Profilers Are Lying Hobbitses

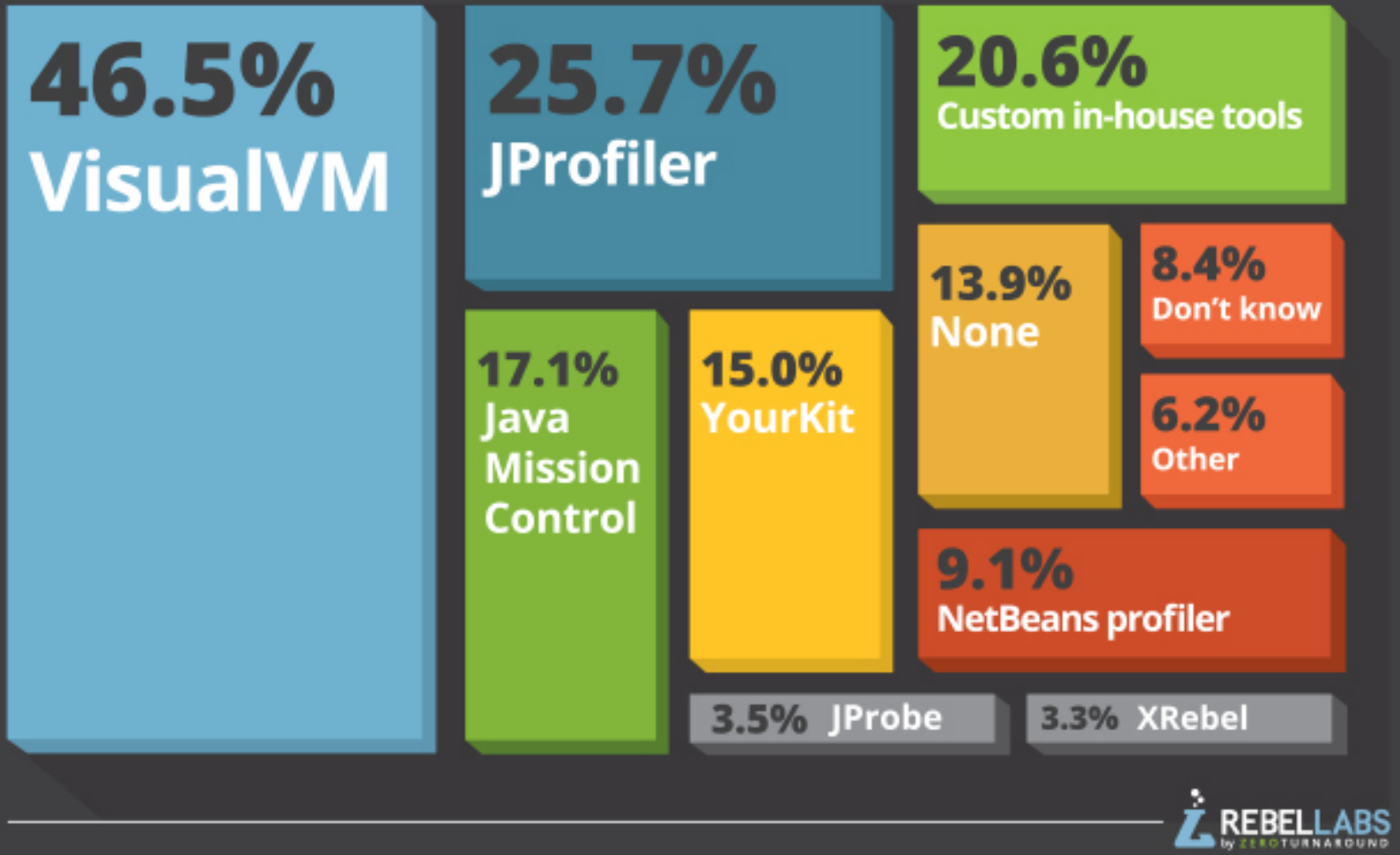
Nitsan Wakart (@nitsanw)
Performance Consultant, TTNR Labs


What are the typical root causes you most often experience[1]
Figure 1.16
28.4% Concurrency issues
18.5% Slow DB
27.6% Memory leak
1.7% Other
4.4% Don't know
Slow/unreliable third party entities 12.0%
13.3% Excessive disk IO
54.8% Slow database queries
38.6% Too many database queries
51.5% Inefficient application code
Excessive network IO 12.9%
17.9% GC pauses
23.4% Configuration issues*
Excessive memory churn 10.6%
*pooling thresholds/ request throttling
7.6% HTTP session bloat
REBELLABS by ZEROTURNAROUND
[1]Answers were multiple choice, so the numbers don't add up to 100%. Deal with it :)

0. Do U even profile?

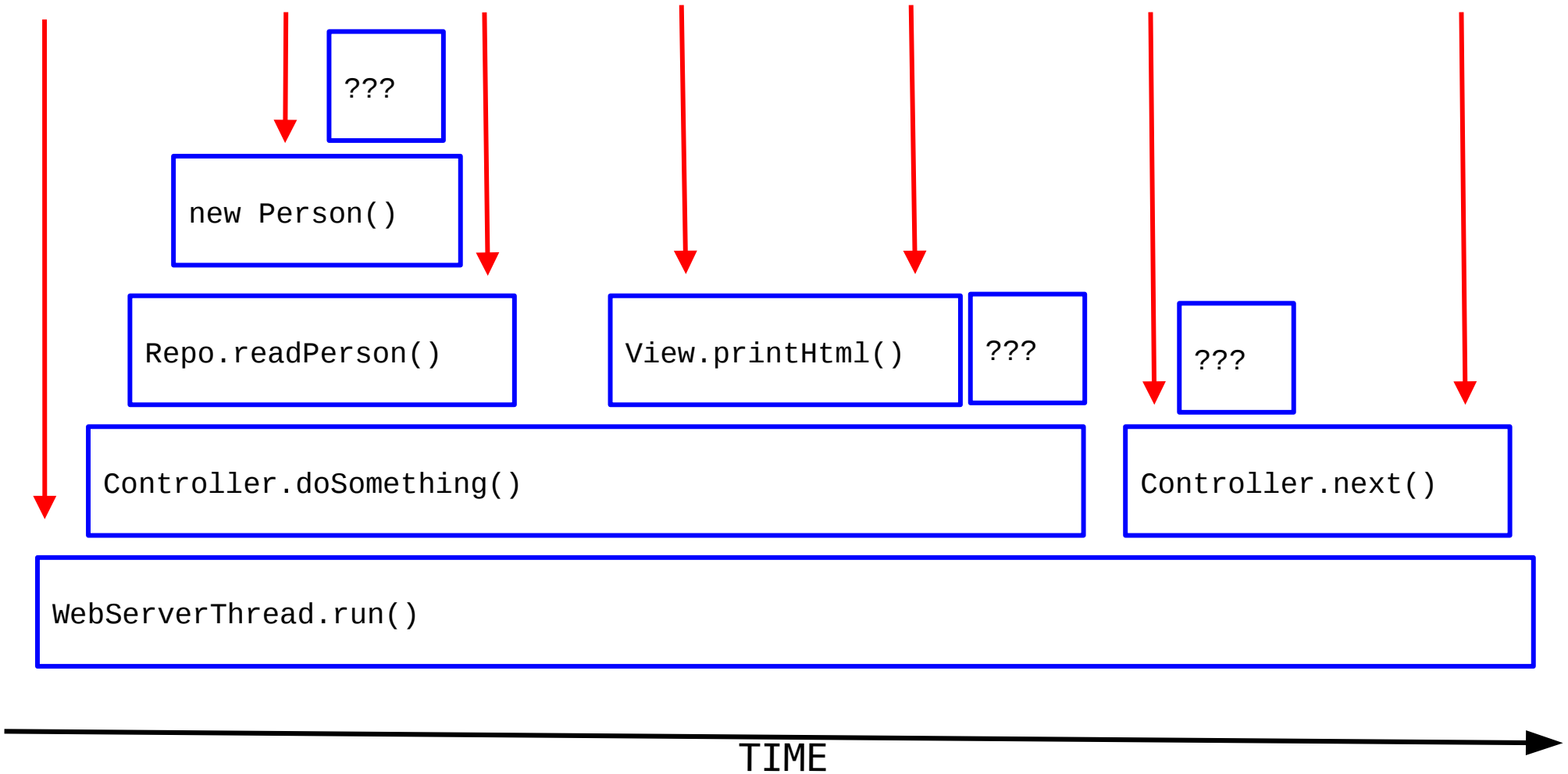
Which tools do you use for application profiling?

Figure 1.12



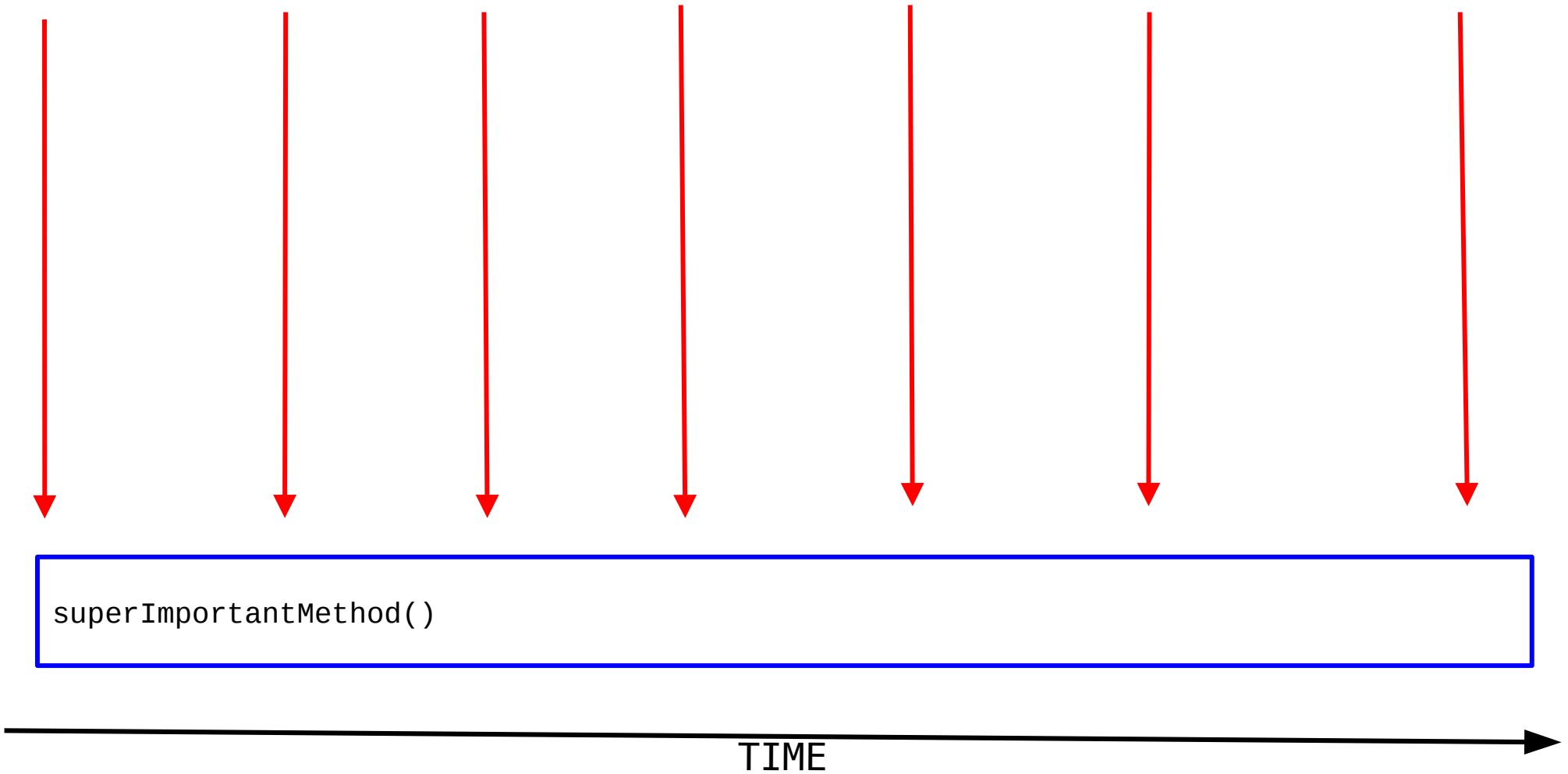
Sampling Profilers

- Sample program on interval
- Distribution of samples highlights hotspots
- **Assumption**: *Samples are 'random'*
- **Assumption**: Sample distribution approximates 'Time Spent' distribution


???
new Person()
Repo.readPerson()
View.printHtml()
???
???
Controller.doSomething()
Controller.next()
WebServerThread.run()
TIME

LOC in Method?

- Find slow lines of code
- Optimize
- WIN

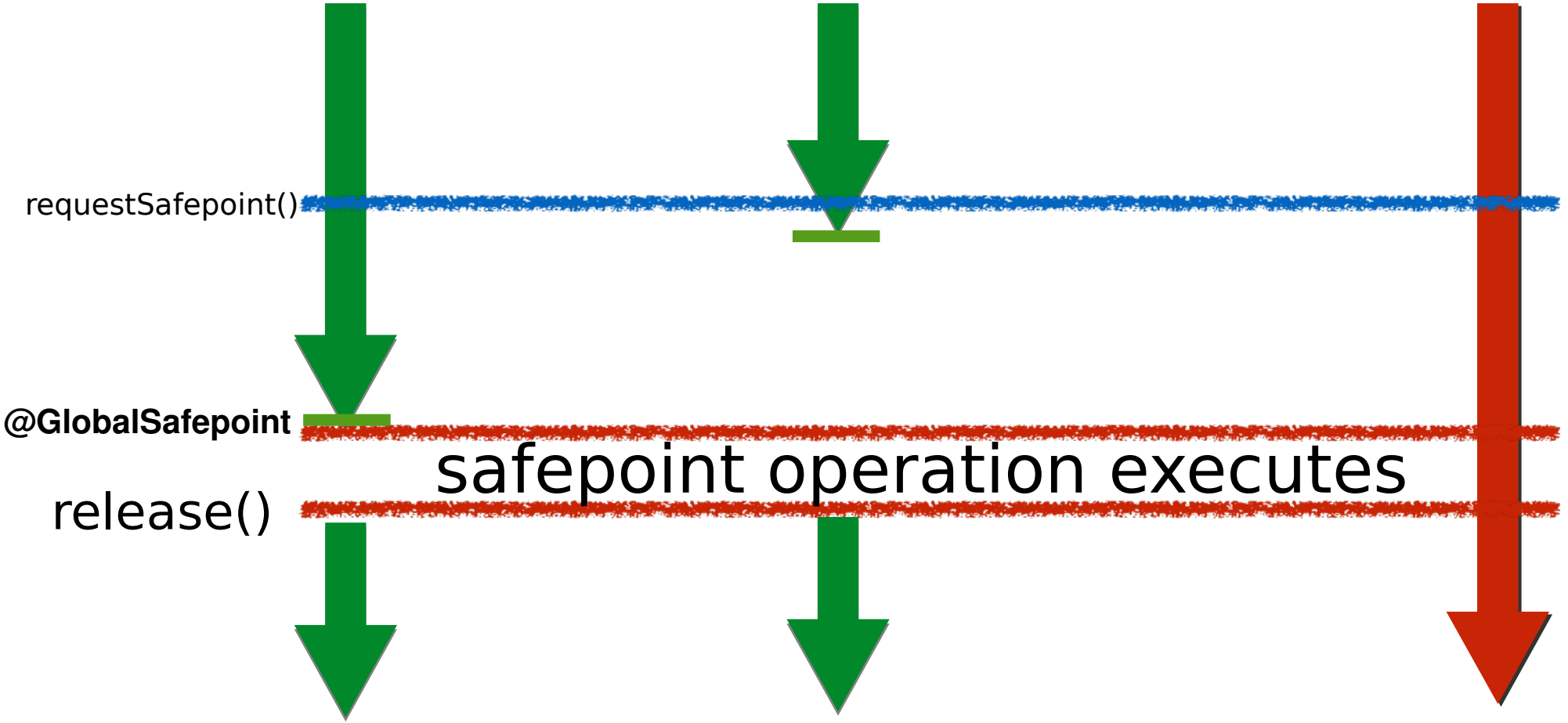

superImportantMethod()
TIME

```
public void superImportantMethod()
{
  for (int i = 0; i < buffer.length; i++) // Line 1
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1; // Line 13
}
```

1. JVisualVM much?

Most Java Profilers: Safepoint Bias

- Samples only at **safepoint polls**
- Each sample is a **safepoint operation**
- Each sample includes **all** threads


requestSafepoint()
@GlobalSafepoint
safepoint operation executes
release()

Safepoint Poll

- Method exit
- Some loop back edge

```
public void superImportantMethod()
{
  for (int i = 0; i < buffer.length; i++)
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);//5 safepoint?
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];//10 safepoint?
  }

  result = (sum & 1) == 1;//13 safepoint?
}
```

```
public void superImportantMethod()
{
  for (int i = 0; i < buffer.length; i++) // 100% blame!
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1;
}
```

```
public void superImportantMethod()
{
  final int divisor = this.divisor; // hoist volatile read
  for (int i = 0; i < buffer.length; i++) // 1?
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++) // 2?
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1; // 3?
}
```

```
public void superImportantMethod() // Oracle blame 0%!!!
{
  final int divisor = this.divisor; // Zing blame 100%
  for (int i = 0; i < buffer.length; i++)
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1;
}
```

Safepoint biased profiles are:

a) Tricksy

b) Sneaksy

c) Filthy

d) All of the above

2. But... Mission Control!?

Better Profilers: No Safepoint Bias

- Sample at **signal**
- Each sample includes **1** thread
- JMC/honest-profiler/async-profiler

```java
public void superImportantMethod()
{
  final int divisor = this.divisor;
  for (int i = 0; i < buffer.length; i++)
  {
    dst[i] = buffer[i];// Blame 100%!!!
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1;
}
```

Better Profilers Need DebugInfo

- By default, too little debug info generated
- Enable : -XX:+DebugNonSafepoint
- Should be the default.... but it ain’t

```
public void superImportantMethod()
{
  final int divisor = this.divisor;
  for (int i = 0; i < buffer.length; i++) // 1.8%
  {
    dst[i] = buffer[i]; // 2.1%
    buffer[i] = (byte) (buffer[i] / divisor); // 9.4%
    buffer[i] = (byte) (buffer[i] * multiplier); // 66.6%
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++) // 1.9%
  {
    sum += buffer[i] + dst[i]; // 18%
  }

  result = (sum & 1) == 1;
}
```

2.1 O'Mission Control

JFR Code Profiling is Borken

- Failed samples silently ignored
  - Can’t see ‘stubs’ (e.g. array copy/fill, crc32...)
  - Some native code (e.g. NIO select)
- No native code/threads

Honest-Profiler

- Failed samples reported
- Still can’t see ‘stubs’
- Native code still missing

Async-Profiler

- Failed samples reported AND minimized
- Shows native, user AND KERNEL!!!
- No LOC precision (currently)

JFR profiles:

a) MUST HAVE

DebugNonSafepoints

b) Watch out for sample count

c) Great improvement over JVisualVM

3. Instruction Profiling?

Code?

- There is no Code -> No Line of Code
- There's no Bytecode -> No BC Index
- Only instructions...

Sample The **P**rogram **C**ounter!

Program Counter/Instruction Pointer

The instruction pointer (EIP) register contains the offset in the current code segment for the **next instruction to be executed**.

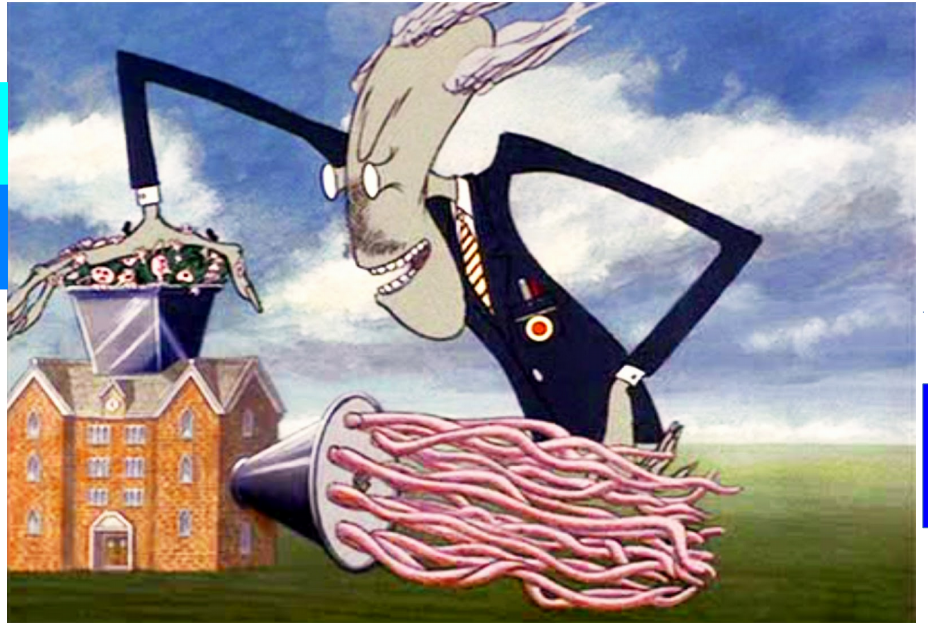
```
R  Address                          Instruction
E  0x00007fbaa414e3da: mov    %rax,0x30(%rbx)
D  0x00007fbaa414e3de: mov    %r12,0x20(%rbx)
F  0x00007fbaa414e3e2: add    $0x30,%rsp
   0x00007fbaa414e3e6: pop    %rbp
   0x00007fbaa414e3e7: test   %eax,0x15e1dc13(%rip)
   0x00007fbaa414e3ed: retq
```

PC

Fetch

Decode

Execute



Retire

Program Counter

- The next instruction to be fetched
- How do we sample it?

Sampling PC Mechanics

- Install a Signal Handler
- Set up a counter to signal on overflow
- When handler invoked -> ucontext

```
                    ↗  0x00007f4e611f9ca0: movslq %r9d,%r8
         0.02%      |  0x00007f4e611f9ca3: vmovd  0x10(%r13,%r8,1),%xmm1
 1.97%   3.12%      |  0x00007f4e611f9caa: vmovd  %xmm1,0x10(%rsi,%r8,1)  ;*bastore; superImportantMethod@27 (line 3)
                    |  0x00007f4e611f9cb1: movsbl 0x10(%r13,%r8,1),%eax  ;*baload; superImportantMethod@38 (line 4)
                    |  0x00007f4e611f9cb7: cmp    $0x80000000,%eax
                    |  0x00007f4e611f9cbc: jne    0x00007f4e611f9cc6
                    |  0x00007f4e611f9cbe: xor    %edx,%edx
                    |  0x00007f4e611f9cc0: cmp    $0xffffffffffffffff,%r14d
                    |  0x00007f4e611f9cc4: je     0x00007f4e611f9cca
 0.07%   0.04%      ↘  0x00007f4e611f9cc6: cltd
 2.23%   3.52%      |  0x00007f4e611f9cc7: idiv   %r14d              ;*idiv; superImportantMethod@40 (line 4)
 8.85%   4.92%      ↘  0x00007f4e611f9cca: movslq %r9d,%rdi
                    |  0x00007f4e611f9ccd: movsbl %al,%eax
 1.71%   0.86%      |  0x00007f4e611f9cd0: imul   %r11d,%eax
 6.44%   4.83%      |  0x00007f4e611f9cd4: mov    %al,0x10(%r13,%r8,1)  ;*bastore; superImportantMethod@60 (line 5)
 2.35%   0.80%      |  0x00007f4e611f9cd9: movsbl 0x11(%r13,%rdi,1),%eax  ;*baload; superImportantMethod@38 (line 4)
```

```
public void superImportantMethod()
{
  final int divisor = this.divisor;
  for (int i = 0; i < buffer.length; i++) // 1.8%
  {
    dst[i] = buffer[i]; // 2.1%
    buffer[i] = (byte) (buffer[i] / divisor); // 9.4%
    buffer[i] = (byte) (buffer[i] * multiplier); // 66.6%
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++) // 1.9%
  {
    sum += buffer[i] + dst[i]; // 18%
  }

  result = (sum & 1) == 1;
}
```

```
0.07%  cltd
2.23%  idiv   %r14d                    ;*idiv;@40 (line 4)
8.85%  movslq %r9d,%rdi
       movsbl %al,%eax
1.71%  imul   %r11d,%eax
6.44%  mov    %al,0x10(%r13,%r8,1)  ;*bastore;@60 (line 5)
2.35%  movsbl 0x11(%r13,%rdi,1),%eax  ;*baload;@38 (line 4)
```

That’s good, right?

- Precision == instruction
- Accuracy == cycle-ish?

-ish?

“> I think Andi mentioned this to me last year --
> that instruction profiling was no longer reliable.

`It never was.`”

http://permalink.gmane.org/gmane.linux.kernel.perf.user/1948
Exchange between Brenden Gregg and Andi Kleen

Model & Tool Issues

- Non-uniform instruction cost
- Pipelined + Super Scalar CPU
- Out-of-order + Speculative execution
- Signal latency (a.k.a Skid)

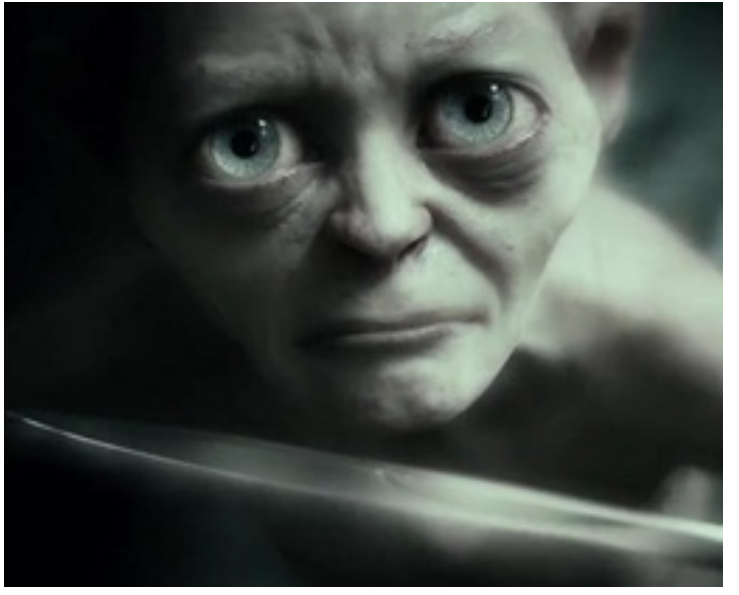
The blamed instruction is often **'shortly' after** where the big cost lies

PEBS

- Precise Event-Based Sampling
- Supported:
  - Branch events
  - Cache events
  - Instructions

Borken?

- Limited accuracy
- Works well enough for assembly: Oracle Studio/Zvision/VTune
- I <3 ASM?



4. Instructions to LOC?

PC -> BCI

- Instruction -> Byte Code Index
- Where do instructions come from?

JIT Compiler: BC -> Instruction

- 1 : 1?
- 1 : N?
- N : 1?
- M : N???
- 0 : N?!

JIT Compiler DebugInfo

- Works well for 1:1, 1:N
- Otherwise, pick the closest!
- Enable -XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints

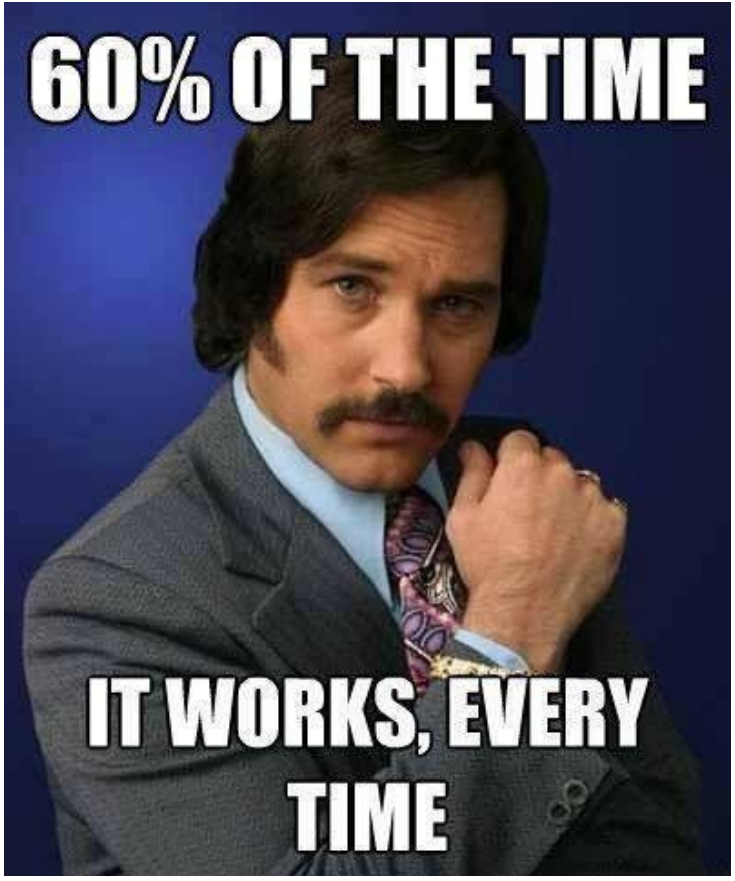
BCI → Line of Code

- BCI – Byte Code Index
- Not every BCI has a line of code
- Find the closest...

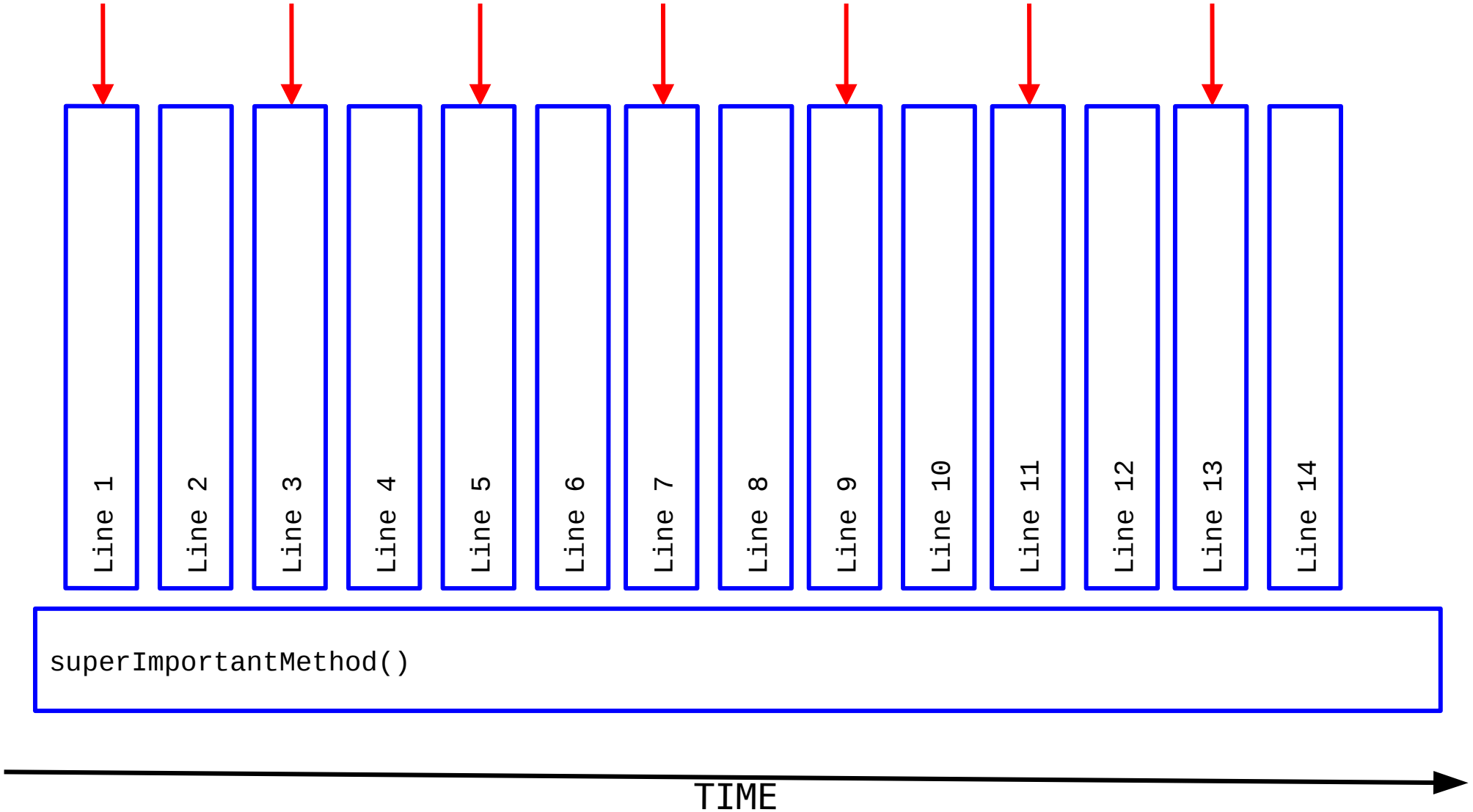
Look in hprof for example: <OPENJDK-HOME>/demo/jvmti/hprof

PC → BCI → Line of Code

- This is as good as it gets
- Look for other suspects nearby

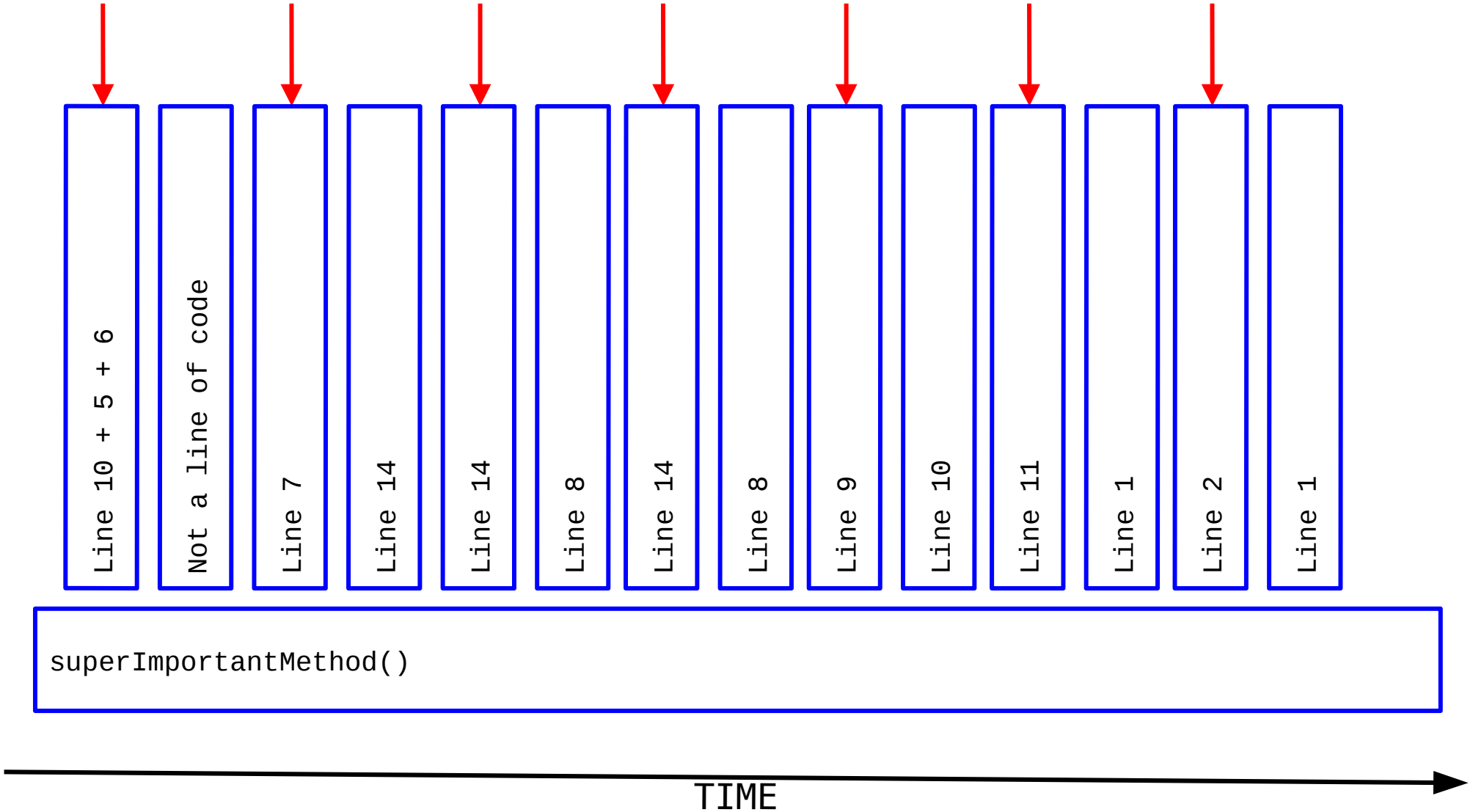


Nearby?


Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
Line 13
Line 14
superImportantMethod()
TIME

Compiler Remixed!

- Scheduling
- Runtime instructions
- Loop unrolling
- Constant folding
- Superword!
- AND MUCH MUCH MORE!


Line 10 + 5 + 6
Not a line of code
Line 7
Line 14
Line 14
Line 8
Line 14
Line 8
Line 9
Line 10
Line 11
Line 1
Line 2
Line 1
superImportantMethod()
TIME

LOC in Method: Kinda Works

- Look for obvious wins
- Use common sense
- Look at nearby lines
- Maybe look at ASM?

5. Instructions to Methods?

Method Granularity?

- PC -> Method!
- Find slow methods
- Optimize
- WIN

Safepoint Poll

- **Method exit**
- Should be fine, right?

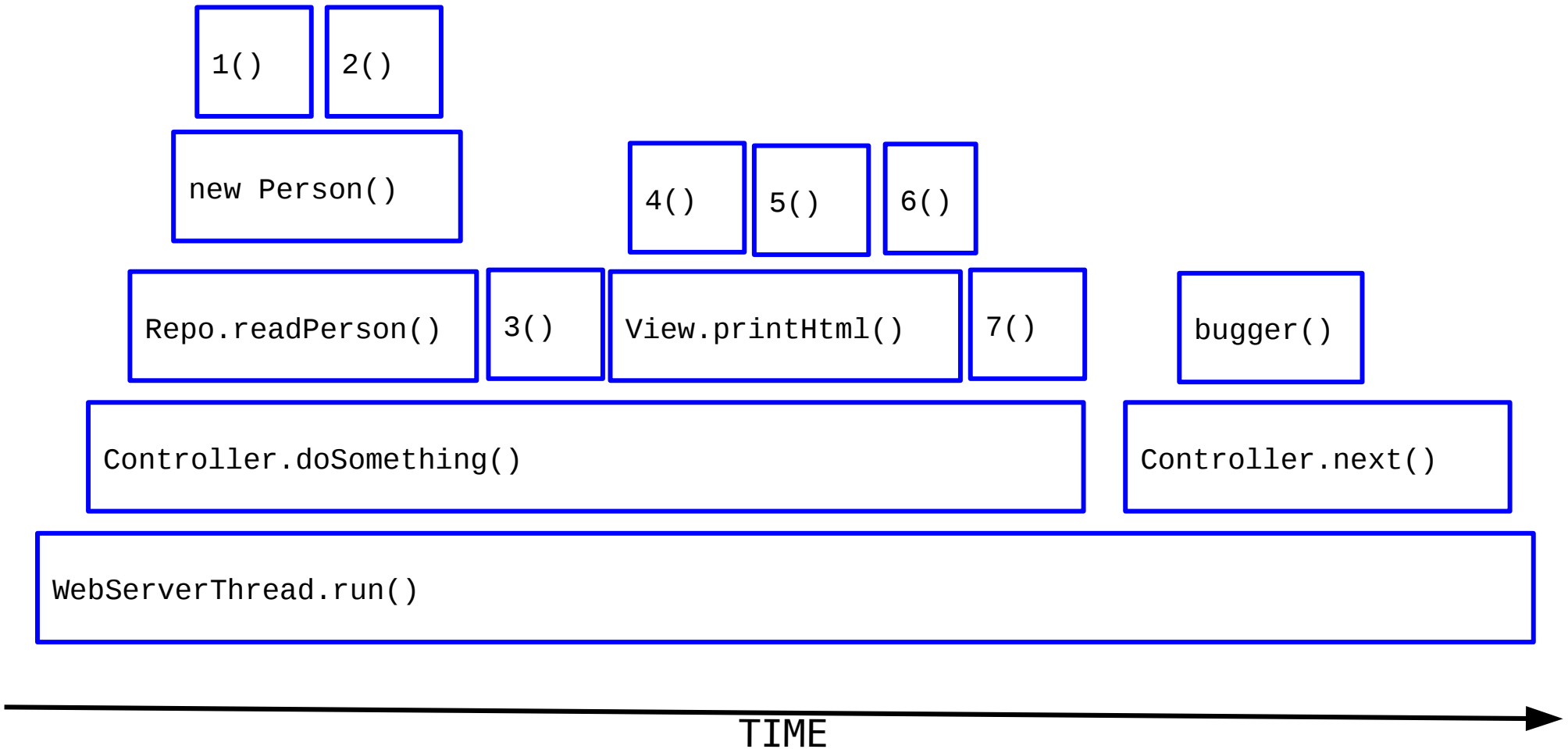
```
public void superImportantMethod() // Oracle blame 0%!!!
{
  final int divisor = this.divisor;
  for (int i = 0; i < buffer.length; i++)
  {
    dst[i] = buffer[i];
    buffer[i] = (byte) (buffer[i] / divisor);
    buffer[i] = (byte) (buffer[i] * multiplier);
  }

  int sum = 0;
  for (int i = 0; i < buffer.length; i++)
  {
    sum += buffer[i] + dst[i];
  }

  result = (sum & 1) == 1;
}
```

Safepoint Poll

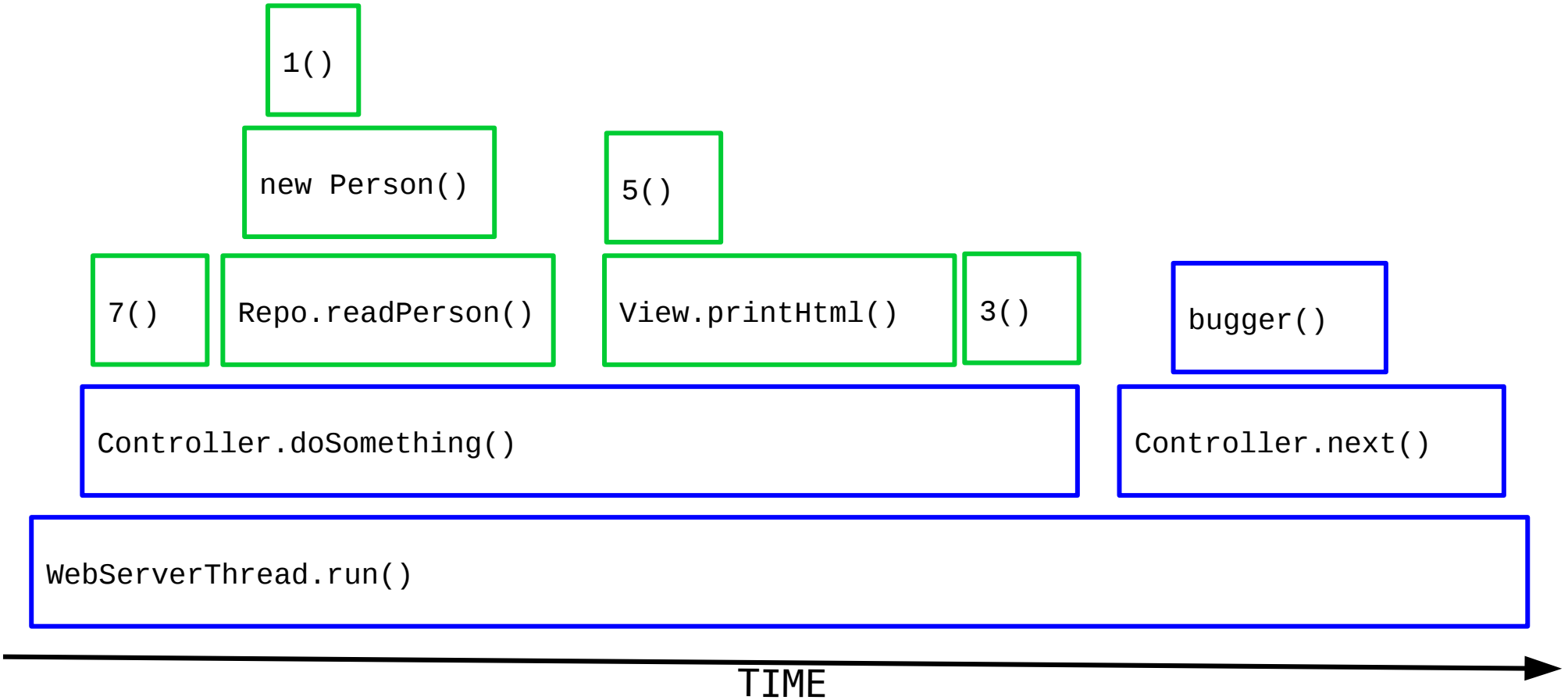
- The return line of code is blamed instead...
- ... so still need to look at line of code
- That is, if there is a method...

Inlining

- Replace method call with method!!!
- Great optimization!
- Removes safepoint poll...
- Expands optimization horizon...


1()
2()
new Person()
4()
5()
6()
Repo.readPerson()
3()
View.printHtml()
7()
bugger()
Controller.doSomething()
Controller.next()
WebServerThread.run()
TIME

Inlining + Reordering

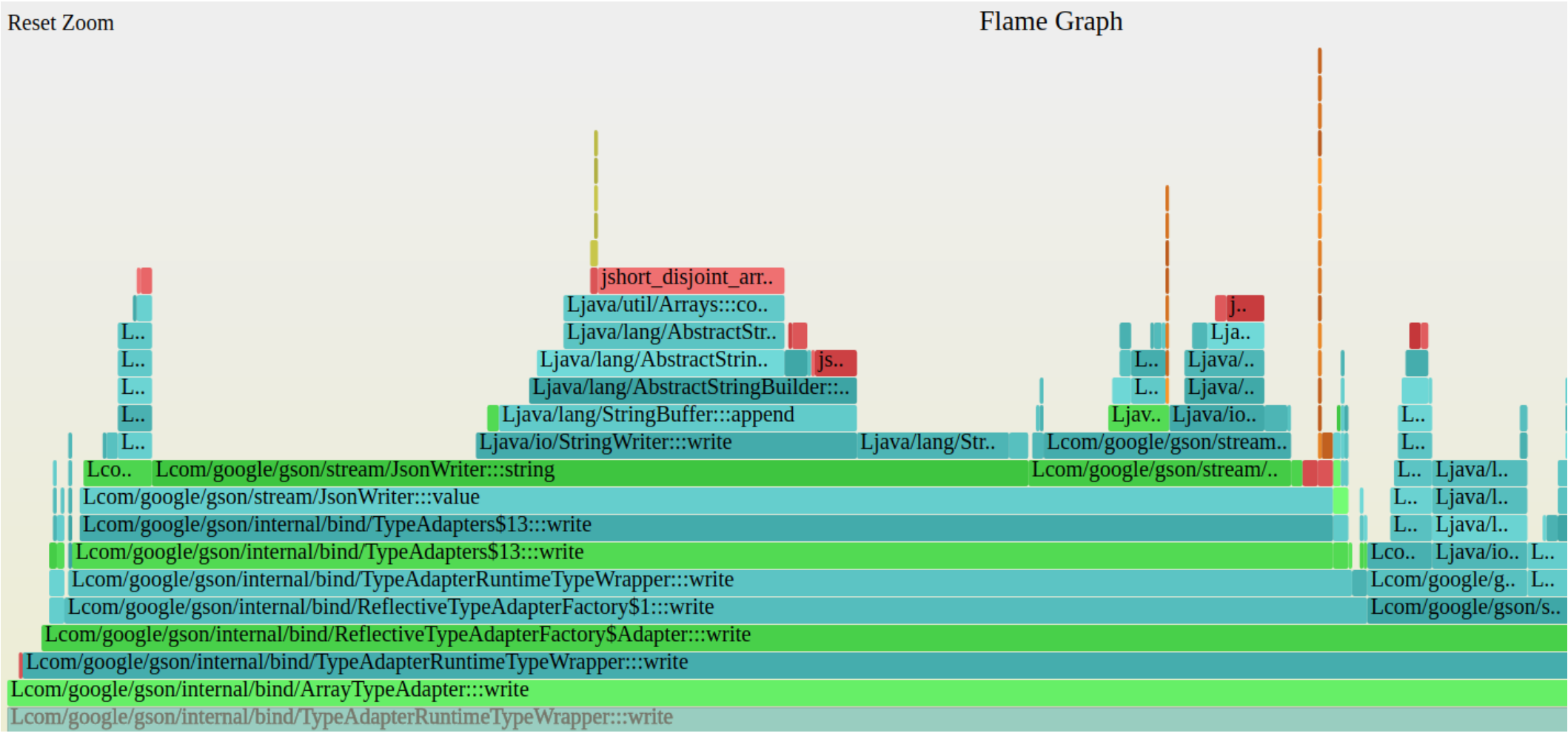


Inlining vs. Profilers

- Can cause great confusion
  - LOC from different methods are instructions apart
  - Easy to skid between methods
- Need inlining information
- Checkout perf-map-agent + perf + FlameGraph

Inlining Scope

- MaxInlineLevel=9 (and a bit)
- FreqInlineSize=325
- Large compiled methods are common
  - And that’s a good thing


Reset Zoom
Flame Graph
jshort_disjoint_arr..
Ljava/util/Arrays:::co..
Ljava/lang/AbstractStr..
Ljava/lang/AbstractStrin..
Ljava/lang/AbstractStringBuilder:::..
Ljava/lang/StringBuffer:::append
Ljava/io/StringWriter:::write
Ljava/lang/Str..
Lcom/google/gson/stream/JsonWriter:::string
Lcom/google/gson/stream/..
Lcom/google/gson/stream/JsonWriter:::value
Lcom/google/gson/internal/bind/TypeAdapters$13:::write
Lcom/google/gson/internal/bind/TypeAdapterRuntimeTypeWrapper:::write
Lcom/google/gson/internal/bind/ReflectiveTypeAdapterFactory$1:::write
Lcom/google/gson/internal/bind/ReflectiveTypeAdapterFactory$Adapter:::write
Lcom/google/gson/internal/bind/ArrayTypeAdapter:::write
Lcom/google/gson/internal/bind/TypeAdapterRuntimeTypeWrapper:::write
Ljava/..
Ljava/io..
Ljava/l..
Lcom/google/g..
Lcom/google/gson/s..

Method Granularity: Kinda Works

- Real frames
- Virtual frames
- Maybe look at ASM?

6. Which Method?

Native Profilers vs. JIT

- Rely on static debug info
- JIT -> no static debug info
- Needs mapping

Compilation over time

- V0 – Interpreter
- V1 – C1/client (level 3)
- V2 – C2/server (level 4)
- V3 – deoptimize/reoptimize
- V4 – deoptimize/reoptimize ...

Inlined Callsite Versions

- Method A -> Method B = X
- Method C -> Method B = Y
- X != Y

7. Utilization?

Comparing Method Utilization

- 20% of samples in Method A
- 80% in Method B

Which one to optimize?

Concurrency vs. Profilers

- Need to record expected max sample
  - Allows per thread utilization observation
- Need to record tid
  - Use perf+PMA

Questions???

Thanks!