Java Virtual Machine

Session 3, Wednesday November 3rd, 2010

Robin Schirrmeister

Outline

- Basic Idea
- Reasons
- More Details
- Bytecode
- Advantages and Disadvantages
- Benchmarking Problems
- Benchmarking Framework
- Performance Tips / different Implementations

Basic Idea of the Java Virtual Machine

Quellcode

```
class Student
{
  public int matrNr;
  Student(int matrNr)
  {
     this.matrNr = matrNr;
   }
....
}
```

compiled to

Bytecode

```
class Student extends
java.lang.Object{
public int matrNr;
Student(int);
  Code:
   0:    aload_0
   1:    invokespecial  #1;
   4:    aload_0
   5:    iload_1
   6:    putfield       #2;
   9:    return
}
```

Is loaded and verified by

JVM

Compiles Bytecode and executes it on

Reasons for the Java Virtual Machine

- Cross-Platform
  - Machine-independent Bytecode
  - „Write once, run anywhere“
- Security Reasons
  - Execute an unknown program, don't damage the PC
    - Restricted applets
    - Code safety

Java Virtual Machine – more precise

- JVM not a program, but a specification how to load and run bytecode files
- => different JVMs for different Platforms and for same platform by different vendors
- Modern JVMs have many optimizations (JIT/Hotspot...)
- Bytecode can be compiled from any language, not just Java! (e.g. Jython, Jruby, Jscheme, Groovy, Scala)

Overview JVM Specification Structure

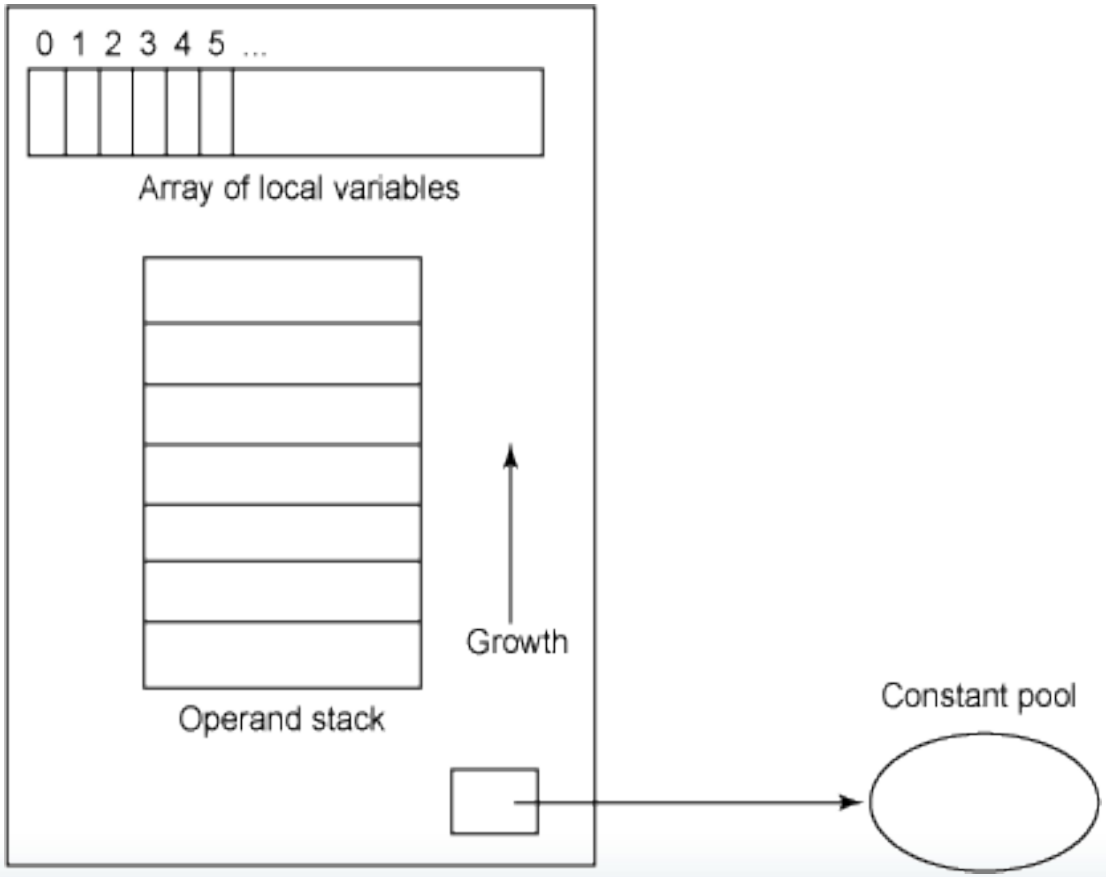
- Important Parts of the JVM Specification:
  - Structure of the class files
  - Bytecode instructions and data types
  - Security requirements (verification)
  - Data areas
  - Rules for threads and concurrency

Data Areas

- Stack
  - Local variables of the functions
- Heap
  - Instances of objects
- Method Area
  - Code of the methods
- Runtime Constant Pool
  - Class variables, e.g. static variables

Methods in the JVM

- Method Data Structures



Bytecode and Verification

- Bytecode structure
- Decompilation with javap -c [-private]
- Compilation possible with Jasmin
- Bytecode instructions

Advantages/Disadvantages of JVM Concept

- Advantages
  - Cross-platform, machine-independent
  - Higher code safety
  - Advanced optimizations possible for the JVM
- Disadvantages
  - Extra-compilation step does slow performance down (big question: how much? :))
  - Unable to use certain OS-APIs directly
  - Unable to hand-optimize as much

Problems with Benchmarking

- JVMs today have many complicated optimizations => Benchmarking to get meaningful results is very tricky
- First problem: The time-measurement used
  - System.currentTimeMillis has platform dependent-resolution, e.g. on Windows XP multicore ~15 ms, System.nanoTime can be better, both can have own overhead

Problems with Benchmarking 2

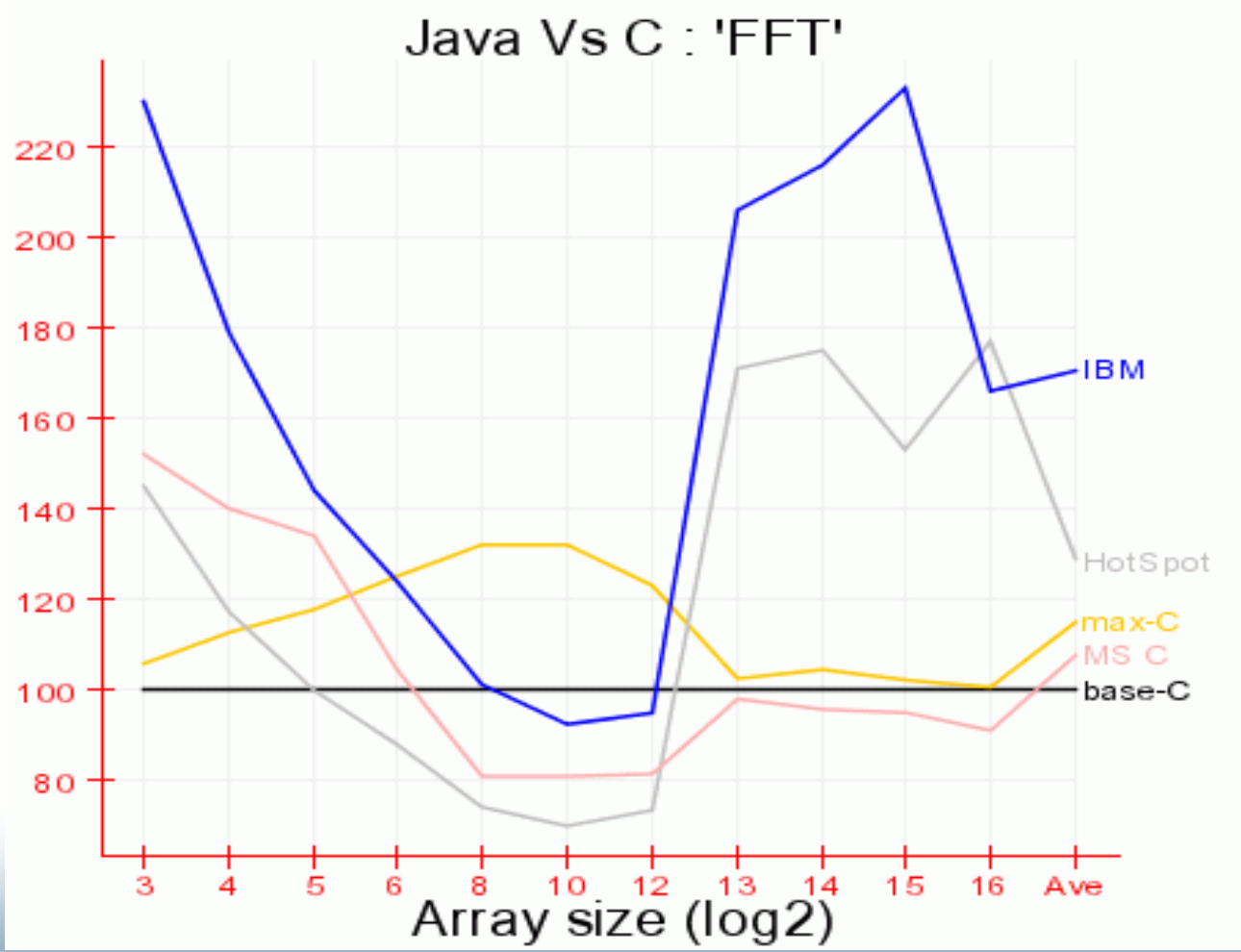
- Warmup Time
  - JVMs often load classes only on first use => first time a task is run can be much slower than next times
  - Similarly JIT-Compilations happens only after a lot of runs => necessary to run code for many times to get meaningful performance evaluation
- Dynamic Optimization
  - Even after many runs, it might happen a method gets compiled or an already compiled method gets interpreted

Problems with Benchmarking 3

- On Stack Replacement
  - Sometimes can actually slow things down
- Dead Code Elimination
  - Difficult to tell when code will get eliminated
- Memory Deallocation
  - Not necessarily predictable
- Cache and other hardware effects

Problems with Benchmarking 4

- Fast Fourier Transformation on different data sizes



Benchmark Framework

- Deals with some of the mentioned problems by:
  - Garbage collecting completely before the benchmark
  - Running task until there is no more new JIT compilation
  - After reaching steady state, run task many times and take average, also record mean and standard deviation

Performance / Tuning Tips

- Command-Line-Parameters for JVM:
  - -server for slow start, high performance, -client for fast start, slower performance
  - -Xms initial heap size, -Xmx maximum heap size, -Xss stack size per thread
- Use newer libraries, especially if old ones are only there for backwards compatibility
  - e.g. use NIO-classes instead of old IO-classes
- Profiling and analyzing
  - jconsole, -Xprof

Different JVM Implementations

- Most prominent JVM is Sun Hotspot JVM
- Other Major JVM Implementations usually optimized for specific hardware:
  - Oracle JRockit for Oracle Hardware
  - HP-UX for Risc-HP-Architectures
  - IBM J9 for IBM Hardware
- JVM developed at Uni Freiburg: TakaTuka!
  - Lead developer: Faisal Aslam
  - For small wireless devices, very small JVM size (can run on devices with 4 KB RAM)

Sources

- http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html (JVM Specification)
- http://en.wikipedia.org/wiki/Java_Virtual_Machine (Wikipedia Article on the general concept)
- http://en.wikipedia.org/wiki/Java_performance (Some informations about java performance, always disputed ;))
- http://java.sun.com/javase/technologies/performance.jsp (Sun/Oracle center for informations about Java Performance)
- http://www.ibm.com/developerworks/java/library/j-benchmark1.html (IBM Paper about benchmarking issues with java and the mentionjed benchmarking framework)
- http://scribblethink.org/Computer/javaCbenchmark.html (numerical benchmarks c and java)
- http://www.oracle.com/technetwork/java/hotspotfaq-138619.html (FAQ about Hotspot JVM)
- http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/ex (JVM Tuning Tips by IBM)
- http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/ (IBM Paper about Bytecode)
- http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings (Listings of all bytecode instructions)
- http://sourceforge.net/apps/mediawiki/takatuka/index.php?title=Main_Page (TakaTuka Homepage)