Richard L. Hudson and J. Eliot B. Moss*  
IWMM 1992: 388-403
- **A Fast Write Barrier for Generational Garbage Collectors**, *by Urs Hölzle*  
OOPSLA 1993 Workshop on Garbage Collection, Washington, D.C., October 1993
- **A Simple Graph-Based Intermediate Representation**, *by Cliff Click*.  
Proceedings of the Intermediate Representations' 95 Workshop, pages 35-49.
- **Global Code Motion, Global Value Numbering**, *by Cliff Click*.  
Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation '95.