ORACLE®

Keep Learning with Oracle University

ORACLE
UNIVERSITY

Classroom Training

Learning Subscription

Live Virtual Class

Training On Demand

Cloud

Technology

Applications

Industries

education.oracle.com

Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

Improving the Performance of Your Java Application: Getting Beyond the Basics

Yukon Maruyama and Marty Itzkowitz

Performance Analyzer Leads

Oracle Solaris Studio

October 29, 2015

Program Agenda

- Introduction: What is a Performance Problem
- Performance Analysis Tools
- Profiling Accuracy: a Simple Example
- Understanding Java Execution
- Identifying Programming Inefficiencies

What is a Performance Problem?

- Objective (quantitative) criteria:
  - It can't handle the required load
  - It consumes too many resources to do its work
- Is it worth fixing?
- Subjective (qualitative) criteria:
  - It takes too long to finish
  - It responds too slowly
  - Cost of fixing vs. aggregate cost of problem
- Most untuned codes have low-hanging fruit!

Triaging Performance Problems

- Is there a problem?
  - **`/bin/time`**, stopwatch, wall clock reveal problems
- Can you do repeatable performance runs?
  - With realistic data and scale, and test case for any specific problems
- Is it a scaling problem?
  - Most programs have some intrinsic scale factor N
    - Compare times with small, medium, and large N
      - ~ N? ~ ln N? ~ N^2? ~ N^m?
- Is it a global performance issue or a corner-case?
- Is it a problem with average performance, or distribution?

JavaOne ORACLE

Diagnosing Performance Problems

- Diagnosis needs repeatable test cases
- As in medical diagnosis, performance diagnosis requires measurement
  - Medical diagnosis
    - Blood pressure, temperature, blood chemistry, ...
    - X-ray, MRI, CT-Scan -- relate to body structure
    - Angiography -- time-based behavior
  - Program performance diagnosis
    - Resource usage, run-time, transactions/second, ...
    - Performance data relating to program structure
    - Time-based picture of execution
- Tools are needed to do the measurements

Program Agenda

- Introduction: What is a Performance Problem
- Performance Analysis Tools
- Profiling Accuracy: a Simple Example
- Understanding Java Execution
- Identifying Programming Inefficiencies

Performance Analysis Tools: Requirements, I

- Accuracy
  - Preserve behavior of the application being tested
    - Some tools and technologies are better than others
    - Minimize increase in running time under measurement
  - Capture key data to show what happened
  - Data collected should be unbiased
    - No difference in the relative weights of elements in the application
    - Accurate representation of what execution would be without measurement
- Accuracy is NOT a given
  - Some tools distort behavior and the measured data

Performance Analysis Tools: Requirements, II

- Scalability
  - To target problem size
    - *e.g.*, size of database being queried
    - *e.g.*, number of clients accessing a server
  - Program size – multi-GB executables, 100,000 source files
  - Thread and CPU count – potentially 1000's of each in modern systems
  - Running time – 10's of seconds to many hours

Data Collection Options: Statistical Sampling

- Statistical sampling
  - Pro: very scalable: can throttle profile rate as needed
    - Millions of instructions between samples
  - Con: risk of non-representative sampling
    - May miss non-repeating short-duration events (but are they significant for performance?)
    - Behavior correlated with sampling mechanism
- Trigger by profiling clock-tick
  - Shows where CPU time is spent
  - On Solaris, also shows why program is not running: wait for I/O, page -fault, *etc.*
- Trigger by HW Counter overflow: CPU Stalls, cache misses, *etc.*
- Recommended for general performance analysis

Data Collection Options: Tracing

- Tracing interesting events
  - Mechanisms
    - Instrumentation of key methods
    - Interposition on library functions
    - JVMTI event generation
  - Pro: data based on the behavior traced
    - Memory allocation/deallocation
    - I/O operation tracing
    - Synchronization operations
  - Con: requires careful interposition
    - Scales poorly: can distort behavior significantly
- Recommended <u>only</u> if necessary to get the specific data needed

Data Collection Options: Capturing the Callstack

- Callstack capture can be expensive
  - However, unwind cost for sampling is usually only a few percent
- Full callstack capture is recommended
  - Provides dynamic calltree with performance metrics
    - Inclusive time (total time) is time in the function + time in all the functions it calls
    - Exclusive time (self time) is time in the function only
  - Essential for identifying hot branches (*e.g.*, callers of access functions)
- Java stack unwind is not supported equally by all tools
  - Deferred sampling to avoid JVM safepoints will give incorrect data
  - JIT inlining can be difficult for a tool to understand and represent
  - Callstacks after native calls (JNI) are often not supported

Program Agenda

- Introduction: What is a Performance Problem
- Performance Analysis Tools
- Profiling Accuracy: a Simple Example
- Understanding Java Execution
- Identifying Programming Inefficiencies

Profiling Accuracy: a Simple Example

- Example code does the same calculation two ways
  - One with an explicit triple-nested loop doing a sum
  - One with each level of the loop calling a method for the next level
    - The inner-most level does the sum
  - Expected performance of the two is about the same
- Data collected with three different profilers
  - Java Flight-Recorder (from Java SE 1.8.0_40)
  - NetBeans Profiler (NB 8.0.1, run with Java SE 1.8.0_40)
  - Studio Performance Analyzer (run with Java SE 1.8.0_40)
- Source code on the next two slides
  - Try your own favorite tool

Profiling Accuracy: Source Code, I

```
// Some profilers may not identify that ~50% of time is spent in innerSum()
public class InliningTest {
    public static void main(String[] args) {
        InliningTest test = new InliningTest();
        for (int i = 0; i < 20; i++) {
            test.computeSimple();
            test.computeDeep();
        }
    }
// - computeSimple() - Computes a sum in a triply nested loop
    public double sum = 0.0;
    public void computeSimple() {
        for (int i = 0; i < 1000; i++) {                      // outer
            for (int j = 0; j < 1000; j++) {                  // middle
                for (int k = 0; k < 1000; k++) {              // inner
                    sum += 1.0;               // ~48% of time // sum
                }
            }
        }
    }
```

Profiling Accuracy: Source Code, II

```
// - computeDeep() - Computes the same sum, but implements the
//             outer, middle, and inner loops as nested methods.
    public double sum_deep = 0.0;
    public void computeDeep() {
        for (int i = 0; i < 1000; i++) { nested_level_1(); }  // outer
    }
    private void nested_level_1() {
        for (int i = 0; i < 1000; i++) { nested_level_2(); }  // middle
    }
    private void nested_level_2() {
        for (int i = 0; i < 1000; i++) { innerSum(); }        // inner
    }
    private void innerSum() {
        sum_deep += 1.0;                     // ~48% of time // sum
    }
}
```

Profiling Accuracy: Expanded Calltree (NetBeans Profiler)

(Runtime dilated from 34 to 40.5 seconds)

Call Tree - Method	Total Time
main	40,531 ... (100%)
InliningTest.main (String[])	40,531 ... (100%)
InliningTest.computeDeep ()	39,449 ...(97.3%)
Self time	39,344 ...(97.1%)
InliningTest.nested_level_1 ()	104 ms (0.3%)
InliningTest.computeSimple ()	876 ms (2.2%)
Self time	205 ms (0.5%)

97% Self time is shown in `computeDeep()`? 2% in `computeSimple()`?
(Wrong answer)

Profiling Accuracy: Hot Functions (NetBeans Profiler)

Hot Spots - Method	Self Time
InliningTest.**computeDeep** ()	39,344 ... (97.1%)
InliningTest.**computeSimple** ()	876 ms (2.2%)
InliningTest.**main** (String[])	205 ms (0.5%)
InliningTest.**nested_level_1** ()	104 ms (0.3%)

97 % Self Time in `computeDeep()`? `inner_sum()` is missing?
(Same wrong answer as in Calltree)

Profiling Accuracy: Calltree (Java Flight-Recorder)

Stack Trace	Sample Count	Percentage
InliningTest.main(String[])	158	100.00%
InliningTest.computeSimple()	157	99.37%
InliningTest.computeDeep()	1	0.63%
InliningTest.nested_level_1()	1	0.63%
InliningTest.nested_level_2()	1	0.63%

99% Samples shown in `computeSimple()`; `inner_sum()` is missing?
(Wrong answer)

Profiling Accuracy: Calltree (Java Flight-Recorder)
Run with `-XX:+DebugNonSafepoints` setting

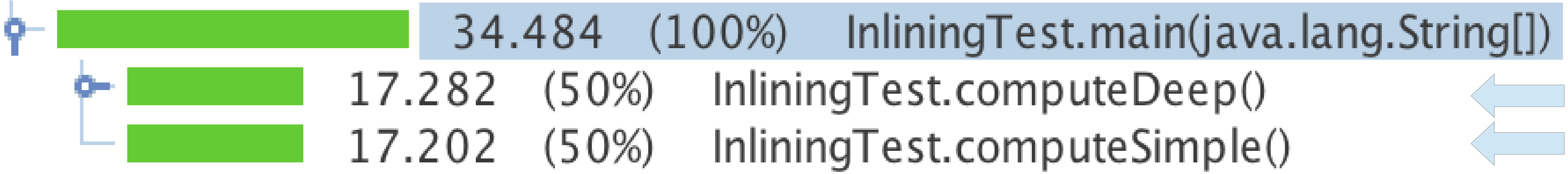
Stack Trace	Sample Count	Percentage
InliningTest.main(String[])	3,047	100.00%
InliningTest.computeSimple()	1,514	49.69%
InliningTest.computeDeep()	1,533	50.31%
InliningTest.nested_level_1()	1,533	50.31%
InliningTest.nested_level_2()	1,532	50.28%
InliningTest.innerSum()	1,500	49.23%

With a non-default setting, JFR gives the right answer

CPU time is <u>not</u> provided; hard to tell if calltree is complete

JavaOne ORACLE

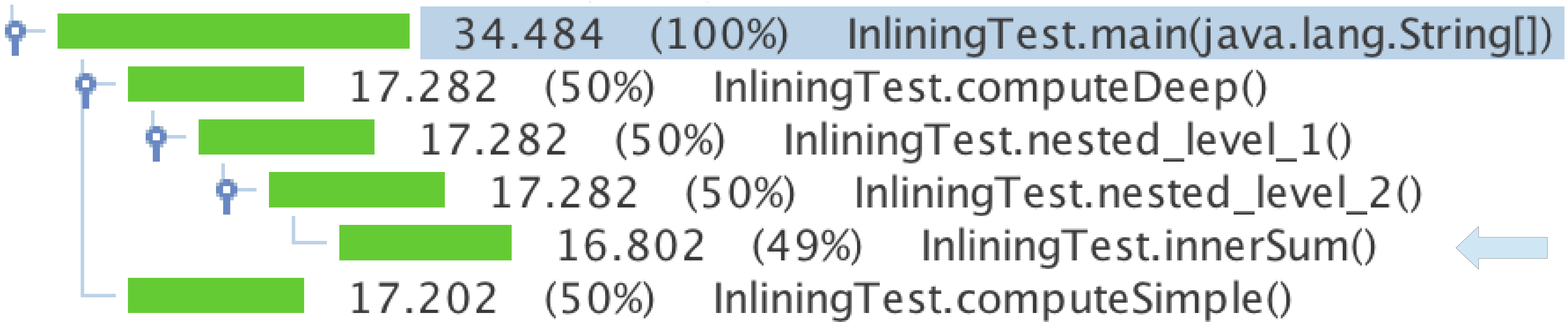
Profiling Accuracy: Calltree (Performance Analyzer)



Compute branches have equal time despite nesting in `computeDeep()`
Performance is the same thanks to JIT compilation

Run time of 34.5 seconds is about the same as run time without data collection

Profiling Accuracy: Expanded Calltree (Performance Analyzer)



Expansion shows time in `computeDeep()` comes from `innerSum()`

Profiling Accuracy: Hot Functions (Performance Analyzer)

Name	Excl. Total CPU (sec.)	Excl. Total CPU (%)
<Total>	34.484	100.00
InliningTest.computeSimple()	17.202	49.88
InliningTest.innerSum()	16.802	48.72
InliningTest.nested_level_2()	0.480	1.39
InliningTest.computeDeep()	0.	0.
InliningTest.main(java.lang.String[])	0.	0.
InliningTest.nested_level_1()	0.	0.

Exclusive Total CPU time (Self time) is split evenly between `innerSum()` and `computeSimple()`

Program Agenda

- Introduction: What is a Performance Problem
- Performance Analysis Tools
- Profiling Accuracy: a Simple Example
- Understanding Java Execution
- Identifying Programming Inefficiencies

Understanding Java Execution

- JVM execution is complicated
  - Some code is executed by Interpreter
  - Some code is dynamically constructed into JVM's data space
    - HotSpot-compiled methods; the Interpreter
  - Application has two callstacks: Java callstack and Native callstack
    - Synthesis of the two is required to give a true callstack in the user's model
  - JVM consumes resources
    - Garbage Collection: induces stalls in user Java code execution
    - Many threads on CPU (User and JVM System threads)
  - Safepoints during execution can induce sampling bias

Profiler must deal with all these complications

The Studio Performance Analyzer

- A set of tools for collecting and examining performance data
- Runs on Linux and Solaris, x86 and SPARC
- Collects data on a wide variety of applications
  - Supports C, C++, Fortran, Java, OpenMP, MPI
  - Can deal with complex enterprise-scale applications
- GUI and command-line interfaces
  - Many views of the data are available
  - Drill down by filtering in each view
    - Timeline, Processes, Threads, Functions, Memory Pages, ...

Available with free download and use-license

Performance Analyzer Goals

- Collect accurate data
  - Use internal JVM interfaces, captures inlined methods
  - Avoid sampling-bias with respect to safepoints
- Show Java source-level abstraction and hardware-level execution
  - Show source code line-level metrics
  - Allow seamless navigation between Java and JNI/Native code
  - Present data at the bytecode level for user Java
  - Present data at machine-code level for HotSpot-compiled methods and the JVM
- Expose internal JVM activity that uses resources
  - HotSpot compilation, Garbage collection, *etc.*

Java Source-Level Visualization (“User-mode”)

- Show user-Java threads only
  - Show complete Java callstack
    - Show all Java methods, even methods inlined by JIT
    - Show native portion of mixed Java/Native JNI callstacks
  - For Disassembly of Java, show bytecode
- Account for cases where Java stack cannot be unwound by JVM
  - Typically happens less than 3% of the time
  - Time attributed to `<no Java callstack recorded>`
  - Preferable to throwing away or deferring a sample
- Account for time in JVM runtime as `<JVM-System>`

Machine-Level Visualization ("Machine-mode")

Show what really happened...

- Show all versions of JIT compiled methods separately
  - Show `Interpreter` for interpreted methods
- Show all threads, both user-Java and JVM-internal
  - System threads identified by functions in callstack
    - Garbage collector
    - HotSpot compiler
- For Disassembly, show machine code

Understanding Java Execution: Source View, I

```
 (sec.)     (%)     Load Object: InliningTest.class (found as          experiments.demos/1509
 0.         0.      public class InliningTest {
                    <Function: InliningTest.<init>()>
                        public static void main(String[] args) {
 0.         0.              InliningTest test = new InliningTest();
                    <Function: InliningTest.main(java.lang.String[])>
 0.         0.              for (int i = 0; i < 20; i++) {
17.202     49.50                test.computeSimple();
17.282     49.73                test.computeDeep();
                            }
 0.         0.          }
                    // - computeSimple() - Computes a sum in a triply nested loop
 0.         0.          public double sum = 0.0;
                        public void computeSimple() {
 0.         0.              for (int i = 0; i < 1000; i++) {                    // outer
                    <Function: InliningTest.computeSimple()>
 0.         0.                  for (int j = 0; j < 1000; j++) {                // middle
 0.300      0.86                    for (int k = 0; k < 1000; k++) {            // inner
16.902     48.63                        sum += 1.0;            // ~48% of time // sum
                                    }
                                }
                            }
 0.         0.          }
```

Equal time spent in `computeSimple()` and `computeDeep()`

Hot line in `computeSimple()` shown

Understanding Java Execution: Source View, II

(sec.)	(%)	Load Object: InliningTest.class (found as experiments.demos/1509
		21. // – computeDeep() – Computes the same sum, but implements the
		22. // outer, middle, and inner loops as nested methods.
0.	0.	23. public double sum_deep = 0.0;
		24. public void computeDeep() {
17.282	49.73	25. for (int i = 0; i < 1000; i++) { nested_level_1(); } // outer
		<Function: InliningTest.computeDeep()>
0.	0.	26. }
		27. private void nested_level_1() {
17.282	49.73	28. for (int i = 0; i < 1000; i++) { nested_level_2(); } // middle
		<Function: InliningTest.nested_level_1()>
0.	0.	29. }
		30. private void nested_level_2() {
17.282	49.73	31. for (int i = 0; i < 1000; i++) { innerSum(); } // inner
		<Function: InliningTest.nested_level_2()>
0.	0.	32. }
		33. private void innerSum() {
16.802	48.34	34. sum_deep += 1.0; // ~48% of time // sum
		<Function: InliningTest.innerSum()>
0.	0.	35. }
		36. }

Correct data on each source line

Understanding Java Execution: Bytecode

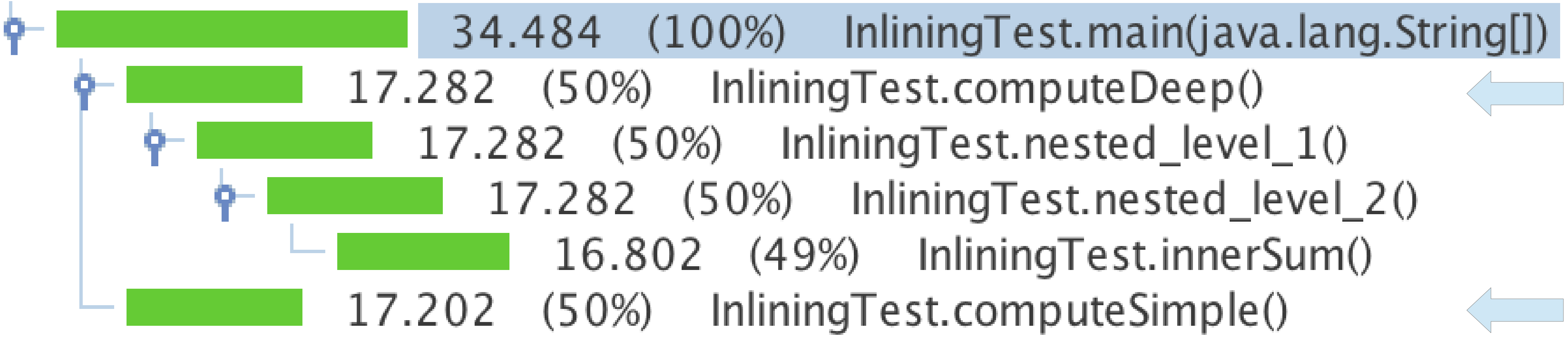
(sec.)	(%)	Load Object: InliningTest.class (found as experiments.demos/1509.
		33. private void innerSum() {
		34. sum_deep += 1.0; // ~48% of time // sum
		<Function: InliningTest.innerSum()>
0.	0.	[34] 00000000: aload_0
0.	0.	[34] 00000001: dup
0.030	0.00	[34] 00000002: getfield #3
0.	0.	[34] 00000005: dconst_1
15.621	0.00	[34] 00000006: dadd
1.151	0.00	[34] 00000007: putfield #3
		35. }
0.	0.	[35] 0000000a: return
		36. }

Attr. Total CPU (sec.)	(%)	InliningTest.innerSum() is called by
16.802	100.00	InliningTest.nested_level_2()

Most time on a single bytecode -- `dadd`

Called-by shows `innerSum()` called from `nested_level_2()`

Understanding Java Execution: Calltree, User-mode

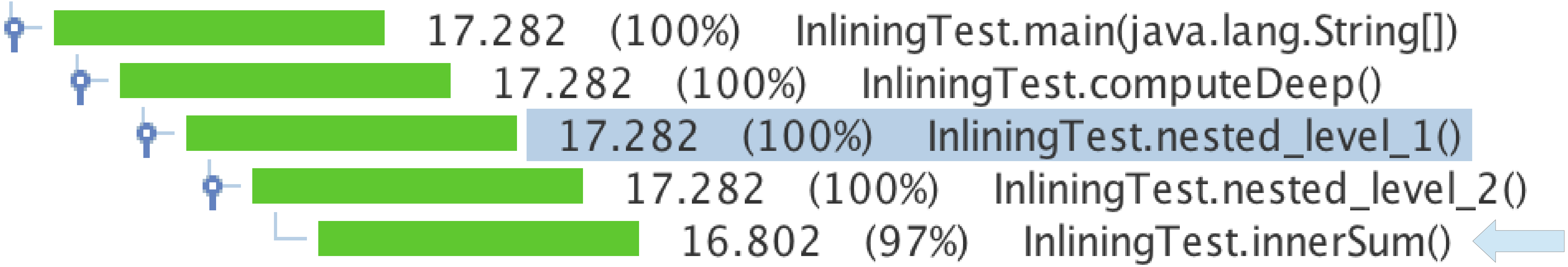


34.48 seconds in `main()` comes from two branches:

17.28 seconds from `computeDeep()`

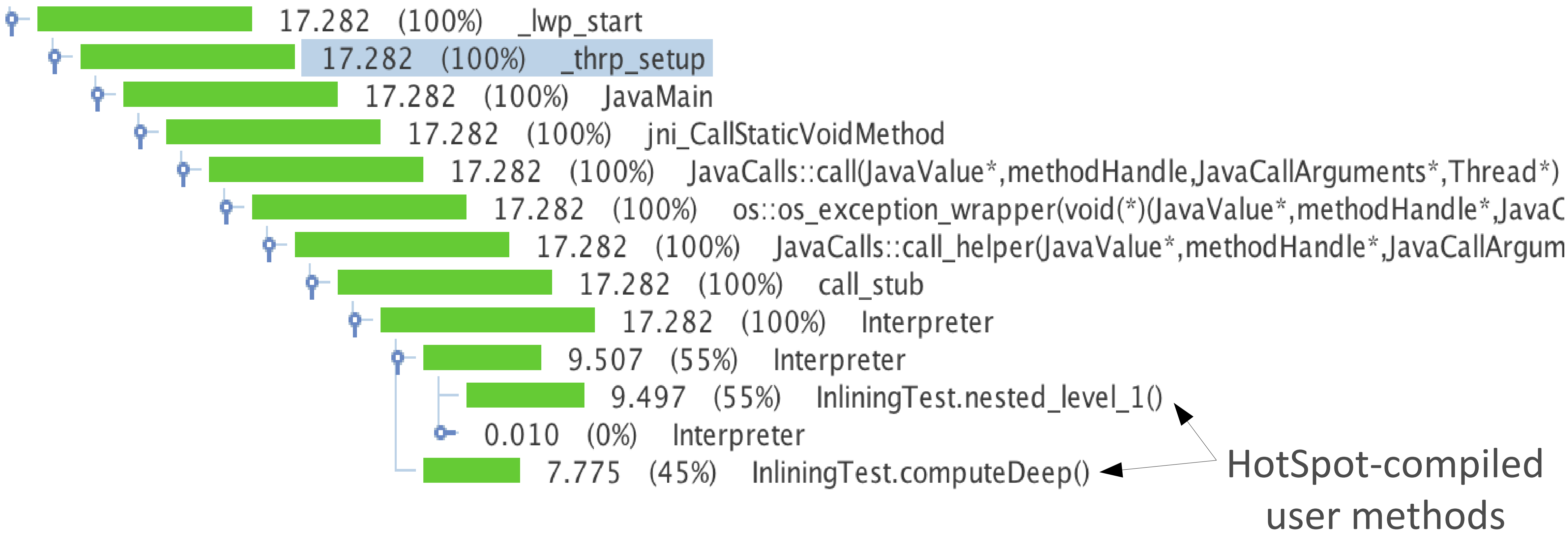
17.20 seconds from `computeSimple()`

Understanding Java Execution: Calltree, User-mode, Filtered



Calltree after filtering to include only callstacks containing `computeDeep()`

Understanding Java Execution: Calltree, Machine Mode, Filtered

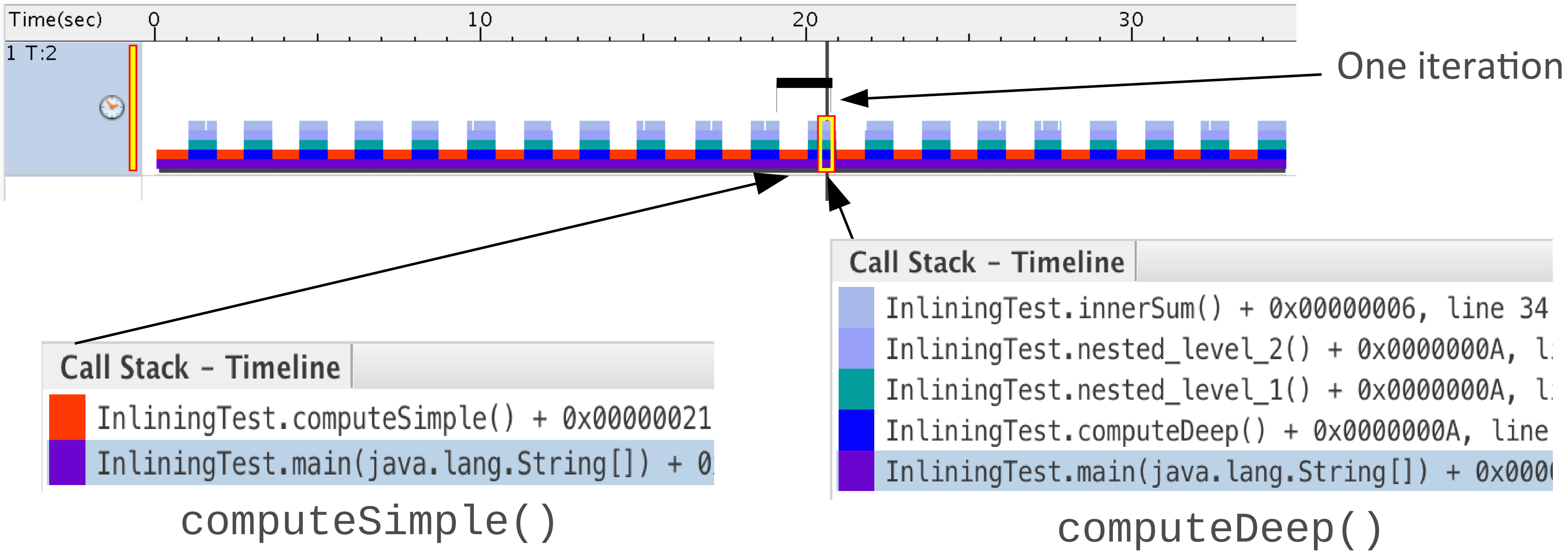


Machine-mode version of previous calltree: what the machine really executed

Only two frames are user methods

Others are in the JVM, from `libc.so`, or in the Java Interpreter

Understanding Java Execution: User-mode Timeline

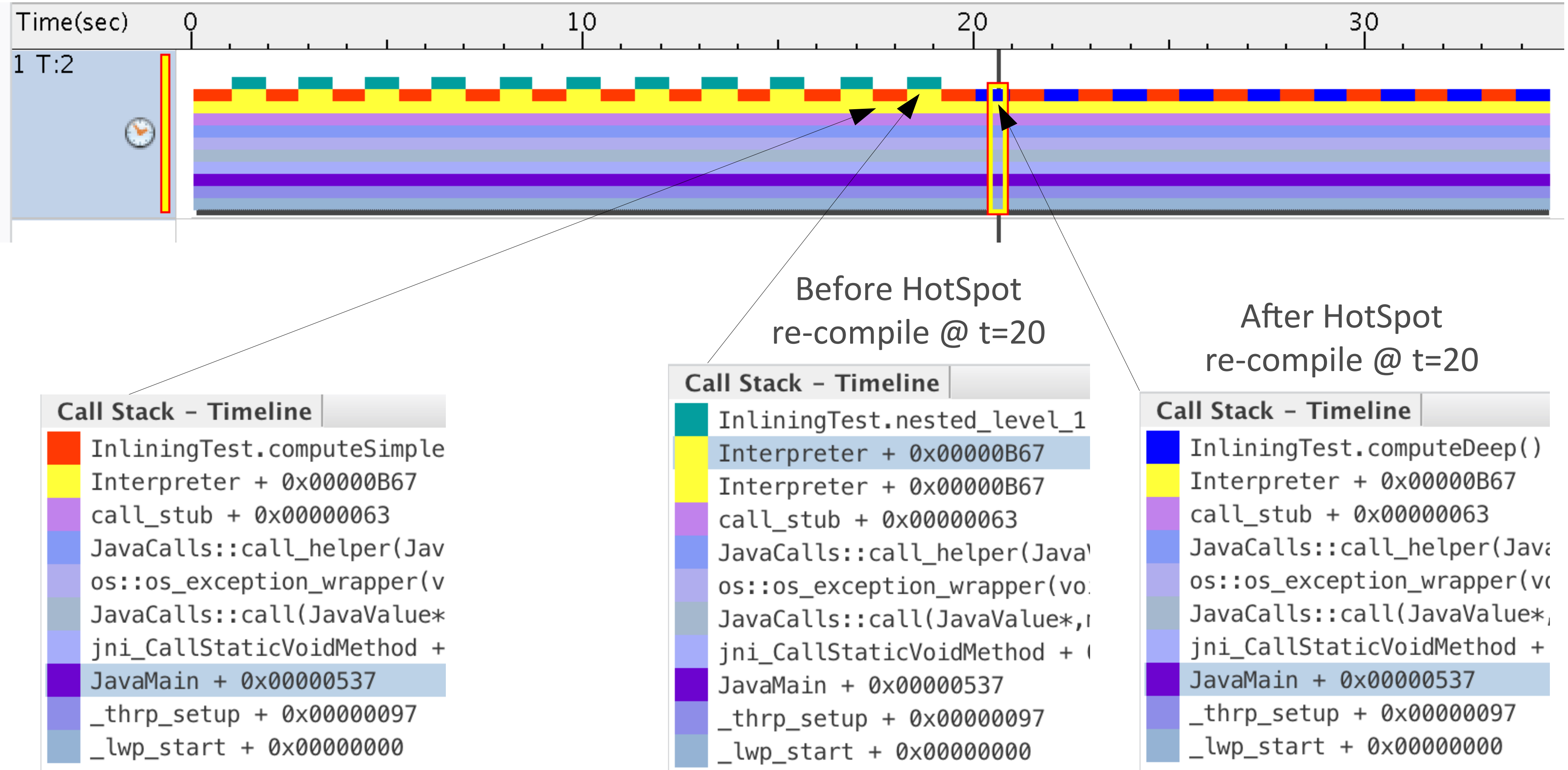


Callstacks shown on timeline

Name of method in each frame color-coded in representation

Twenty iterations of loop in `main()` shown

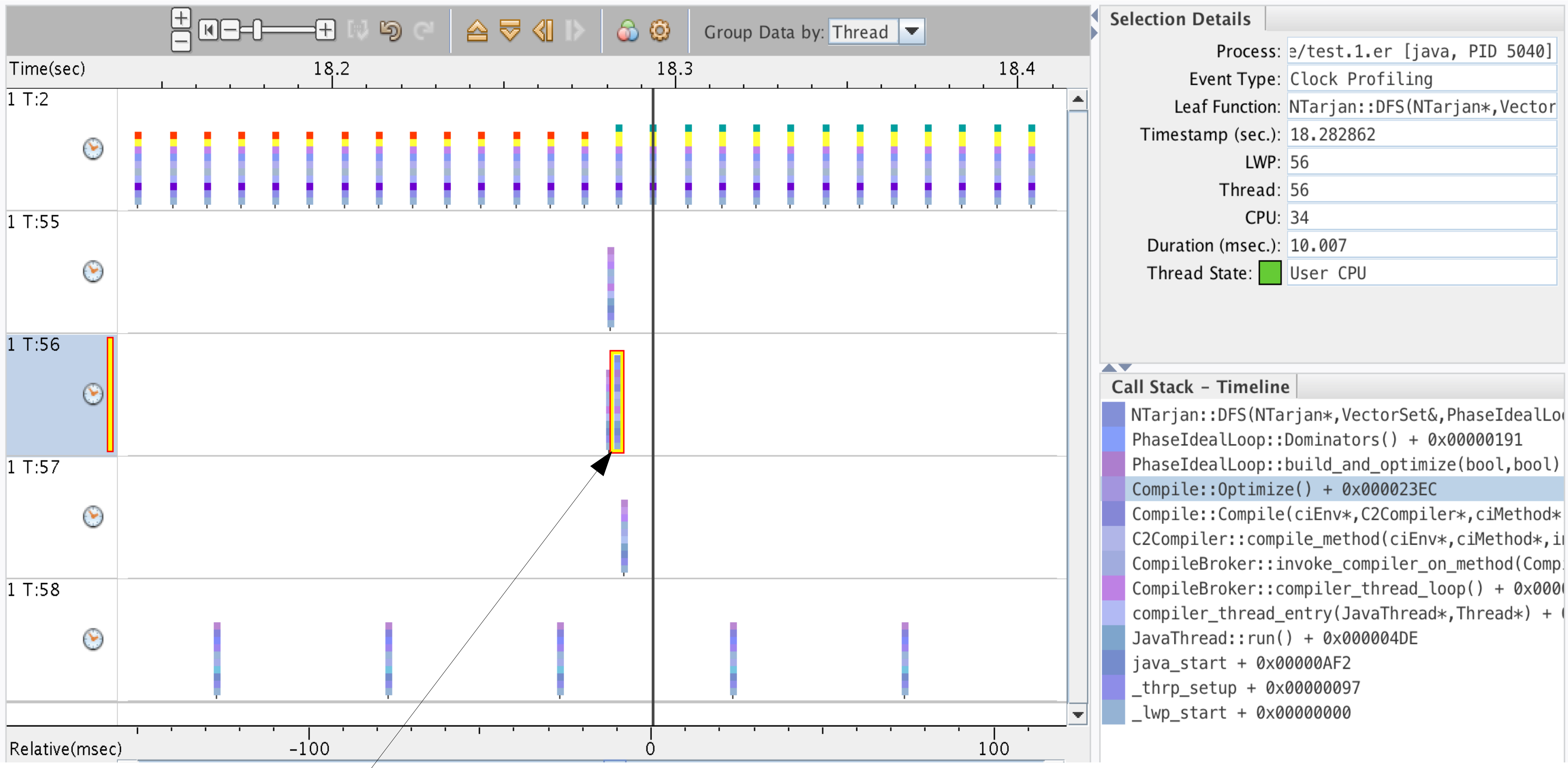
Understanding Java Execution: Machine-mode Timeline



`computeSimple()`

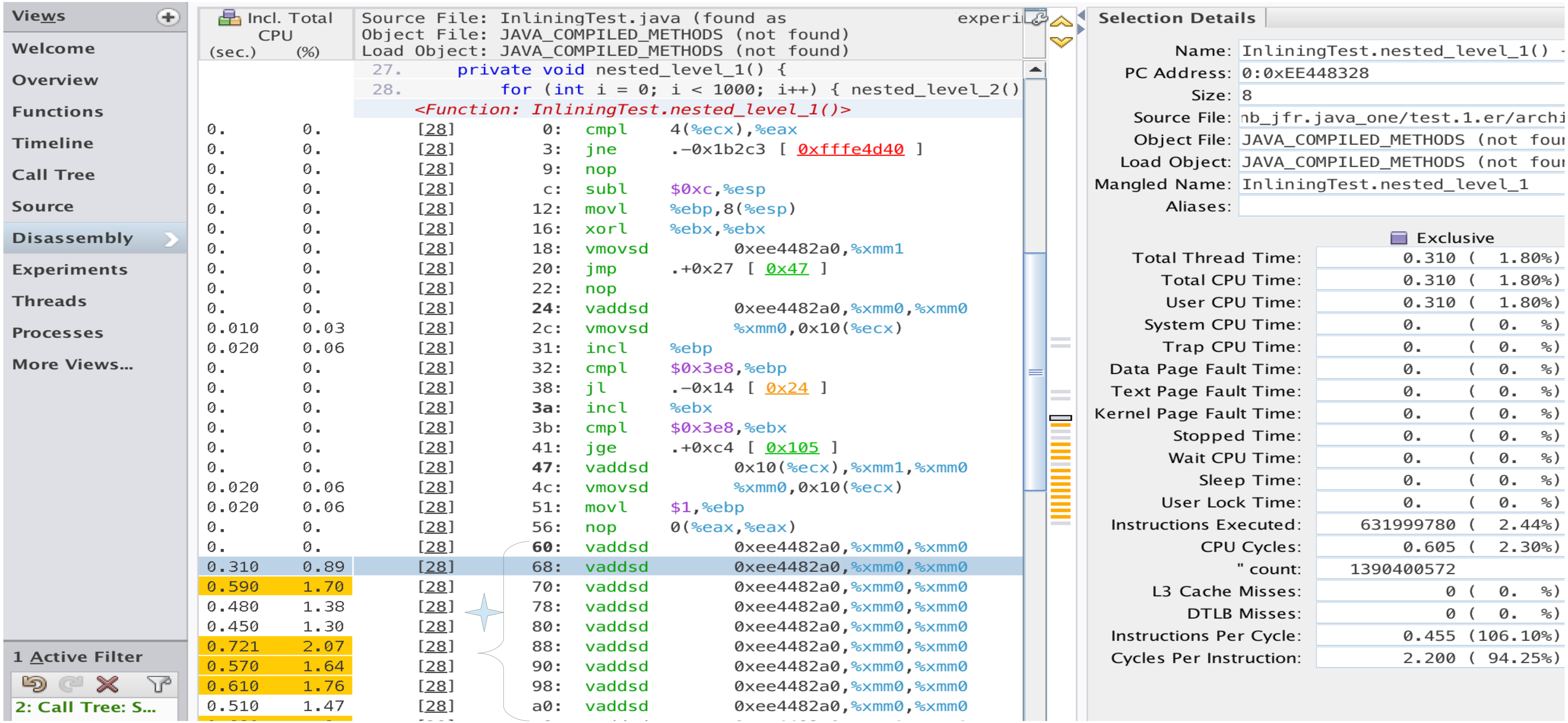
`ComputeDeep()`
`(two machine-stack variants)`

Understanding Java Execution: HotSpot Compilation



Selected event shows when HotSpot re-compile was triggered

Understanding Java Execution: Machine Mode Disassembly



Inner loop has been inlined and unrolled

Program Agenda

- Introduction: What is a Performance Problem
- Performance Analysis Tools
- Profiling Accuracy: a Simple Example
- Understanding Java Execution
- Identifying Programming Inefficiencies

Workflow for Identifying Inefficiencies

- Pick a repeatable test case for measurements
- Find the CPU hot-spots: collect performance data
  - What are the hot methods, call-paths, source-lines, instructions?
  - If the CPU pipeline is stalled:
    - If waiting for data: TLB misses, cache misses, memory latency ==> Memory access issues
    - If waiting for instruction completion: floating-point ops, divide, ...?
  - If CPU pipeline is not stalled:
    - If there are high call counts, there is too much overhead in call/return
    - If there is non-productive CPU time, *e.g.*, busy wait on a lock, there is wasted computation
- Fix the performance problems revealed in the data
  - Strategies for improvement depend on the cause

JavaOne ORACLE

Finding CPU Pipeline Stalls

- Using HW counters
  - Counters that directly measure stalls
    - Direct (statistical) measurement of time lost
    - SPARC commit-stalls has exactly one instruction skid
      - Useful for memory stalls and other pipeline stalls
  - Computing instructions-per-cycle (IPC) or cycles-per-instruction (CPI)
    - Profile with both instruction-counter and cycle-counter
      - Available on many chips
    - Low IPC or high CPI indicate inefficient use of the machine pipeline
    - Difficult to use to estimate time lost
    - Noisy when measured at source-line or instruction level

Reducing CPU Stalls due to Memory Access

- Memory access stalls include cache-miss and memory latency times
- Stalls can be reduced by:
  - Increasing data density so hot data fits in caches
  - Reducing levels of indirection
  - Co-locating the hot fields of a structure on the same cache-line
  - Fetching data in order to exploit HW pre-fetching

CPU Tuning Example

- An older benchmark code (SPECjbb2005)
  - Note: source changes are not allowed for published benchmark results
- Models a three-tier business system
  - Random input models user transactions for the first tier
  - Application to be measured implements the middle tier
  - Java Collections used for the third tier
  - Benchmark does no disk or network I/O

CPU Tuning Example: Finding a Hot Spot

Total CPU Time		Stall Cycles Time	Lines
EXCLUSIVE sec.	INCLUSIVE sec.	EXCLUSIVE #	Function, line # in "sourcefile"
478.054	478.054	299.348	<Total>
42.029	42.510	40.267	*<Function: java.util.HashMap.getNode(int, java.l*
33.974	69.298	33.012	spec.jbb.Warehouse.retrieveStock(int), line 121
54.428	54.428	17.645	java.util.TreeMap.successor(java.util.TreeMap$Er
12.549	59.301	13.522	spec.jbb.CustomerReportTransaction.process(), li
11.618	11.618	10.456	java.util.TreeMap.successor(java.util.TreeMap$Er
10.677	10.677	10.256	*<Function: java.lang.String.length(), instructic*

Hottest source line

Hot spot in line 121 in method `retrieveStock()`

CPU Tuning Example: Hot Spot in Source

Total CPU Time INCLUSIVE sec.	spec/jbb/Warehouse.java
	119.
	120. public Stock retrieveStock(int inItemId) {
69.298	121. return (Stock) stockTable.get(inItemId);
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
	122. }
	123.

Hot spot in one-line access method

CPU Tuning Example: Hot Spot in Bytecode

Total CPU Time INCLUSIVE sec.	Stall Cycles Time INCLUSIVE #	spec/jbb/Warehouse.java
		119.
		120. public Stock retrieveStock(int inItemId) {
		121. return (Stock) stockTable.get(inItemId);
		<Function: spec.jbb.Warehouse.retrieveStock(int)>
0.	0.	[121] 00000000: aload_0
3.472	3.689	[121] 00000001: getfield #8
0.	0.	[121] 00000004: iload_1
0.751	0.189	[121] 00000005: invokestatic valueOf()
36.486	34.645	[121] 00000008: invokeinterface #29, 2) #29
28.590	27.789	[121] 0000000d: checkcast spec.jbb.Stock
0.	0.	[121] 00000010: areturn
		122. }
		123.

Total C... ATTRIBUTED sec.	spec.jbb.Warehouse.retrieveStock(int) calls
34.584	spec.jbb.MapDataStorage.get(java.lang.Object)
0.741	java.lang.Integer.valueOf(int)

35 stall seconds on `invokeinterface` bytecode; call goes to two places

Most time attributed to `MapDataStorage.get()` call

CPU Tuning Example: Hot Spot due to Memory Access

- Data structure being referenced is in `stockTable()`
  - Underlying data structure is `MapDataStorage()`
    - It is based on `HashMap()` class
    - Index is consecutive integers
- Choice of storage class can be critical for performance
  - In this particular case, replace `HashMap()` with `ArrayList()`
    - Optimization 1

CPU Tuning Example: Effect of Optimization 1

Before

Total CPU Time INCLUSIVE sec.	spec/jbb/Warehouse.java
	119.
	120. public Stock retrieveStock(int inItemId) {
69.298	121. return (Stock) stockTable.get(inItemId);
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
	122. }
	123.

After

Total CPU Time INCLUSIVE sec.	spec/jbb/Warehouse.java
	119.
	120. public Stock retrieveStock(int inItemId) {
45.512	121. return (Stock) stockTable.get(inItemId);
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
	122. }
	123.

Improvement of overall application time of ~18%

CPU Tuning Example: Type Casting

Total CPU Time INCLUSIVE sec.	spec/jbb/Warehouse.java
	119.
	120. public Stock retrieveStock(int inItemId) {
	121. return (Stock) stockTable.get(inItemId);
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
0.	[121] 00000000: aload_0
3.753	[121] 00000001: getfield #8
0.	[121] 00000004: iload_1
0.610	[121] 00000005: invokestatic valueOf()
2.522	[121] 00000008: invokeinterface #29, 2) #29
38.627	[121] 0000000d: checkcast spec.jbb.Stock
0.	[121] 00000010: areturn
	122. }
	123.

`checkcast` bytecode verifies a cast, taking ~39 seconds

Type-safety for Java Generics comes at a cost

Optimization 2: replace `stockTable.get(id)` with `stockTable[id]`

CPU Tuning Example: Effect of Optimization 2, I

Before

	120. public Stock retrieveStock(int inItemId) {
	121. return (Stock) stockTable.get(inItemId);
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
0.	[121] 00000000: aload_0
3.753	[121] 00000001: getfield #8
0.	[121] 00000004: iload_1
0.610	[121] 00000005: invokestatic valueOf()
2.522	[121] 00000008: invokeinterface #29, 2} #29
38.627	[121] 0000000d: checkcast spec.jbb.Stock
0.	[121] 00000010: areturn
	122. }

After

	123. public Stock retrieveStock(int inItemId) {
	124. return stockTable[inItemId];
	<Function: spec.jbb.Warehouse.retrieveStock(int)>
0.	[124] 00000000: aload_0
4.313	[124] 00000001: getfield #7
0.	[124] 00000004: iload_1
0.340	[124] 00000005: aaload
0.	[124] 00000006: areturn
	125. }

Time in `checkcast` is gone…

…but overall throughput did not change much!

JavaOne ORACLE

CPU Tuning Example: Effect of Optimization 2, II

Before

Total CPU Time INCLUSIVE sec.	Stall Cycles Time INCLUSIVE #	spec/jbb/DeliveryTransaction.java	
3.502	3.256	140.	`int itemId = orderline.getItemId();`
41.239	39.578	141.	`Stock stock = warehousePtr.retrieveStock(itemId);`
3.492	2.478	142.	`int availableQuantity = stock.getQuantity();`
0.140	0.067	143.	`if (availableQuantity >= requiredQuantity) {`
0.580	0.378	144.	`stock.changeQuantity(-requiredQuantity);`
0.	0.022	145.	`break;`
		146.	`}`

After

Total CPU Time INCLUSIVE sec.	Stall Cycles Time INCLUSIVE #	spec/jbb/DeliveryTransaction.java	
3.562	3.311	140.	`int itemId = orderline.getItemId();`
0.360	0.211	141.	`Stock stock = warehousePtr.retrieveStock(itemId);`
34.624	33.489	142.	`int availableQuantity = stock.getQuantity();`
0.050	0.011	143.	`if (availableQuantity >= requiredQuantity) {`
0.430	0.411	144.	`stock.changeQuantity(-requiredQuantity);`
0.	0.	145.	`break;`
		146.	`}`

Stall time has moved to caller line 142!

Stall was due to first-touch cache miss; stalls simply moved to the next access

Lessons from the CPU Tuning Example

- Using a more efficient storage class saved 18%
  - Replaced `HashMap()` with `ArrayList()`
- Eliminating an expensive type check did not help
  - Memory access cost moved, but did not go away
  - CPU hardware counters can be used to confirm cache misses
- Memory latency can dwarf other inefficiencies

- Understanding can not be achieved without good tools!

When to consider Studio Performance Analyzer

- General CPU-time performance tuning
  - Java and Native (C, C++, Fortran)
- To measure real behavior of production code, production-scale runs
- To drill-down with views and filtering
- For Linux and Solaris
  - Supports cross-platform Linux/Solaris and x86/SPARC analysis
  - Supports remote analysis with GUI on Windows and MacOS

Advantages of Performance Analyzer Approach

- Collects accurate data
  - Uses internal JVM interfaces, captures inlined methods
  - Avoids sampling-bias with respect to safepoints
- Shows Java source-level abstraction and hardware-level execution
  - Shows source code line-level metrics
  - Allows seamless navigation between Java and JNI/Native code
  - Presents data at the bytecode level for user Java
  - Presents data at machine-code level for HotSpot-compiled methods and the JVM
- Exposes internal JVM activity that uses resources
  - HotSpot compilation, Garbage collection, *etc.*

Available with free download and use-license

For More Information

- Studio Website
  - http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
- Community
  - http://www.oracle.com/technetwork/server-storage/solarisstudio/community/index.html
- Related Sessions
  - CON 8216: Inoculating Software, Boosting Quality: SAS & Oracle Experience with Application Data Integrity
    - Maureen Chew, Chandra Garud, and Sheldon Lobo
  - CON 8337: Developer Cloud Made Simple: How to Build an OpenStack Developer Cloud
    - Deepankar Bairagi, Liang Chen, and Nasser Nouri

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.