Designing for Performance

Rex Kerr
HHMI Janelia Farm Research Campus

Scala Days 2013
Designing for Performance

- Some conventional wisdom, and when to be unconventional
- Getting the big picture from profiling and why you can’t count on getting the small picture
- Getting the small picture from microbenchmarking and why you can’t count on getting the big picture
- Timings: the small picture from which performance is built
- Design guidelines for high-performance code
Designing for Performance

- Some conventional wisdom, and when to be unconventional
- Getting the big picture from profiling and why you can’t count on getting the small picture
- Getting the small picture from microbenchmarking and why you can’t count on getting the big picture
- Timings: the small picture from which performance is built
- Design guidelines for high-performance code

This space intentionally left blank for people in the back
Designing for Performance

- Some conventional wisdom, and when to be unconventional
- Getting the big picture from profiling and why you can’t count on getting the small picture
- Getting the small picture from microbenchmarking and why you can’t count on getting the big picture
- Timings: the small picture from which performance is built
- Design guidelines for high-performance code

This space intentionally left blank for people in the back
Designing for Performance

- Some conventional wisdom, and when to be unconventional
- Getting the big picture from profiling and why you can’t count on getting the small picture
- Getting the small picture from microbenchmarking and why you can’t count on getting the big picture
- Timings: the small picture from which performance is built
- Design guidelines for high-performance code
Designing for Performance

- Some conventional wisdom, and when to be unconventional
- Getting the big picture from profiling and why you can’t count on getting the small picture
- Getting the small picture from microbenchmarking and why you can’t count on getting the big picture
- Timings: the small picture from which performance is built
- Design guidelines for high-performance code
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
Wisdom #1

“Premature optimization is the root of all evil”
–Donald Knuth

What is premature and what is mature?

“Premature optimization for speed is the root of all evil in Formula One racing” (?!)

“In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.”
–Donald Knuth

Optimize wisely: know when and how; don’t waste your time.
Wisdom #1

“Premature optimization is the root of all evil”
– Donald Knuth

- What is premature and what is mature?

“Premature optimization for speed is the root of all evil in Formula One racing” (?!)

“In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.”
– Donald Knuth

Optimize wisely: know when and how; don’t waste your time.
Wisdom #1

“Premature optimization is the root of all evil”

– Donald Knuth

● What is premature and what is mature?

“Premature optimization for speed is the root of all evil in Formula One racing” (?!)

“In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.”

– Donald Knuth

Optimize wisely: know when and how; don’t waste your time.
Wisdom #1

“Premature optimization is the root of all evil”

–Donald Knuth

What is premature and what is mature?

“Premature optimization for speed is the root of all evil in Formula One racing” (?)

“In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.”

–Donald Knuth

Optimize wisely: know when and how; don’t waste your time.
Wisdom #2

“design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized”

–Wikipedia article on Program Optimization

For this to be good advice, it assumes

- Profiling will show you which parts are slow
- Code is modular: for any slow $X$, you can rewrite it as $X_{fast}$

But neither of these is consistently true.

“Design your Formula One race car first, and then test the resulting vehicle to see which parts can be modified to make the car faster.” (?!?!)

Anticipate performance problems and design to admit optimization, or build the performance-critical core first.
Wisdom #2

“design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized”

–Wikipedia article on Program Optimization

For this to be good advice, it assumes

- Profiling will show you which parts are slow
- Code is modular: for any slow X, you can rewrite it as $X_{fast}$

But neither of these is consistently true.

“Design your Formula One race car first, and then test the resulting vehicle to see which parts can be modified to make the car faster.” (?!?!)

Anticipate performance problems and design to admit optimization, or build the performance-critical core first.
Wisdom #2

“design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized”

–Wikipedia article on Program Optimization

For this to be good advice, it assumes

- Profiling will show you which parts are slow
- Code is modular: for any slow $X$, you can rewrite it as $X_{fast}$

But neither of these is consistently true.

“Design your Formula One race car first, and then test the resulting vehicle to see which parts can be modified to make the car faster.” (?!?!)

Anticipate performance problems and design to admit optimization, or build the performance-critical core first.
Wisdom #2

“design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized”

–Wikipedia article on Program Optimization

For this to be good advice, it assumes

- Profiling will show you which parts are slow
- Code is modular: for any slow $X$, you can rewrite it as $X_{fast}$

But neither of these is consistently true.

“Design your Formula One race car first, and then test the resulting vehicle to see which parts can be modified to make the car faster.” (?!?!)

Anticipate performance problems and design to admit optimization, or build the performance-critical core first.
Wisdom #2

“design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized”

–Wikipedia article on Program Optimization

For this to be good advice, it assumes

- Profiling will show you which parts are slow
- Code is modular: for any slow $X$, you can rewrite it as $X_{fast}$

But neither of these is consistently true.

“Design your Formula One race car first, and then test the resulting vehicle to see which parts can be modified to make the car faster.” (?!?!)
Wisdom #3

“The bottleneck isn’t where you think it is.”

“Even experienced programmers are very poor at predicting (guessing) where a computation will bog down.”

–Various people on various blogs etc.

- Predicting performance problems is a skill.
- Like most skills it can be learned.
- You can’t learn from nothing. You need data.
Wisdom #3

“The bottleneck isn’t where you think it is.”

“Even experienced programmers are very poor at predicting (guessing) where a computation will bog down.”

—Various people on various blogs etc.

- Predicting performance problems is a skill.
- Like most skills it can be learned.
- You can’t learn from nothing. You need data.
Wisdom #3

“The bottleneck isn’t where you think it is.”

“Even experienced programmers are very poor at predicting (guessing) where a computation will bog down.”

–Various people on various blogs etc.

- Predicting performance problems is a skill.
- Like most skills it can be learned.
- You can’t learn from nothing. You need data.
Wisdom #3

“The bottleneck isn’t where you think it is.”

“Even experienced programmers are very poor at predicting (guessing) where a computation will bog down.”

—Various people on various blogs etc.

- Predicting performance problems is a skill.
- Like most skills it can be learned.
- You can’t learn from nothing. You need data.
Wisdom #3

“The bottleneck isn’t where you think it is.”

“Even experienced programmers are very poor at predicting (guessing) where a computation will bog down.”

–Various people on various blogs etc.

- Predicting performance problems is a skill.
- Like most skills it can be learned.
- You can’t learn from nothing. You need data.
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
What is the bottleneck in this code?

object ProfEx1 {
    val dict = Seq(
        "salmon", "cod", "grouper", "bass", "herring",
        "eel", "trout", "perch", "halibut", "dorado"
    )
    def permuted = dict.permutations.map(_.mkString).to[Vector]
    def scanAll(sought: Seq[String]) = {
        def scan(s: String) = sought.exists(s contains _)
        permuted.filter(scan)
    }
    def report(sought: Seq[String], scanned: Seq[String]) = sought map { word =>
        scanned find(_ contains word) match {
            case Some(s) => s"found $word in $s"
            case None => s"could not find $word"
        }
    }
    def printOut(lines: Seq[String]) = lines.foreach(println)
    def main(args: Array[String]) {
        val answer = report(args, scanAll(args))
        printOut(answer)
    }
}
Wait, maybe it’s not even too slow.

```
$ time scala -J-Xmx1G ProfEx1 snakes say sss
could not find snakes
could not find say
found sss in codgrouperbasssalmonherringeeltoutperchhalibutdorado

real 0m5.861s
user 0m10.790s
sys 0m1.080s
```

Okay, that’s slow. We need a profiler?
The basics: what is a profiler?

- Provide information about time spent in various parts of code; may track memory usage, class loads, and other parameters also
- Broken roughly into two categories: *instrumenting* and *sampling*
  - Instrumenting profilers rewrite the bytecode so that running a method will report information about it (e.g. number of times called, duration inside, etc.)
  - Sampling profilers take snapshots of the stacks for each thread to infer where the most time is spent
- Oracle JVM has one built in (-Xrunhprof) and one external (VisualVM)
- IDEs may include one (e.g. Eclipse, NetBeans)
- Commercial profilers may have superior features (e.g. YourKit)
The basics: what is a profiler?

- Provide information about time spent in various parts of code; may track memory usage, class loads, and other parameters also
- Broken roughly into two categories: *instrumenting* and *sampling*
- Instrumenting profilers rewrite the bytecode so that running a method will report information about it (e.g. number of times called, duration inside, etc.)
- Sampling profilers take snapshots of the stacks for each thread to infer where the most time is spent
- Oracle JVM has one built in (-Xrunhprof) and one external (VisualVM)
- IDEs may include one (e.g. Eclipse, NetBeans)
- Commercial profilers may have superior features (e.g. YourKit)
The basics: what is a profiler?

- Provide information about time spent in various parts of code; may track memory usage, class loads, and other parameters also
- Broken roughly into two categories: *instrumenting* and *sampling*
- Instrumenting profilers rewrite the bytecode so that running a method will report information about it (e.g. number of times called, duration inside, etc.)
- Sampling profilers take snapshots of the stacks for each thread to infer where the most time is spent
- Oracle JVM has one built in (*-Xrunhprof*) and one external (VisualVM)
- IDEs may include one (e.g. Eclipse, NetBeans)
- Commercial profilers may have superior features (e.g. YourKit)
Instrumentation everywhere is terrible

- Slow: extensive instrumentation greatly slows runtime

$ time scala -J-Xmx1G -J-Xrunhprof:cpu=times ProfEx1 snakes say sss
could not find snakes
could not find say
found sss in codgrouperbasssalmonherringeeltroutperchhalibutdorado
Dumping CPU usage by timing methods ... done.

real 137m36.535s
user 138m24.740s
sys 0m4.170s

- Rather inaccurate: JVM makes all sorts of different decisions about
  inlining, etc., with radically changed bytecode

Instrumentation profilers will not reliably tell you where your
bottlenecks are, and may not be deployable in the relevant context

This space intentionally left blank for people in the back
Instrumentation everywhere is **terrible**

- **Slow**: extensive instrumentation greatly slows runtime

```
$ time scala -J-Xmx1G -J-Xrunhprof:cpu=times ProfEx1 snakes say sss
could not find snakes
could not find say
found sss in codgrouperbasssalmonherringeeltoutperchhalibutdorado
Dumping CPU usage by timing methods ... done.

real 137m36.535s
user 138m24.740s
sys 0m4.170s
```

- Rather inaccurate: JVM makes all sorts of different decisions about inlining, etc., with radically changed bytecode

**Instrumenting profilers will not reliably tell you where your bottlenecks are, and may not be deployable in the relevant context**
Instrumentation everywhere is **terrible**

- **Slow**: extensive instrumentation greatly slows runtime

```sh
time scala -J-Xmx1G -J-Xrunhprof:cpu=times ProfEx1 snakes say sss
could not find snakes
could not find say
found sss in codgrouperbasssalmonherringeeltrotperchhalibutdorado
Dumping CPU usage by timing methods ... done.
```

real 137m36.535s
user 138m24.740s
sys 0m4.170s

- **Rather inaccurate**: JVM makes all sorts of different decisions about inlining, etc., with radically changed bytecode

Instrumenting profilers will not reliably tell you where your bottlenecks are, and may not be deployable in the relevant context

This space intentionally left blank for people in the back
Instrumentation everywhere is **terrible**

- **Slow**: extensive instrumentation greatly slows runtime

```bash
$ time scala -J-Xmx1G -J-Xrunhprof:cpu=times ProfEx1 snakes says sss
could not find snakes
could not find say
found sss in codgrouperbasssalmonherringeeltroutperchhalibutdorado
Dumping CPU usage by timing methods ... done.

real 137m36.535s
user 138m24.740s
sys 0m4.170s
```

- **Rather inaccurate**: JVM makes all sorts of different decisions about inlining, etc., with radically changed bytecode

**Instrumentation profilers will not reliably tell you where your bottlenecks are, and may not be deployable in the relevant context**
Sampling is worse than you think

- JVM will not sample just anywhere! It selects “safe” locations for you.

Evaluating the Accuracy of Java Profilers

Todd Mytkowicz  
Amer Diwan  
University of Colorado at Boulder  
{mytkowit,diwan}@colorado.edu

Matthias Hauswirth  
University of Lugano  
Matthias.Hauswirth@unisi.ch

Peter F. Sweeney  
IBM Research  
pfs@us.ibm.com

---

**Figure 1.** Disagreement in the hottest method for benchmark pmd across four popular Java profilers.

See also

Profiling our example

By method (hprof output; run took 6.5 s):

1 21.74% 21.74% 75 300555 scala.collection.mutable.StringBuilder.append
2 17.10% 38.84% 59 300582 java.lang.String.indexOf
3 10.14% 48.99% 35 300560 scala.collection.mutable.ArrayBuffer.foreach
4 4.06% 53.04% 14 300568 scala.collection.mutable.StringBuilder.append
5 3.19% 56.23% 11 300551 scala.collection.immutable.VectorPointer$class.gotoNextBlockStartWritable
6 3.19% 59.42% 11 300565 scala.collection.Iterator$$anon$11.next
7 2.61% 62.03% 9 300562 scala.collection.mutable.ArrayBuffer.$plus$plus$eq
8 2.61% 64.64% 9 300586 scala.collection.IndexedSeqOptimized$class.segmentLength
9 2.32% 66.96% 8 300564 scala.collection.TraversableOnce$class.mkString
10 2.03% 68.99% 7 300559 scala.collection.mutable.StringBuilder.append

By line (analysis of hprof output):

64% #6 def permuted = dict.permutations.map(_.mkString).to[Vector]
22% #8 permuted.filter(scan)
7% (all the rest put together)
7% ?? (startup, etc.)

Conclusion: making permuted strings is slow.
Profiling our example

By method (hprof output; run took 6.5 s):

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Time (%)</th>
<th>Profiling (%)</th>
<th>Count</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>scala.collection.mutable:StringBuilder.append</td>
<td>21.74%</td>
<td>21.74%</td>
<td>75</td>
<td>300555</td>
</tr>
<tr>
<td>2</td>
<td>java.lang.String.indexOf</td>
<td>17.10%</td>
<td>38.84%</td>
<td>59</td>
<td>300582</td>
</tr>
<tr>
<td>3</td>
<td>scala.collection.mutable:ArrayBuffer.foreach</td>
<td>10.14%</td>
<td>48.99%</td>
<td>35</td>
<td>300560</td>
</tr>
<tr>
<td>4</td>
<td>scala.collection.mutable:StringBuilder.append</td>
<td>4.06%</td>
<td>53.04%</td>
<td>14</td>
<td>300568</td>
</tr>
<tr>
<td>5</td>
<td>scala.collection.immutable:VectorPointer$class.gotoNextBlockStartWritable</td>
<td>3.19%</td>
<td>56.23%</td>
<td>11</td>
<td>300551</td>
</tr>
<tr>
<td>6</td>
<td>scala.collection.Iterator$anon$11.next</td>
<td>3.19%</td>
<td>59.42%</td>
<td>11</td>
<td>300565</td>
</tr>
<tr>
<td>7</td>
<td>scala.collection.mutable:ArrayBuffer.$plus$plus$eq</td>
<td>2.61%</td>
<td>62.03%</td>
<td>9</td>
<td>300562</td>
</tr>
<tr>
<td>8</td>
<td>scala.collection.IndexedSeqOptimized$class.segmentLength</td>
<td>2.61%</td>
<td>64.64%</td>
<td>9</td>
<td>300586</td>
</tr>
<tr>
<td>9</td>
<td>scala.collection.TraversableOnce$class.mkString</td>
<td>2.32%</td>
<td>66.96%</td>
<td>8</td>
<td>300564</td>
</tr>
<tr>
<td>10</td>
<td>scala.collection.mutable:StringBuilder.append</td>
<td>2.03%</td>
<td>68.99%</td>
<td>7</td>
<td>300559</td>
</tr>
</tbody>
</table>

By line (analysis of hprof output):

64%  #6  def permuted = dict.permutations.map(_.mkString).to[Vector]
22%  #8  permuted.filter(scan)
  7%  (all the rest put together)
  7%  ?? (startup, etc.)

Conclusion: making permuted strings is slow.
## Profiling our example

By method (hprof output; run took 6.5 s):

<table>
<thead>
<tr>
<th>Method Details</th>
<th>Time %</th>
<th>Call %</th>
<th>Count</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. scala.collection.mutable.StringBuilder.append</td>
<td>21.74%</td>
<td>21.74%</td>
<td>75</td>
<td>300555</td>
</tr>
<tr>
<td>2. java.lang.String.indexOf</td>
<td>17.10%</td>
<td>38.84%</td>
<td>59</td>
<td>300582</td>
</tr>
<tr>
<td>3. scala.collection.mutable:ArrayBuffer.foreach</td>
<td>10.14%</td>
<td>48.99%</td>
<td>35</td>
<td>300560</td>
</tr>
<tr>
<td>4. scala.collection.mutable:StringBuilder.append</td>
<td>4.06%</td>
<td>53.04%</td>
<td>14</td>
<td>300568</td>
</tr>
<tr>
<td>5. scala.collection.immutable:VectorPointer$class.gotoNextBlockStartWritable</td>
<td>3.19%</td>
<td>56.23%</td>
<td>11</td>
<td>300551</td>
</tr>
<tr>
<td>6. scala.collection.Iterator$$anon$11.next</td>
<td>3.19%</td>
<td>59.42%</td>
<td>11</td>
<td>300565</td>
</tr>
<tr>
<td>7. scala.collection.mutable:ArrayBuffer.$plus$plus$eq</td>
<td>2.61%</td>
<td>62.03%</td>
<td>9</td>
<td>300562</td>
</tr>
<tr>
<td>8. scala.collection.IndexedSeqOptimized$class.segmentLength</td>
<td>2.61%</td>
<td>64.64%</td>
<td>9</td>
<td>300586</td>
</tr>
<tr>
<td>9. scala.collection.TraversableOnce$class.mkString</td>
<td>2.32%</td>
<td>66.96%</td>
<td>8</td>
<td>300564</td>
</tr>
<tr>
<td>10. scala.collection.mutable:StringBuilder.append</td>
<td>2.03%</td>
<td>68.99%</td>
<td>7</td>
<td>300559</td>
</tr>
</tbody>
</table>

### By line (analysis of hprof output):

- **64% #6** def permuted = dict.permutations.map(_.mkString).to[Vector]
- **22% #8** permuted.filter(scan)
- **7%** (all the rest put together)
- **7%??** (startup, etc.)

### Conclusion: making permuted strings is slow.
Checking profiler accuracy with direct timing

object Ex1Time {
    val th = new ichi.bench.Thyme
    val dict = Seq(
        "salmon", "cod", "grouper", "bass", "herring",
        "eel", "trout", "perch", "halibut", "dorado"
    )
    def permuted = th.petime{ dict.permutations.map(_.mkString).to[Vector] }
    def scanAll(sought: Seq[String]) = {
        def scan(s: String) = sought.exists(s contains _)
        val p = permuted; th.petime{ p.filter(scan) }
    }
    def report(sought: Seq[String], scanned: Seq[String]) = th.petime{
        sought map { word =>
            scanned find(_ contains word) match {
                case Some(s) => s"found $word in $s"
                case None    => s"could not find $word"
            }
        }
    }
    def printOut(lines: Seq[String]) = th.petime{ lines.foreach(println) }
    def main(args: Array[String]) {
        val answer = report(args, scanAll(args))
        printOut(answer)
    }
}

Rex Kerr (JFRC)
Checking profiler accuracy, cont.

$ time scala -cp /home/kerrrr/code/scala/github/Thyme/Thyme.jar:.
  -J-Xmx1G Ex1Time snakes say sss

// permuted
Elapsed time: ~1.835 s (inaccurate)
  Garbage collection (36 sweeps) took: 2.628 s
  Total time: 4.463 s

// p.filter(scan)
Elapsed time: ~983. ms (inaccurate)
  Garbage collection (1 sweeps) took: 12. ms
  Total time: 995.0 ms

// Everything else < 100 ms

real 0m6.070s
user 0m12.270s
sys 0m0.790s

Close...I guess...75% / 16% vs. 64% / 22%
Profiling bottom line: use it, don’t trust it

Profiling is good for

- Long-running processes
- Finding unexpected blocks in multithreaded applications
- Getting a general sense of which methods are expensive

Profiling is not good for

- Identifying the hottest method
- Identifying anything inlined
- Quantitatively assessing modest speed improvements

If you need speed, design for speed. Use the profiler to catch surprises.
Profiling bottom line: use it, don’t trust it

Profiling is good for
- Long-running processes
- Finding unexpected blocks in multithreaded applications
- Getting a general sense of which methods are expensive

Profiling is not good for
- Identifying the hottest method
- Identifying anything inlined
- Quantitatively assessing modest speed improvements

If you need speed, design for speed. Use the profiler to catch surprises.
Profiling bottom line: use it, don’t trust it

Profiling is good for
- Long-running processes
- Finding unexpected blocks in multithreaded applications
- Getting a general sense of which methods are expensive

Profiling is not good for
- Identifying the hottest method
- Identifying anything inlined
- Quantitatively assessing modest speed improvements

If you need speed, design for speed. Use the profiler to catch surprises.
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
Microbenchmarking seems almost impossible

- JVM/JIT compiler decides
  - whether to compile your code (100x speed difference)
  - how much to inline
  - whether it can elide multiple dispatch, branching, bounds-checking, etc.

- Can’t measure anything fast due to poor timing utilities
- Context of a microbenchmark is surely different than production code
  - Different GC load / JIT decisions / pattern of use
- The gold-standard tools (Google Caliper (Java), ScalaMeter (Scala), Criterium (Clojure), etc.) take a nontrivial investment of time to use:
  - Not always easy to get working at all
  - Require non-negligible infrastructure to run anything as a benchmark
  - Do all sorts of things with class loaders and loading whole JVMs that take a while to complete (seconds–minutes)
Microbenchmarking seems almost impossible

- JVM/JIT compiler decides
  - whether to compile your code (100x speed difference)
  - how much to inline
  - whether it can elide multiple dispatch, branching, bounds-checking, etc.

- Can’t measure anything fast due to poor timing utilities

- Context of a microbenchmark is surely different than production code
  - Different GC load / JIT decisions / pattern of use

- The gold-standard tools (Google Caliper (Java), ScalaMeter (Scala), Criterium (Clojure), etc.) take a nontrivial investment of time to use:
  - Not always easy to get working at all
  - Require non-negligible infrastructure to run anything as a benchmark
  - Do all sorts of things with class loaders and loading whole JVMs that take a while to complete (seconds–minutes)
Microbenchmarking seems almost impossible

- JVM/JIT compiler decides
  - whether to compile your code (100x speed difference)
  - how much to inline
  - whether it can elide multiple dispatch, branching, bounds-checking, etc.

- Can’t measure anything fast due to poor timing utilities
- Context of a microbenchmark is surely different than production code
  - Different GC load / JIT decisions / pattern of use

- The gold-standard tools (Google Caliper (Java), ScalaMeter (Scala), Criterium (Clojure), etc.) take a nontrivial investment of time to use:
  - Not always easy to get working at all
  - Require non-negligible infrastructure to run anything as a benchmark
  - Do all sorts of things with class loaders and loading whole JVMs that take a while to complete (seconds–minutes)
Microbenchmarking seems almost impossible

- JVM/JIT compiler decides
  - whether to compile your code (100x speed difference)
  - how much to inline
  - whether it can elide multiple dispatch, branching, bounds-checking, etc.

- Can’t measure anything fast due to poor timing utilities
- Context of a microbenchmark is surely different than production code
  - Different GC load / JIT decisions / pattern of use
- The gold-standard tools (Google Caliper (Java), ScalaMeter (Scala), Criterium (Clojure), etc.) take a nontrivial investment of time to use:
  - Not always easy to get working at all
  - Require non-negligible infrastructure to run anything as a benchmark
  - Do all sorts of things with class loaders and loading whole JVMs that take a while to complete (seconds–minutes)
Microbenchmarking usually works anyway

Most of the time:

- The hottest code is JITted anyway
- The hottest code is called a lot, so it’s fair to batch calls in a loop
- If foo is faster than bar in some context, it is faster in most/all
- You can monitor or control for variability from GC, class loading, etc. by using JVM monitoring tools and robust statistics
Microbenchmarking usually works anyway

Most of the time:

- The hottest code is JITted anyway
- The hottest code is called a lot, so it’s fair to batch calls in a loop
- If `foo` is faster than `bar` in some context, it is faster in most/all
- You can monitor or control for variability from GC, class loading, etc. by using JVM monitoring tools and robust statistics
Microbenchmarking usually works anyway

Most of the time:

- The hottest code is JITted anyway
- The hottest code is called a lot, so it’s fair to batch calls in a loop
- If \texttt{foo} is faster than \texttt{bar} in some context, it is faster in most/all
- You can monitor or control for variability from GC, class loading, etc. by using JVM monitoring tools and robust statistics
Microbenchmarking usually works anyway

Most of the time:

- The hottest code is JITted anyway
- The hottest code is called a lot, so it’s fair to batch calls in a loop
- If `foo` is faster than `bar` in some context, it is faster in most/all
- You can monitor or control for variability from GC, class loading, etc. by using JVM monitoring tools and robust statistics
Avoid the common pitfalls of microbenchmarking

- Read “So You Want to Write a Micro-Benchmark” by John Rose and the linked paper by Brian Goetz:
  https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks
- Be aware of the top reasons why apparently “correct” microbenchmarks fail, including:
  - Real code requires multiple dispatch, test is single
  - Real code runs with heavily impacted GC, test is not
  - Real code uses results of computation, test does not
  - Real code isn’t even CPU bound, test is (ask a profiler!)
- Use a benchmarking tool to get the details right. If you don’t like the others, try Thyme—it’s lightweight and fast:
  https://github.com/Ichoran/thyme
- Just because a pattern is slow it does not follow that this is why your code is slow. impact = time × calls
Avoid the common pitfalls of microbenchmarking

- Read “So You Want to Write a Micro-Benchmark” by John Rose and the linked paper by Brian Goetz: https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks
- Be aware of the top reasons why apparently “correct” microbenchmarks fail, including:
  - Real code requires multiple dispatch, test is single
  - Real code runs with heavily impacted GC, test is not
  - Real code uses results of computation, test does not
  - Real code isn’t even CPU bound, test is (ask a profiler!)
- Use a benchmarking tool to get the details right. If you don’t like the others, try Thyme—it’s lightweight and fast: https://github.com/Ichoran/thyme
- Just because a pattern is slow it does not follow that this is why your code is slow. impact = time × calls

This space intentionally left blank for people in the back
Avoid the common pitfalls of microbenchmarking

- Read “So You Want to Write a Micro-Benchmark” by John Rose and the linked paper by Brian Goetz:
  https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks
- Be aware of the top reasons why apparently “correct” microbenchmarks fail, including
  - Real code requires multiple dispatch, test is single
  - Real code runs with heavily impacted GC, test is not
  - Real code uses results of computation, test does not
  - Real code isn’t even CPU bound, test is (ask a profiler!)
- Use a benchmarking tool to get the details right. If you don’t like the others, try Thyme—it’s lightweight and fast:
  https://github.com/Ichoran/thyme
- Just because a pattern is slow it does not follow that this is why your code is slow.  impact = time \times \text{calls}
Avoid the common pitfalls of microbenchmarking

- Read “So You Want to Write a Micro-Benchmark” by John Rose and the linked paper by Brian Goetz:
  https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks
- Be aware of the top reasons why apparently “correct” microbenchmarks fail, including
  - Real code requires multiple dispatch, test is single
  - Real code runs with heavily impacted GC, test is not
  - Real code uses results of computation, test does not
  - Real code isn’t even CPU bound, test is (ask a profiler!)
- Use a benchmarking tool to get the details right. If you don’t like the others, try Thyme—it’s lightweight and fast:
  https://github.com/Ichoran/thyme
- Just because a pattern is slow it does not follow that this is why your code is slow. impact = time × calls
Can microbenchmarking speed up our example?

StringBuilder.append was hot. Can we do it faster with char arrays?

object BenchEx1 {
  val dict = Seq(
    "salmon", "cod", "grouper", "bass", "herring",
    "eel", "trout", "perch", "halibut", "dorado"
  )
  val cdict = dict.map(_.toCharArray).toArray
  val n = cdict.map(_.length).sum
  def main(args: Array[String]) {
    val th = new ichi.bench.Thyme
    val a = th.Warm{ dict.mkString }
    val b = th.Warm{
      val c = new Array[Char](n)
      var i,j = 0
      while (i < cdict.length) {
        System.arraycopy(cdict(i), 0, c, j, cdict(i).length)
        j += cdict(i).length
        i += 1
      }
      new String(c)
    }
    th.pbenchOffWarm()(a, wtitle="mkString")(b, vtitle="charcat")
  }
}
Microbenchmark + profiler was actionable

$ scala -cp /jvm/Ichi.jar:. BenchEx1.scala
Benchmark comparison (in 4.145 s)
  mkString vs charcat
Significantly different (p \sim 0)
  Time ratio: 0.50403 95% CI 0.50107 - 0.50700 (n=20)
    mkString 250.5 ns 95% CI 249.5 ns - 251.6 ns
    charcat 126.3 ns 95% CI 125.7 ns - 126.8 ns

Char arrays are almost twice as fast in a microbenchmark.

Don’t believe it! Does it hold in real code? Best of five runs:

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m6.400s</td>
<td>0m5.537s</td>
<td></td>
</tr>
</tbody>
</table>

Timing on permuted method (best of 5):

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>1.779 s</td>
<td>1.315 s</td>
</tr>
<tr>
<td>GC</td>
<td>2.343 s</td>
<td>1.875 s</td>
</tr>
</tbody>
</table>
Microbenchmark + profiler was actionable

$ scala -cp /jvm/Ichi.jar:. BenchEx1.scala
Benchmark comparison (in 4.145 s)
  mkString vs charcat
Significantly different (p ~= 0)
  Time ratio: 0.50403 95% CI 0.50107 - 0.50700 (n=20)
  mkString 250.5 ns 95% CI 249.5 ns - 251.6 ns
  charcat 126.3 ns 95% CI 125.7 ns - 126.8 ns

Char arrays are almost twice as fast in a microbenchmark.

Don’t believe it! Does it hold in real code? Best of five runs:

<table>
<thead>
<tr>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m6.400s</td>
<td>0m5.537s</td>
</tr>
</tbody>
</table>

Timing on permuted method (best of 5):

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>1.779 s</td>
<td>1.315 s</td>
</tr>
<tr>
<td>GC</td>
<td>2.343 s</td>
<td>1.875 s</td>
</tr>
</tbody>
</table>
Microbenchmark + profiler was actionable

$ scala -cp /jvm/Ichi.jar:. BenchEx1.scala

Benchmark comparison (in 4.145 s)

mkString vs charcat

Significantly different (p ~ = 0)

<table>
<thead>
<tr>
<th>Time ratio: 0.50403 95% CI 0.50107 - 0.50700 (n=20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkString 250.5 ns 95% CI 249.5 ns - 251.6 ns</td>
</tr>
<tr>
<td>charcat 126.3 ns 95% CI 125.7 ns - 126.8 ns</td>
</tr>
</tbody>
</table>

Char arrays are almost twice as fast in a microbenchmark.

Don’t believe it! Does it hold in real code? Best of five runs:

<table>
<thead>
<tr>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m6.400s</td>
<td>0m5.537s</td>
</tr>
</tbody>
</table>

Timing on permuted method (best of 5):

<table>
<thead>
<tr>
<th>Original</th>
<th>With char array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>1.779 s</td>
</tr>
<tr>
<td>GC</td>
<td>2.343 s</td>
</tr>
<tr>
<td></td>
<td>1.315 s</td>
</tr>
<tr>
<td></td>
<td>1.875 s</td>
</tr>
</tbody>
</table>
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
A word about timing methodology

- All timings are from warmed Thyme microbenchmarks.
- Timings may have been subtracted from each other
- Assumes GC system is not heavily taxed
- These are guidelines not truths. If it’s essential, measure in your context (architecture, JVM, etc. etc.).

This space intentionally left blank for people in the back
A word about timing methodology

- All timings are from warmed Thyme microbenchmarks.
- Timings may have been subtracted from each other
- Assumes GC system is not heavily taxed
- These are guidelines not truths. If it’s essential, measure in your context (architecture, JVM, etc. etc.).
A word about timing methodology

- All timings are from warmed Thyme microbenchmarks.
- Timings may have been subtracted from each other
- Assumes GC system is not heavily taxed
- These are guidelines not truths. If it’s essential, measure in your context (architecture, JVM, etc. etc.).
A word about timing methodology

- All timings are from warmed Thyme microbenchmarks.
- Timings may have been subtracted from each other
- Assumes GC system is not heavily taxed
- These are *guidelines* not *truths*. If it’s essential, measure in your context (architecture, JVM, etc. etc.).
Boxing

- Turtles: mutable, copy, \( f:T \rightarrow T \), Shapeless lens
- Method vs. implicit class enriched method
- Method vs. value class enriched method
- Object: method vs. structural type
- Object method vs. boxed object method
- Array summation: ints vs. boxed ints
- Array creation: ints vs. boxed ints
Control flow

new exception with stack
new control-flow exception
local stackless preallocated exception

inner for
inner tailrec
inner loop-with-return
inner while loop
manually unrolled loop-in-a-loop
simple loop

with match
with if-else
with indicator
with &

Iterator
For (range)
While loop with anon function
Tail recursion
While loop

Nanoseconds per operation
Inheritance

- Multimorphic: 8 of 8, pattern match
- Multimorphic: 8 of 8, inheritance
- Multimorphic: 4 of 8, pattern match
- Multimorphic: 4 of 8, inheritance
- Multimorphic: 4 of 4, pattern match
- Multimorphic: 4 of 4, inheritance
- Multimorphic: 2 of 4, pattern match
- Multimorphic: 2 of 4, inheritance
- Multimorphic: 2 of 2, pattern match
- Multimorphic: 2 of 2, inheritance

Method call typed as superclass
Method call typed as implementing subclass
Just code

Nanoseconds per operation

This space intentionally left blank for people in the back
Mathematics

- BigInt /, 1000 & 500 digits
- BigInt *, 1000 & 500 digits
- BigInt +, 1000 & 500 digits
- BigInt /, 100 & 50 digits
- BigInt *, 100 & 50 digits
- BigInt +, 100 & 50 digits
- BigInt /, 10 digits
- BigInt *, 10 digits
- BigInt +, 10 digits

- pow(x, 0.3)
- sin(x)
- log(x)
- with / x
- with / 3.0
- double loop with +, *

Nanoseconds per operation
Collections

This space intentionally left blank for people in the back
Parallelization

This space intentionally left blank for people in the back

Rex Kerr (JFRC)  Designing for Performance  31 / 37
Outline

1. Conventional wisdom
   - Three things you may have heard that may not be entirely true

2. Profiling the Big Picture
   - What you should and should not expect from your profiler

3. Microbenchmarking the Small Picture
   - How to write an actionable microbenchmark

4. Timings
   - Building intuition about how long things take

5. Strategic Summary
   - Suggestions for Performance-Aware Design
Step one: understand/define requirements

- Is performance a concern at all?
- Is performance in your control at all (is the slow part an external service?)
- Where does speed matter
  - Visual system is happy with ~10–20 ms worst case
  - Anything interactive seems instant with latency of \( \leq 100 \text{ ms} \)
- Do you need to optimize for latency or throughput?
Step two: identify the likely bottlenecks

- What do you need to do a *lot*? That’s probably where the bottleneck will be.
- Understand what “a lot” is—adding a million ints is not “a lot” compared to a single ping across a typical network.
- Ask: are you using the right algorithms?
- Isolate performance-critical pieces in a modular way
- Use parallelism with the correct amount of work
  - Overhead is considerable
  - Deciding how to split is (may be) serial
Step three: measure performance early and often

- Set up so performance measurements are painless
- Only fix immediately if performance is alarmingly bad and might require a complete redesign
  - A system that does not work has zero performance
- Use the REPL to microbenchmark bits of code
  - (You are already testing/building bits of code in the REPL, right?)
- Don’t waste time measuring things that clearly don’t matter
  - (Your measurements will tell you what doesn’t matter, right?)
Step four: refine working system

- Catch surprises with a profiler
- Get an idea of the big picture with a profiler
- Refine hotspots by
  - choosing a more efficient algorithm
  - choosing higher-performance language constructs
  - choosing a higher-performance library
  - microbenchmarking (possibly in place in running code!)
- Don’t forget that code needs to be maintained—if you do something really clever/nonobvious, try to encapsulate it and explain why it’s done that way
- Don’t fall victim to “never X” rules. There are tradeoffs; make the compromises that serve you best.
Final thoughts: speed levels

- **Sub-ns**
  - Int +*&, single iteration of loop/tailrec, in-order array, var write, method call

- **One ns**
  - Conditional, multicast method call up to 2, non-escaping object “creation”, constant int division, floating point +*

- **A few ns**
  - Object creation, single generic collections operation, division, throw/catch existing stackless exception, compare-and-swap, synchronized (uncontended), @volatile

- **Tens of ns**
  - Single set/map operation, trig/exp/log, throw/catch new stackless exception, multicast method over 3+ classes, anything without JIT, structural typing, small BigInt

- **Hundreds of ns or more**
  - Handing off data between threads, big BigInt; throw/catch exception with stack trace, futures; parallel collection operation overhead