Overview

Focus On:
- small scale performance experiments
- making estimates, testing optimizations
- choosing alternatives and more
- mainstream virtual machine environment
  HotSpot, OpenJDK, Intel, Linux, Windows

Focus Not On:
- distributed applications and networking
- exotic virtual machine environments

... but generalizations should apply.

Experimental Platforms

Presented measurements were collected on:
- multicore Intel Xeon or Intel Core processors
- OpenJDK 1.7.0 or OpenJDK 1.8.0
- Fedora Linux 20 or 21

... detailed specification available.

Disclaimer

The presented measurements are highly platform dependent.

Naive Start

Purpose

Imagine we have two functionally equivalent implementations of the same interface. We want to know which one is faster.

```java
public interface Workload { void method (int param); }

class WorkloadImplOne implements Workload { ... }
class WorkloadImplTwo implements Workload { ... }

good measure (Workload workload) {
    long start = java.lang.System.nanoTime ();
    for (int i = 0 ; i < CYCLES ; i++) {
        workload.method (i);
    }
    long finish = java.lang.System.nanoTime ();
    return ((finish - start) / CYCLES);
}
```
Implementation One

Average invocation time from 1000000 invocations

Execution time [ns]

Naive Start III

Total execution time

Execution time [ns]

Workload workload;

workload = new WorkloadImplOne ();
// Warming up ...;
System.out.println("Warm Up One " + measure (workload));
// Measuring ...
System.out.println("One " + measure (workload));

workload = new WorkloadImplTwo ();
// Warming up ...
System.out.println("Warm Up Two " + measure (workload));
// Measuring ...
System.out.println("Two " + measure (workload));

Naive Start V

Total execution time with warm up

Execution time [ns]

Invocations

5e+02 5e+03 5e+04 5e+05 5e+06
1 10 100 1000 10000 100000 1000000

Naive Start...
Naive Start VI

With enough warm up, we appear to execute an iteration in 0.1 ns.

Take Away

Single iteration in sub-nanosecond range is technical nonsense. If nothing else then this should alert us that our code is naive.

And Besides

The two implementations were the same!

```java
public class WorkloadImpl implements Workload {
    private int last = 0;
    public void method (int param) {
        last = param;
    }
}
```

Tutorial Outline

1 Part 1: Initial vs Sustainable Performance
   - Class Loading
   - Just-In-Time Compilation
2 Part 2: More Just-In-Time Compilation
   - Optimizing Experiment Workload
   - Polymorphic Invocation
   - Optimistic Optimization
   - On Stack Replacement
3 Part 3: Managed Memory
4 Part 4: Parallelism
5 Part 5: Sensors
6 Summary

Outline

1 Part 1: Initial vs Sustainable Performance
2 Part 2: More Just-In-Time Compilation
3 Part 3: Managed Memory
4 Part 4: Parallelism
5 Part 5: Sensors
6 Summary

Initial vs Sustainable Performance

Initial Performance
Performance influenced by one-time artifacts:
- code loaded and profiled and compiled on demand
- heap generations sized and old objects promoted
- various caches primed with content

Sustainable Performance
Performance can exhibit repeatable fluctuations:
- garbage collection cycles
- process scheduling
Assumed indefinitely sustainable.

... both situations can be of interest.
... distinguishing them is important.
Sustainable Performance

Guess Which Is True?

- long execution will exhibit sustainable performance
- low variance indicates sustainable performance
- sustainable performance is repeatable

One time artifacts can appear in many situations:
- performance may change after long period of stability
- large change may be due to minor workload fluctuation
- stable performance does not necessarily repeat after restart

Understanding mechanisms behind one-time artifacts is essential. Some mechanisms require active steps to reach sustainable performance.

Topics

1. Part 1: Initial vs Sustainable Performance
   Class Loading

   Just-In-Time Compilation

   Individual classes are loaded on demand:
   - dynamic construction
   - static class use
   - not sooner

   Classes are rarely unloaded:
   - garbage collection also works on class objects
   - but many references keep classes alive:
     - reference from class loader to class
     - reference from class to class loader

Class Loading II

```
java -verbose:java -jar dacapo-9.12-bach.jar eclipse
[Opened /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Object from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.reflect.AnnotatedElement from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.reflect.GenericDeclaration from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.reflect.Type from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Class from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Cloneable from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.ClassLoader from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.System from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Throwable from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
[Loaded java.lang.Error from /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.25.x86_64/jre/lib/rt.jar]
...
```
Class Loading III

Class loading impacts performance in multiple ways:

- class loading is (obviously) synchronous with program execution
- single class loader execution can be serialized
- presence or absence of classes is considered in optimizations
- class loading can also invalidate optimized code

DOs and DON'Ts: Class Loading

Make sure all relevant classes are used before measurement:

- classes that will be used
- classes that would be loaded in reality

Consult class loading log to detect anomalies.

Topics

1. Part 1: Initial vs Sustainable Performance
   - Class Loading
   - Just-In-Time Compilation

Program Compilation

From Source Code to Bytecode:
- relatively less exciting part

From Bytecode to Native Code:
- can use multiple compilers in multiple configurations
  - client compiler, server compiler, tiered compilation
- impacts performance in multiple ways:
  - initial compilation overhead
  - varying optimization levels
  - deoptimization
JIT: Tiered Compilation

Balancing compilation time against execution time.

**Early Execution**

Quicker compilation but little optimization.
Profiling during execution:
- various levels of profiling
- minimum are invocation counts and backedge counts
- but also branch statistics and type statistics
Profiling entails overhead.

**Late Execution**

Slower compilation but more optimization.
Profiling replaced with guards on assumptions.

JIT: Tiered Compilation II

There are multiple configurable policies that trigger compilation.
Five compilation levels:
- 0 interpreting
- 1 C1 compiler
- 2 C1 compiler with less profiling (MCS)
- 3 C1 compiler with more profiling (MDO)
- 4 C2 compiler

Transitions between levels are based on invocation count and backedge count and queue lengths:
- typical transition 0 - 3 - 4
- when C2 busy 0 - 2 - 3 - 4
- fall back to 1 on trivial methods
- fall back to 0 on deoptimization

JIT: Tiered Compilation III

```
java -XX:+PrintCompilation -jar dacapo-9.12-back.jar eclipse
002 3 3 java.lang.String::charAt (29 bytes)
011 3 3 java.lang.String::equals (81 bytes)
018 3 3 java.lang.String::hashCode (55 bytes)
046 3 0 [pro.Lang.System::arraycopy (native) (static)]
050 1 3 java.lang.String::lastIndexOf (52 bytes)
057 43 3 java.lang.String::lastIndexOf (52 bytes) made notentrant
059 66 3 sun.nio.cs.UTF_8$Encoder::encode (359 bytes) made notentrant
061 172 4 java.lang.String::equals (81 bytes)
069 2 3 java.lang.String::equals (81 bytes) made notentrant
```


DOs and DON'Ts: Tiered Compilation

Make sure all relevant classes are compiled before measurement.

Consult compilation log to detect anomalies:
- -XX:+PrintCompilation
- -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation
JIT: Method Inlining

Compilation acts on methods.
Method inlining extends compilation scope:
• more code gets compiled and therefore optimized together
• same code can be compiled differently in different contexts
Balancing optimization benefits against compilation time:
• inlining is limited by method size and method depth
tens of bytecodes for cold methods
maximum inlining depth usually below ten
• inlining reflects known or expected importance
hundreds of bytecodes for hot methods
code that often throws exceptions
some boxing and constructor code
• complete rules quite complex
Some obvious limitations (known targets, loaded types).

JIT: Method Inlining II

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
   -XX:+PrintCompilation -jar dacapo-9.12-bach.jar eclipse

java.util.concurrent.ConcurrentHashMap::putVal (362 bytes)
   @ 17 java.lang.Object::hashCode (0 bytes) no static binding
   @ 20 java.util.concurrent.ConcurrentHashMap::spread (10 bytes)
   @ 69 java.util.concurrent.ConcurrentHashMap::tabAt (21 bytes)
   @ 14 sun.misc.Unsafe::getObjectVolatile (0 bytes) intrinsic
   @ 92 java.util.concurrent.ConcurrentHashMap$Node::<init> (26 bytes)
   @ 1 java.lang.Object::<init> (1 bytes)
   @ 95 java.util.concurrent.ConcurrentHashMap::casTabAt (20 bytes)
   @ 16 sun.misc.Unsafe::compareAndSwapObject (0 bytes) intrinsic
   @ 121 java.util.concurrent.ConcurrentHashMap::helpTransfer ←
   @ 14 sun.misc.Unsafe::getObjectVolatile (0 bytes) intrinsic
```

DOs and DON'Ts: Method Inlining

Limit instrumentation of short methods:
• adding instrumentation can disrupt inlining
  inlined instrumentation can have smaller overhead
Consider interaction between inlining and measurement harness:
• consult inlining log to detect anomalies
• control inlining where needed:
  • using virtual machine options
  • using intelligent measurement harness


JIT: Determinism

Many compilation inputs change in time:
• execution profiles:
  • invocation counts
  • branch profiles
  • type profiles
• loaded class hierarchy
• compiler queue length
•...
Compilation output therefore depends on compilation timing.
Experiment: Changing Compilation Threshold

Purpose
See how having JIT occur at different times impacts sustainable performance.

```
#!/bin/bash
BENCHMARKS="$(java -jar scala-benchmark-suite.jar -l)"
for THRESHOLD in 1 2 3 5 10 20 30 50 100 200 300 ...
do
  for BENCHMARK in ${BENCHMARKS:?}
  do
    java -XX:CompileThreshold=${THRESHOLD:?} \n      -jar scala-benchmark-suite.jar \n      -n 50 ${BENCHMARK:?}
  done
done
```

Experiment: Changing Compilation Threshold II

![Graphs showing compilation threshold and benchmark scores](image)

Experiment: Changing Compilation Threshold III

Take Away:
- no common trend among benchmarks
- earlier compilation not always better
- occasional anomalous results possible
- magnitude significant

Replay Compilation

It is in principle possible to record execution profile and use it to direct compilation in multiple runs. This would make repeated compilation deterministic:

Challenges and drawbacks:
- requires virtual machine support
- associating profile with code is tricky
- deterministic is not necessarily optimal
- makes sense only for sustainable performance

Support in production virtual machines limited (Azul Zing ReadyNow, HotSpot Inlining Replay).

DOs and DON’Ts: Determinism

Multiple repetitions with restarts are essential.
Platform and configuration heterogeneity is also useful.

Significant open questions:
- how to perform sensitivity analysis?
- how to introduce reasonable disruptions?


Initial Performance

Sometimes initial performance is of interest.
Separating contributing factors can be a problem:
- wait for class loading and compilation to be triggered
- time spent loading classes
- time spent in compilation
- ...

Experiment: Disk Access and Initial Performance

Purpose
See how disk access time contributes to initial performance.

#!/bin/bash
# Flush dirty cache content
sync
# Drop clean cache content (must be root)
echo 3 > /proc/sys/vm/drop_caches

Options to investigate:
- once before experiment
- once before each run with code in JAR
- once before each run with code in directories

Experiment: Disk Access and Initial Performance II
Considering JIT Compilation

Compilation sensitive to minute details:
- relative timing between workload and compilation
- detailed execution profile of the workload
- low level code properties such as size
- ...

We need experiments relevant to reality:
- trying to control all relevant details not practical
- ambition is avoiding systematic difference from reality

Optimizing Workload

We want roughly the same optimizations applied in experiment and in reality. But this is not necessarily the case.

Recall the naive example.

```java
public class WorkloadImpl implements Workload {
  private int last = 0;
  public void method (int param) {
    last = param;
  }
}
long start = System.nanoTime ();
for (int i = 0 ; i < CYCLES ; i ++) {
  workload.method (i);
}
long finish = System.nanoTime ();
System.out.println ("Setter takes " + (finish-start)/CYCLES + "ns");
```
Optimizing Workload II

Sometimes Optimized Too Much:

• experiment can discard output used in reality
  and compiler may remove code producing such output

• experiment can use more limited input than reality
  and compiler may optimize for specific input

• experiment can repeat workload done once in reality
  and compiler may move loop invariant code

• ...

Sometimes Optimized Too Little:

• measures to prevent the opposite may work too well

[23] Shipilev: Java Microbenchmark Harness (The Lesser of Two Evils) (2013)

Experiment: Guessing Optimization Impact

Purpose

Try to estimate what the optimization actually does.

```
start = System.nanoTime ();
for (int i = 0 ; i < CYCLES ; i ++) {
    HashMap<Integer,Integer> m = new HashMap<Integer,Integer> ();
}
finish = System.nanoTime ();
```

```
start = System.nanoTime ();
for (int i = 0 ; i < CYCLES ; i ++) {
    Integer x = m.get (i);
}
finish = System.nanoTime ();
```

Experiment: Guessing Optimization Impact II

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    ...
    final float loadFactor;
    ...

    start = System.nanoTime ();
    for (int i = 0 ; i < CYCLES ; i ++) {
        memoryBarrier ();
    }
    finish = System.nanoTime ();
```

Experiment: Guessing Optimization Impact III

```
start = System.nanoTime ();
for (int i = 0 ; i < CYCLES ; i ++) {
    hashmap (i);
}
finish = System.nanoTime ();
```

The code is an obvious candidate for getting dropped.
Experiment: Guessing Optimization Impact IV

```java
HashMap<Integer,Integer> m = new HashMap<Integer,Integer> ();
start = System.nanoTime ();
for (int i = 0 ; i < CYCLES ; i ++) {
  Integer x = m.get (i);
}
finish = System.nanoTime ();

Again the code is an obvious candidate for getting dropped.

public final class Integer extends Number
implements Comparable<Integer> {

  public static Integer valueOf (int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
      return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer (i);
  }

  ...
```

DOs and DON'Ts: Optimizing Experimental Workload

Use experiment outputs:
- accumulating into visible variable usually works:
  - but requires careful memory model consideration
  - and may happen to work only by coincidence
- framework solutions such as JMH black holes are safer

Randomize experiment inputs:
- having variable inputs can prevent some optimizations
- but careful randomization has additional benefits

Limit optimization scope:
- perhaps disable inlining of measured code
- but be aware of possible consequences

[24] Shipilev: Java vs Scala: Divided We Fail (2014)

Topics

Part 2: More Just-In-Time Compilation
  Optimizing Experiment Workload
  Polymorphic Invocation
  Optimistic Optimization
  On Stack Replacement

Polymorphic Invocation

Calls (and some other constructs) benefit from knowing target type.

Compiler can guess target type:
- through class hierarchy analysis
- through profiling

Some optimizations possible when type known:
- statically determining call target
- caching frequent call targets
- discarding type checks
Experiment: Polymorphic Invocation Overhead

**Purpose**

Measure the cost of polymorphic invocation depending on the number of different targets being actually called.

```java
public interface Target {
    public int work();
}

// Many classes with the same code.
public static class Target00 implements Target {
    public int work() { return (000); }
}

boolean ex = false;
for (int i = 0; i < CYCLES; i++) {
    Target target = targets[random.nextInt(targets.length)];
    // Do something with the result to prevent optimization.
    ex |= target.work() < 0;
}
if (ex) dummy ++;
```

---

Experiment: Polymorphic Invocation Overhead II

---

Experiment: Changing Polymorphic Target

**Purpose**

Measure how the cost of polymorphic invocation behaves with single slowly changing target.

```java
public interface Target {
    public int work();
}

// Many classes with the same code.
public static class Target00 implements Target {
    public int work() { return (000); }
}

boolean ex = false;
Target target = targets[random.nextInt(targets.length)];
for (int i = 0; i < CYCLES; i++) {
    // Do something with the result to prevent optimization.
    ex |= target.work() < 0;
}
if (ex) dummy ++;
```

---

Experiment: Changing Polymorphic Target II

---
Optimistic Optimization

It is difficult to thoroughly appreciate the difference between static and dynamic compilation.

<table>
<thead>
<tr>
<th>Static Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If we can prove an optimization is correct, we can apply it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If we believe an optimization is beneficial, we can apply it.</td>
</tr>
<tr>
<td>If we cannot prove it is correct, we can apply it with a guard.</td>
</tr>
</tbody>
</table>

To represent reality, experiment must trigger guards that signal wrong assumptions.

Experiment: Exploring Rare Execution Path

Purpose
Examine performance of transient execution path changes.

```java
boolean ex = false;
for (int i = 0; i < CYCLES; i++) {
    try {
        // Access a random position within the array.
        ex |= array[random.nextInt(MAX - MIN) + MIN] < 0;
    } catch (ArrayIndexOutOfBoundsException e) {
        ex |= array[0] < 0;
    }
    if (ex) dummy ++;
}
```

Options to investigate:
- start execution with MIN = 0 and MAX = array.length
- then MIN = -array.length and MAX = 2 * array.length
- then MIN = 0 and MAX = array.length

Experiment: Exploring Rare Execution Path II

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Single Access Duration [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 DUO</td>
<td>27.7 27.8 27.9 28.0 28.1 28.2 28.3</td>
</tr>
<tr>
<td>Xeon E5</td>
<td>18.765 18.775 18.785 18.795</td>
</tr>
</tbody>
</table>

Before exception

After exception
Part 2: More Just-In-Time Compilation

Optimizing Experiment Workload
Polymorphic Invocation
Optimistic Optimization
On Stack Replacement

On Stack Replacement

Compilation happens while program executes.

When is compiled code used?
• next method invocation
• next loop iteration

Need for OSR is typical for benchmark workload loops.

Compilation with OSR differs from full compilation:
• different entry point
• profile typically limited

Outline

1 Part 1: Initial vs Sustainable Performance
2 Part 2: More Just-In-Time Compilation
3 Part 3: Managed Memory
4 Part 4: Parallelism
5 Part 5: Sensors
6 Summary

Managed Memory

Can we characterize memory consumption of particular function?

Heap Consumption:
• objects returned as results
• still live objects allocated during invocation
• no longer live objects allocated during invocation

Stack Consumption:
• local variables
• non-escaping objects

All consumption types have potential performance impact.
Generational Garbage Collection

Rough principles:
• objects kept in separate generations depending on age
• allocations consume young generation space
• collections gradually tenure live objects
• collections either young or full
• triggered on low memory

Many parameters vary:
• actual collection algorithms
• conditions for triggering collections
• sizing (initial and runtime) of generations


Experiment: Performance Impact of Heap Size

Purpose
See how heap size relates to performance.

for size in 29 30 31 ...; do
  java -Xms$size -Xmx$size -jar dacapo.jar ...
done

• default benchmark settings with 30 iterations
• GC forced between iterations (not measured)

Experiment: Performance Impact of Heap Size II

<table>
<thead>
<tr>
<th>Heap size [MB]</th>
<th>Iteration execution time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>400, 600, 800, 1000, 1200</td>
</tr>
<tr>
<td>34</td>
<td>fop</td>
</tr>
<tr>
<td>39</td>
<td>2, 6, 10, 15, 20, 25, 30</td>
</tr>
<tr>
<td>44</td>
<td>3500, 4500, 5500, 6500</td>
</tr>
<tr>
<td>49</td>
<td>avrora</td>
</tr>
<tr>
<td>54</td>
<td>DaCapo 9.12, default size, 30 iterations</td>
</tr>
<tr>
<td>59</td>
<td>Heap size</td>
</tr>
</tbody>
</table>

Experiment: Performance Impact of Heap Size III

Take Away
Heap size is important especially when not very large.
Heap configuration part of experiment setup:
• initial size
• ergonomics
Examine garbage collection log.
Avoiding Garbage Collection

Garbage collection tuning introduces too many additional variables. Perhaps we do not want to complicate our experiment.

Common steps to avoid GC:
- force GC between iterations
- disable heap size changes during measurement
- use large enough heap to avoid GC during measurement

Experiment: Forcing Collections Between Iterations

**Purpose**
Measure the changes in GC overhead when forcing GC between workload iterations.

**Before iteration:**
- force garbage collection synchronously—at the start of each iteration

**During iteration:**
- force garbage collection asynchronously—roughly in the middle of each iteration

Collection time included in benchmark time in both cases.

Experiment: Forcing Collections Between Iterations II

DaCapo 9.12, fop, last 10 iterations (of 30)

<table>
<thead>
<tr>
<th>Heap size [MB]</th>
<th>Iteration execution time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>400</td>
</tr>
<tr>
<td>34</td>
<td>400</td>
</tr>
<tr>
<td>38</td>
<td>400</td>
</tr>
<tr>
<td>42</td>
<td>400</td>
</tr>
<tr>
<td>46</td>
<td>400</td>
</tr>
<tr>
<td>50</td>
<td>400</td>
</tr>
<tr>
<td>54</td>
<td>400</td>
</tr>
<tr>
<td>58</td>
<td>400</td>
</tr>
</tbody>
</table>

GC before
GC during

Experiment: Forcing Collections Between Iterations III

DaCapo 9.12, avrora, last 10 iterations (of 30)

<table>
<thead>
<tr>
<th>Heap size [MB]</th>
<th>Iteration execution time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4000</td>
</tr>
<tr>
<td>8</td>
<td>6000</td>
</tr>
<tr>
<td>12</td>
<td>8000</td>
</tr>
<tr>
<td>18</td>
<td>4000</td>
</tr>
<tr>
<td>24</td>
<td>6000</td>
</tr>
<tr>
<td>30</td>
<td>8000</td>
</tr>
<tr>
<td>36</td>
<td>4000</td>
</tr>
<tr>
<td>42</td>
<td>6000</td>
</tr>
<tr>
<td>48</td>
<td>8000</td>
</tr>
<tr>
<td>54</td>
<td>4000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
</tbody>
</table>

GC before
GC during
Experiment: Mixing in Additional Allocations

Purpose
Show how allocation timing impacts garbage collection overhead.

Sequential mix:
- add extra allocations at the end of each iteration

Parallel mix:
- add extra allocations in parallel with each iteration
Extra allocations about as fast as existing workload allocations.

Experiment: Mixing in Additional Allocations II

Take Away
Timing of garbage collections is important:
- complexity of some steps depends on live object count
- workloads with sawtooth live size pattern are sensitive

Garbage collection timing impacts related mechanisms:
- calls to finalizers
- soft and weak and phantom references

Stack Allocation
Object allocation in source code does not necessarily translate to object allocation on heap at runtime.

Escape Analysis
Compiler can recognize whether object reference escapes method scope.
Non-escaping objects:
- can be allocated on stack like local variables
- do not have to adhere to standard object layout
Common use with scalar replacement.

Adding escapes (instrumentation) can break stack allocation.

Storing Samples

Measurements require storing samples:
- heap storage can be disruptive
- other forms of storage available
  - C heap through JNI
  - C heap through sun.misc.unsafe


Storing Samples II

Overhead of storing measurement samples

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Overhead (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>0</td>
</tr>
<tr>
<td>LinkedList</td>
<td>20</td>
</tr>
<tr>
<td>HashMap</td>
<td>40</td>
</tr>
<tr>
<td>HashBag</td>
<td>60</td>
</tr>
</tbody>
</table>

Storing Samples III

Storage in C heap using sun.misc.unsafe:

```java
Field unsafeField = Unsafe.class.getDeclaredField("theUnsafe");
unsafeField.setAccessible(true);
Unsafe unsafe = (Unsafe) unsafeField.get(null);
long dataPtr = unsafe.allocateMemory(VALUES * LONG_SIZE);
unsafe.putLong(dataPtr + index * LONG_SIZE, someValue);
long val = unsafe.getLong(dataPtr + index * LONG_SIZE);
```

Outline

1. Part 1: Initial vs Sustainable Performance
2. Part 2: More Just-In-Time Compilation
3. Part 3: Managed Memory
4. Part 4: Parallelism
5. Part 5: Sensors
6. Summary
Parallel Workload

More issues come up when using multiple threads:
- synchronizing measurements
- workload optimizations
- resource sharing
- ...

Experiment: Thread Sleep Timing

Purpose
Examine how thread sleep behaves.

```java
for (int time = 0; time < 1000000 ; time += 100000) {
    int nanos = time % (1000 * 1000);
    int millis = time / (1000 * 1000);
    long start = System.nanoTime ();
    Thread.sleep (millis, nanos);
    long finish = System.nanoTime ();
}
```

Experiment: Thread Sleep Timing II

Synchronization Optimizations

Synchronization heavily optimized for no contention case. Measurements require realistic contention.

Some possible optimizations:
- biased locking
- adaptive spinning
- lock coarsening
- lock elision
- ...
Parallel Resource Sharing
Many computational resources are shared during parallel execution. Many visible effects are related to sharing of memory hardware.

Memory Hardware
Relies on caching to improve performance. Caching known to introduce performance anomalies for some access patterns:
- non balanced set selection
- false cache line sharing
- ...

Memory Sharing
At source code level hardware memory issues may appear distant. Patterns such as false sharing can still be encountered. Environment makes some anomalies difficult to counter.

Experiment: Effect of Regular Allocation on Cache Set Use
Purpose
See whether regular workload at language level can translate into problematic memory access patterns.

```java
// Hash table initialization.
HashMap<String, String> data = new HashMap<String, String>(16384);
String keys[] = new String[1024];
for (int i = 0; i < 1024; i++)
    keys[i] = randomString(32);
data.put(keys[i], randomString(SIZE));

boolean ex = false;
for (int i = 0; i < steps; i++)
    String value = data.get(keys[random.nextInt(keys.length)]);
    // Do something with the result to prevent optimization.
    ex |= value.charAt(0) == 'X';
if (ex) dummy++;
```

Experiment: Effect of Regular Allocation on Cache Set Use II

![Access time vs. LLC misses graph](image)

Experiment: False Sharing on Card Table
Purpose
Investigate the possibility of false sharing occurring on internal memory management structures.

```java
static Object[] mem = null;
for (int distance = 256; distance < 128*1024; distance += 128)
    mem = new Object[1000000];
int BOUNDARY = getCardTableBoundaryIndex();
executeInNewThread(work(BOUNDARY-3072 + distance, 1024));
executeInNewThread(work(BOUNDARY-4096, 1024));
int BOUNDARY = getCardTableBoundaryIndex();
executeInNewThread(work(BOUNDARY-3072 + distance, 1024));
executeInNewThread(work(BOUNDARY-4096, 1024));

void work(int from, int items)
    for (int i = 0; i < items; i++)
        mem[from + i] = someObjectOnHeap;
```
Generational collection needs to track references between generations.

### Card Table

Data structure used to track references:
- each 1B card records reference updates in 512B of heap
- card address a linear function of update address

### Write Barrier

Write barrier takes care of updating card table on reference write:
- one cache line represents 64B $\times$ 512 = 32KB of heap
- possible false sharing on card table cache line

#### Experiment: False Sharing on Card Table II

<table>
<thead>
<tr>
<th>Access time</th>
<th>LLC misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance [array indices]</td>
<td>Last level cache misses</td>
</tr>
</tbody>
</table>

#### Memory Sharing

- false cache line sharing
- remote memory allocation

Some anomalies can be avoided by design:
- careful placement of shared variables
- careful design of memory structures

Others we can only hope to average out with randomized allocation patterns.

Many anomalies can be identified using processor performance event counters.

#### Outline

1. Part 1: Initial vs Sustainable Performance
2. Part 2: More Just-In-Time Compilation
3. Part 3: Managed Memory
4. Part 4: Parallelism
5. Part 5: Sensors
6. Summary
Timing Sources

java.lang.System.currentTimeMillis:
• returns value of system time since January 1, 1970 UTC
• units are ms, resolution undefined

java.lang.System.nanoTime:
• returns value of “high resolution time source”
• units are ns, resolution undefined
• arbitrary start value, can overflow

Timing Sources II

java.lang.management.ThreadMXBean:
• methods to return thread execution time
• resolution undefined, but usually low
• can throw an exception if not implemented


Nano Time Source Properties

The properties of java.lang.System.nanoTime are version and platform dependent.

Linux
• relies on the CLOCK_MONOTONIC time source
• calibrated against RTC clock on boot
• subject to system time adjustments when using NTP or PTP daemons

Windows
• relies on the QueryPerformanceCounter system call
• usually high frequency but somewhat lower granularity

Nano Time Source Properties II

Miscellanea
• some systems may fall back on low resolution sources
• some systems use synchronization inside queries

Nano Time Boot Calibration

Frequency detection on Intel Core 2 DUO processor

Nano Time Network Adjustment

Initial iterations

Long-term changes

Performance Event Counters

Counters can provide useful additional information:
- access possible through JNI and native code
- inherent overhead of sample collection

Multiple counter usage modes:
- profiling
- sampling

Performance Event Counters II

import cz.cuni.mff.d3s.perf.*;

String [] probes = { "SYS_WALT_CLOCK", "ix86arch::INSTRUCTION_RETIRED", "ix86arch::MISPREDICTED_BRANCH_RETIRED" };

Benchmark.init (SAMPLES, probes);

for (int i = 0 ; i < SAMPLES ; i ++) {
    Benchmark.start ();
    ...
    Benchmark.stop ();
}

BenchmarkResultsPrinter.table (Benchmark.getResults (), System.out);

Outline

1 Part 1: Initial vs Sustainable Performance
2 Part 2: More Just-In-Time Compilation
3 Part 3: Managed Memory
4 Part 4: Parallelism
5 Part 5: Sensors
6 Summary

How To Conclude?

There Are Always More Surprises!
It is important to appreciate the complexity involved.

What are the appropriate safeguards?
- proper experiment design
- avoid regularity
- include cross checks
- keep open mind about results
- publish complete experiments

Thank You For Your Attention

http://d3s.mff.cuni.cz

This work was partially supported by:

References

References VI


References VII
