Modern Virtual Machine Performance

- murphee (Werner Schuster)
- http://jroller.com/page/murphee

Overview

- Virtual Machine Myths
  - Myth: Java is interpreted
  - Myth: Native code is always faster than...
  - Myth: Garbage Collection is slow/has overhead/...
- State of the art Dynamic Compilers
  - Hotspot & Co
  - Java (safety) restrictions and their solutions
- Runtime Code Generation/Specialization
  - A bit of theory: Retargetable Compilers
  - Examples: JSP, XSLT,...

Overview/2

- Garbage Collection
  - Generational Garbage Collectors
  - No more pauses with concurrent collection
- Benchmarking
  - Know what you test
  - Microbenchmarks and Dynamic Compilers
  - Sample
- Misc Java Performance Tips
  - Forget MicroTuning
  - Watch your Strings

Myth: Java is interpreted

- JDK 1.0, JDK 1.1
- JITs after 1.1.6
- Interpretation in Hotspot for initial execution
  - execution is profiled and code is compiled if it's a Hotspot
- Interpretation on devices with restricted power
- Hardware

Myth: Native is always faster than...

- Great. Java is compiled to native code at runtime.
  - simple JIT compilers get rid of interpretation dispatch overhead
  - Dynamic Compilers produce highly optimized native code that is executed
- Native Code(x) != Native Code(y)
  - Native Code (Optimized) != Native Code (NonOptimized)

Dynamic Compilers

- Just In Time – Compilers
  - Compiles method at first use – method stays in memory after that
  - Have little time to work
- Dynamic Compilers
  - Profile code (method counters)
  - Compile only code that is used a lot
  - Basic Idea:
    - Compiler can take more time to optimize code, thus yields better, faster code

Optimizations/OOP

- Virtual Dispatch
  - Virtual Inlining == Inlining of Virtual methods
  - Inline Caches/Polymorphic Inline Caches
    - reduce cost of virtual dispatch
- Technology
  - Class Hierarchy Analysis
  - Deoptimization
  - OSR - OnStackReplacement

Optimizations/ABC

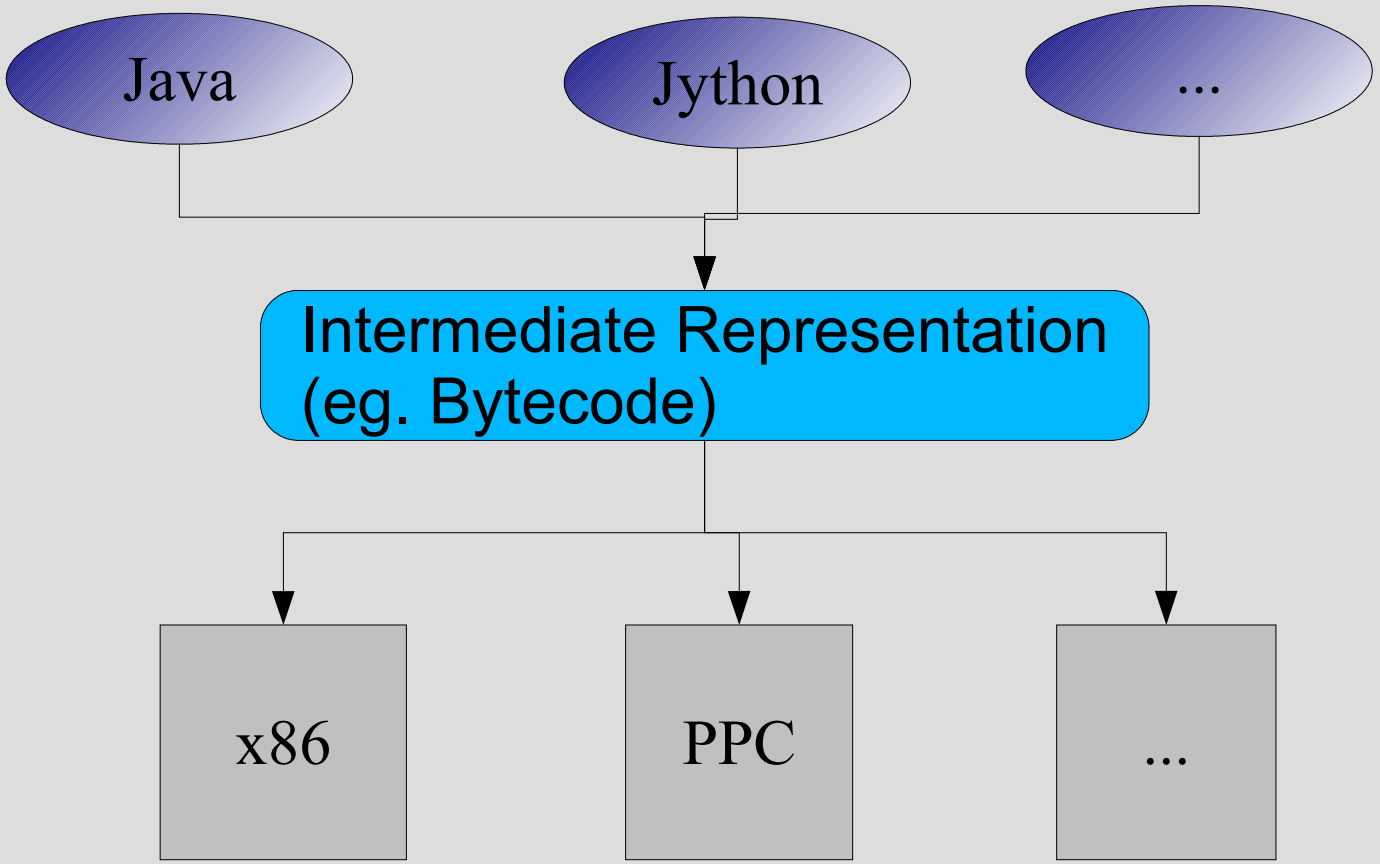
- ArrayBoundsChecking Removal
- In Java each `foo[i]` access must check
  - i >= 0
  - i < foo.length
  - else: Exception
- Can be removed if index is known to be in the allowed range
  - in loops
  - index a constant

Optimizations/Synchronization

- Synchronization Removal
- `synchronized` methods or `synchronized (x)` blocks
- If the lock (ie. some object) is thread local
  - No Contention (ie. no other threads **can** access it)
  - Thus: locking useless and can be removed

Retargetable Compiler

Retargetable Compiler



Languages for the JVM

- Jython, Jruby, Rhino (Ecmascript), ...
- More?
  - http://www.robert-tolksdorf.de/vmlanguages.html
- JVM as platform
  - Unix has C as system language
    - syscalls have C semantics
    - structures are represented as C structs
  - Vast amount of code and libraries available

Runtime Code Generation and Specialization

- Specialized bytecode is generated and loaded at runtime
  - compiled by dynamic compiler and linked to existing code
  - runs as optimized native code
- Existing Uses
  - JSP
  - XSLTC (Java 5.0+)
  - Sun JFluid Profiler

Runtime Code Specialization

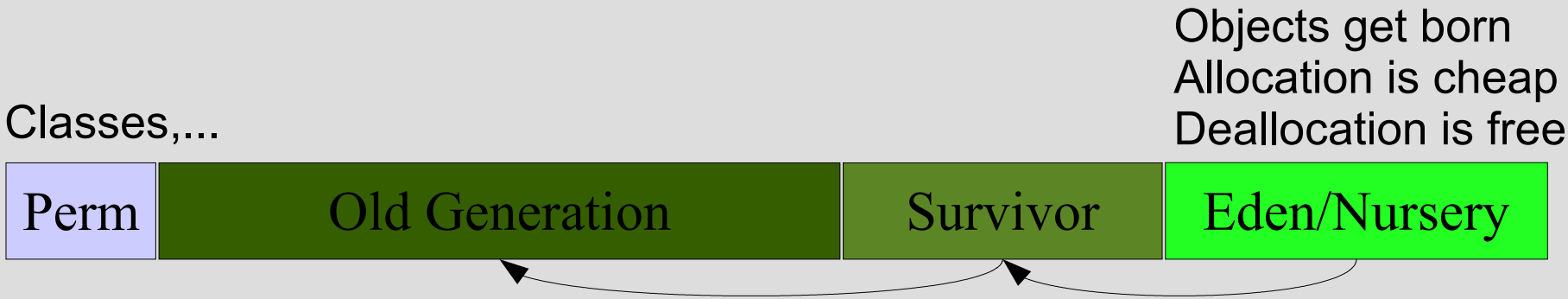
- Performance Advantages
  - less branches from lookups or interpretation
  - inlining easier
- VM and GC make this easy
  - less risk for generated code damaging something
- Further Reading:
  - http://citeseer.ist.psu.edu/massalin92synthesi.html
  - http://www.cse.ogi.edu/DISC/projects/synthetix/overv

Myth: GC is always slow/has overhead/...

- Java 1.0: Simple StopTheWorld MarkSweep FreeList Allocator
  - Overhead for allocation and deallocation
  - Pauses (especially for large heaps)
- Java 1.5: Too much to list...
  - Generational GC makes allocation very cheap, deallocation free (for short lived objects)
  - Incremental/Concurrent GC reduce pauses or eliminate them at all
  - Compacting reduces fragmentation and improves locality

Garbage Collection

Generational Garbage Collection



Tip: Check out JConsole in Java 5.0

Concurrent GC/1

- Basic GC algorithm:
  - `new( )` requests come in, memory gets allocated
  - Until: no memory left, which means
    - Stop The World = threads are stopped
    - Mark & Collect Garbage
- Problem: “StopTheWorld” pauses
  - are particularly bad for apps that must be responsive, eg GUI apps
  - gets worse with larger heaps (longer collection times)

Concurrent GC/2

- Solution: Concurrent GC (sometimes called “Incremental”)
- Algorithm:
  - Stop threads for a **short** time
  - Mark Garbage
  - Continue threads
  - Stop threads for a **short** time
  - Remark
  - Collect

Concurrent GC/3

- Benefit
  - Pauses can be kept short so they cannot be noticed
- Cost
  - concurrent GC does a little more work to avoid long pauses
  - if pauses are irrelevant, GC can be tuned for throughput
  - http://java.sun.com/docs/hotspot/gc5.0/ergo5.html
  - http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.l

Sample

Java Code: Parser + AST Builder, built with ANTLR

Equivalent code: handcrafted C/C++
(Note: don't do exactly the same, but can be compared)

Test Code

```
e = exprParser.parse(new FileInputStream(“foo.m”));
```

Java Version: 1300 ms

C/C++ Version: 200 ms

Sample

Identified Problem

FileInputStream does not buffer, thus each read() causes a syscall.

Test Code

```
fis = new FileInputStream(“foo.m”);
e = exprParser.parse(new BufferedInputStream(fis));
```

Java Version: 950 ms

C/C++ Version: about 200 ms

Sample

Identified Problem

Test code was only run once, in a “cold” JVM (newly started JVM).

Test Code

```
for(int x = 0; i<10; i++){
fis = new FileInputStream(“foo.m”);
e = exprParser.parse(new BufferedInputStream(fis));
}
```

Java Version: 1st loop iteration: 950 ms,
2nd loop iteration 250 ms (!)

C/C++ Version: about 200 ms

Performance Tips/Locals

- Micro-Optimizations: Just say no

```
String foo;
while(something){
   foo = getMeSome Foo()
   // do something with foo
}
```

```
while(something){
   String foo = getMeSome Foo()
   // do something with foo
}
```

There's no difference. Period.

For sceptics:
http://www.javalobby.org/java/forums/m91823466.html
(If you don't want to read all... read only my postings, they should explain the matter).

Performance Tips/Strings

- String objects are immutable
  - Creation of a String means all its data is in memory
  - At design time:
    - Always think whether Strings are the best way to go
    - If lots of data is needed/handled, check out CharSequence, CharBuffer, ...
- “+” and “+=” for Strings
  - OK for low frequency concatenation
  - Careful if used in loops
    - **Maybe** using StringBuffer is better (depends on situation)

Links

- Jikes RVM
  - http://jikesrvm.sourceforge.net/info/papers.shtml
  - Lots of papers and research about Garbage Collection, JIT or Dynamic Compilers, Virtual Machines,...
- Sun Hotspot
  - http://java.sun.com/docs/performance/index.html
  - All about Hotspot and Sun JVM performance
- Java Performance Tuning
  - http://www.javaperformancetuning.com/
  - By the writers of “Java Performance Tuning”, monthly newsletters with tips and links
- Anatomy of a flawed microbenchmark
  - http://www-128.ibm.com/developerworks/java/library/j-jtp02225.html?ca=c
  - Benchmarking is hard...