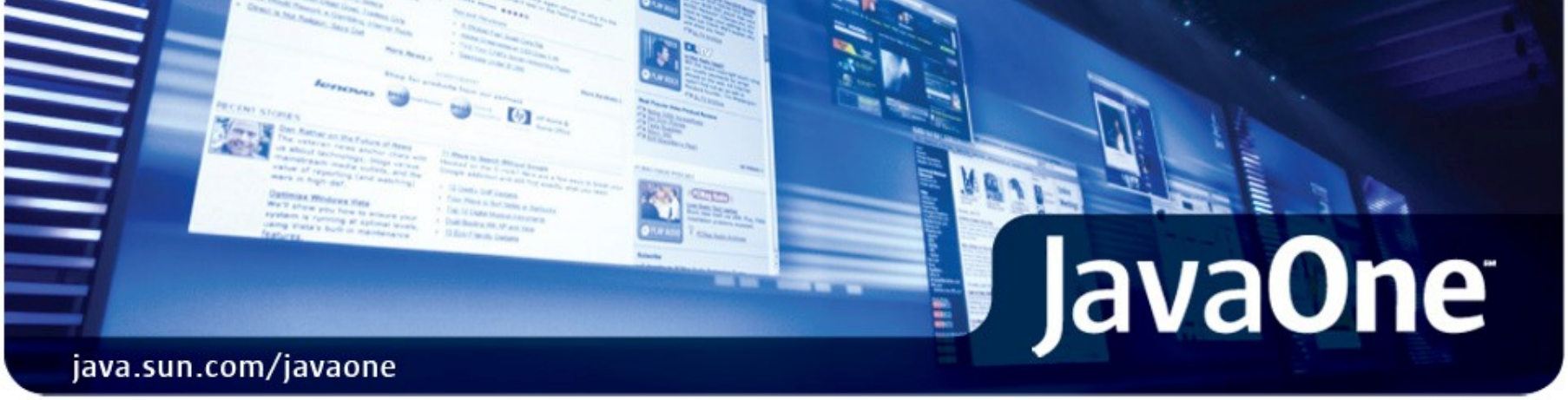


Java™ Platform Performance: Case Studies in Bottleneck Identification and Removal

Kumar Shiv, Intel
Paul Hohensee, Sun Microsystems

TS-6434

Sun microsystems

Java

Sun microsystems

Trademarks And Abbreviations
(to get them out of the way ...)

- Java™ Platform, Standard Edition (Java SE)
- Java HotSpot™ Virtual Machine (HotSpot JVM™)

Who Are These Guys?

- **Kumar Shiv**
  - Principal Performance Architect, Managed Runtimes
  - Intel, Hillsboro, OR
- **Paul Hohensee**
  - Senior Staff Engineer, Hotspot JVM Tech Lead
  - Sun Microsystems, Burlington, MA

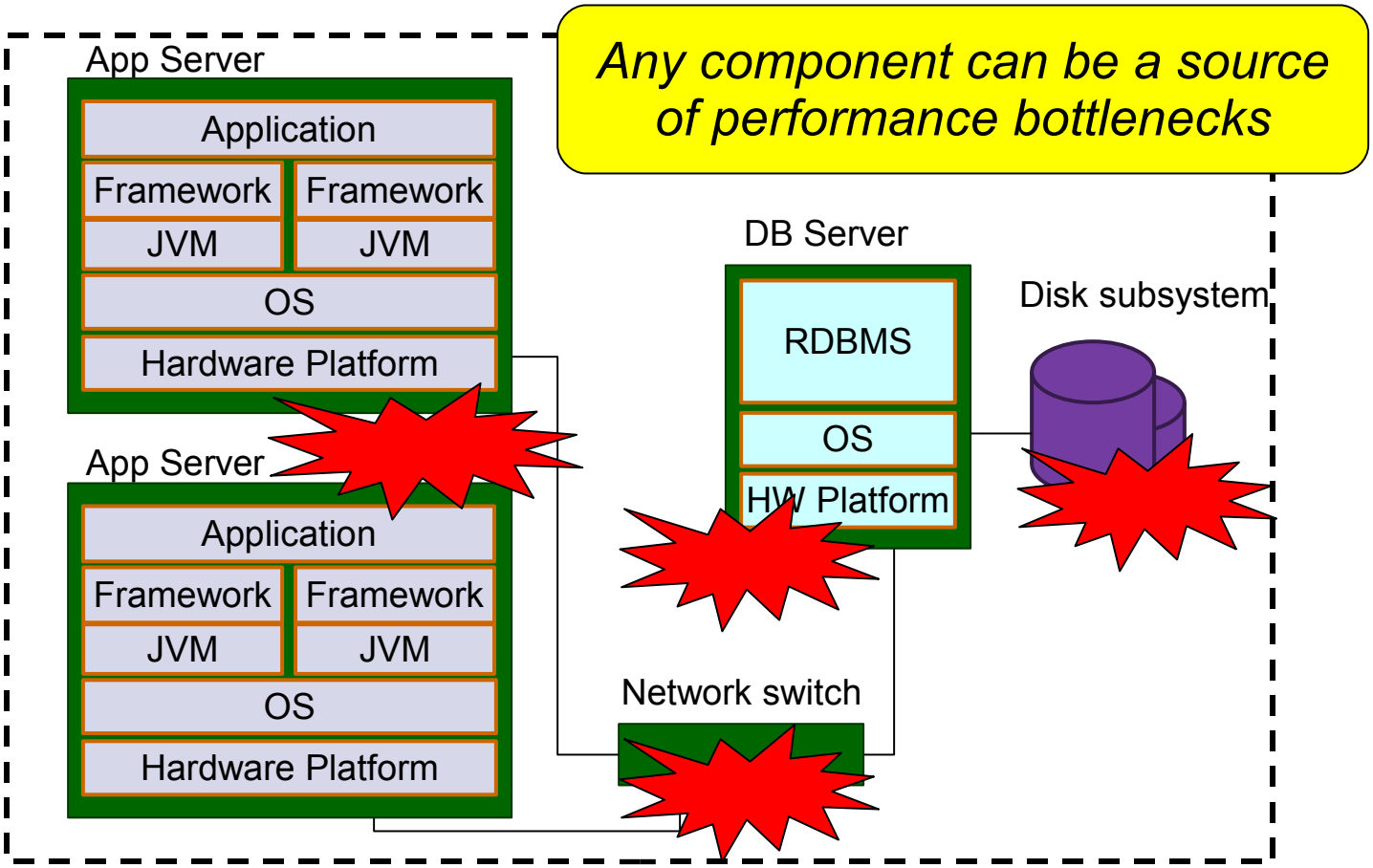
Learn how performance bottlenecks are found and fixed in application code, Java library code and the JVM machine.

GOAL

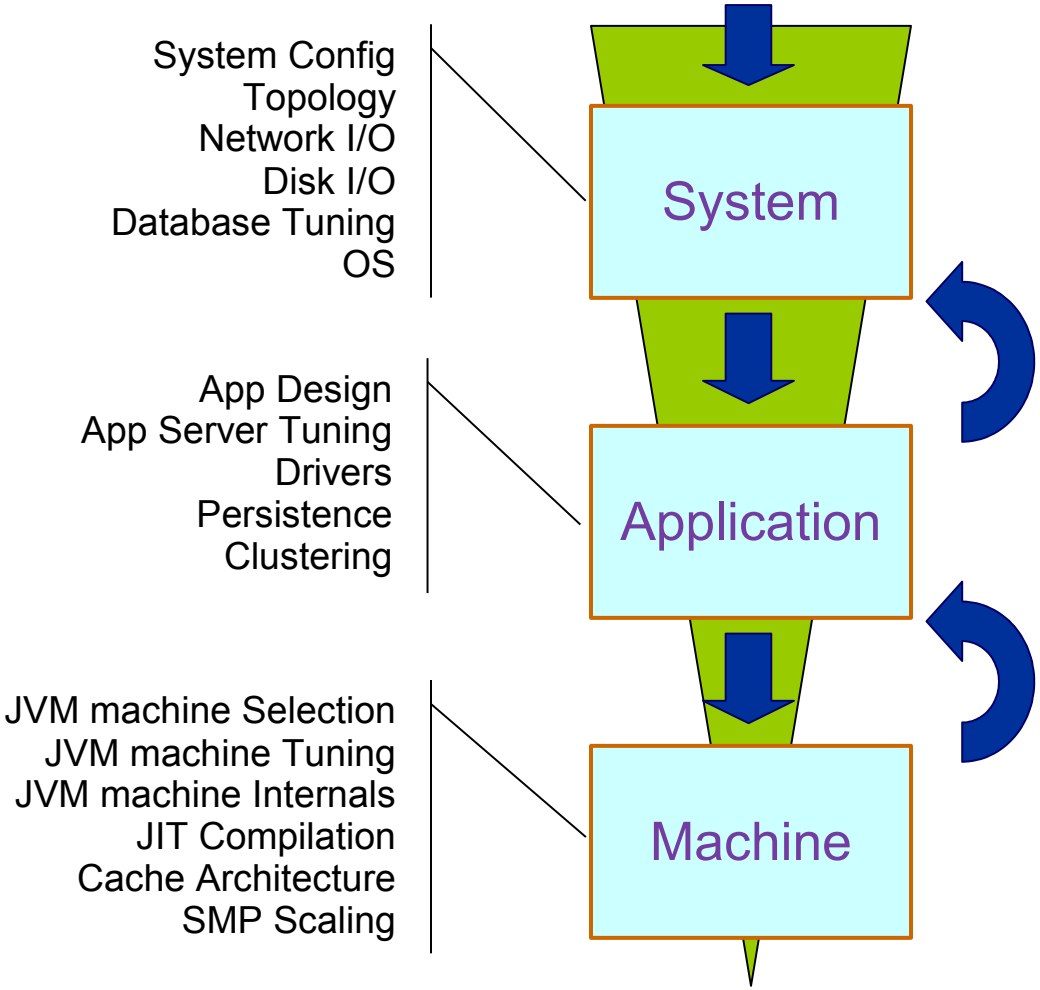
Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

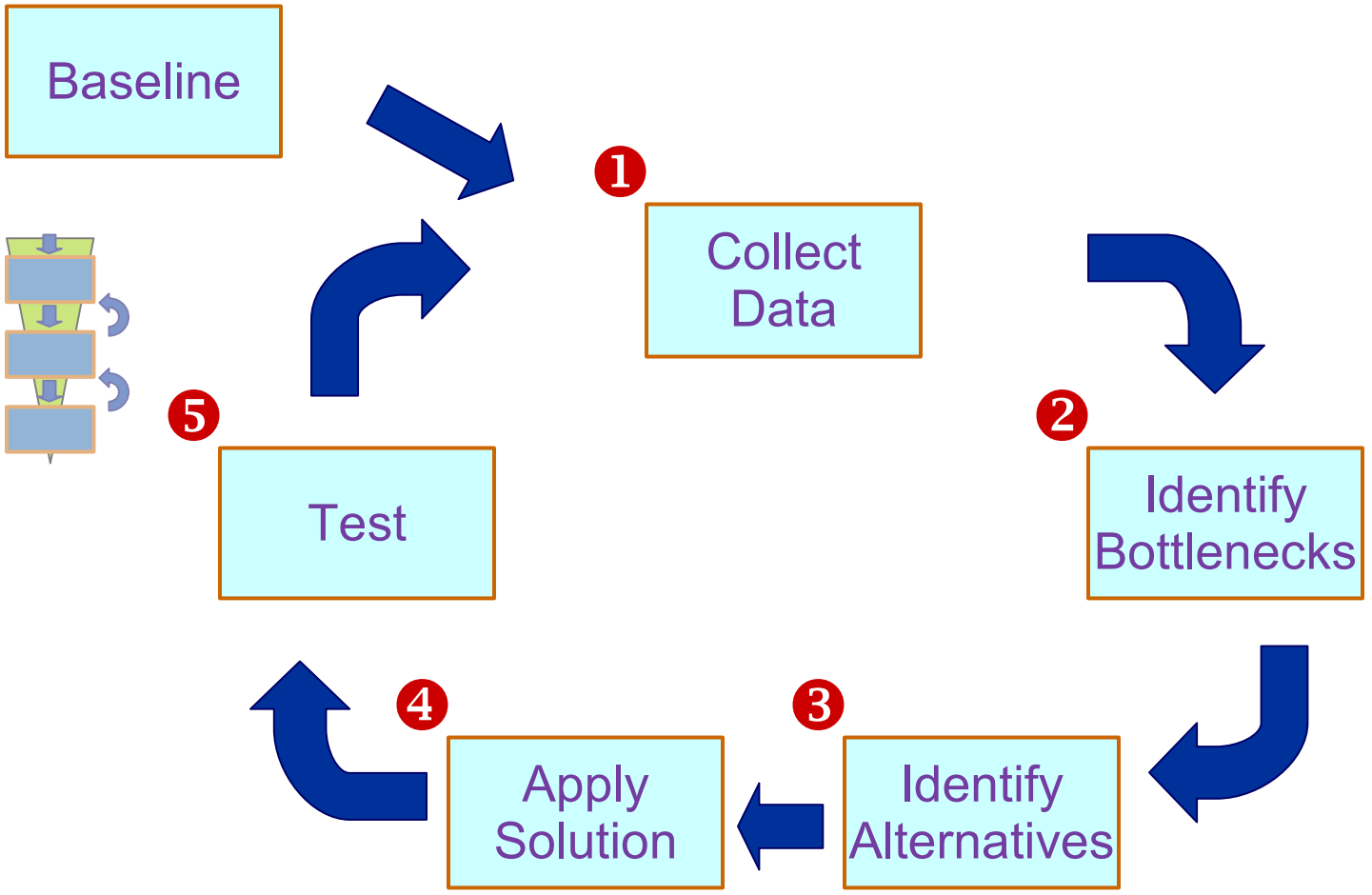
Multi-tier Solution Deployments Are Complex



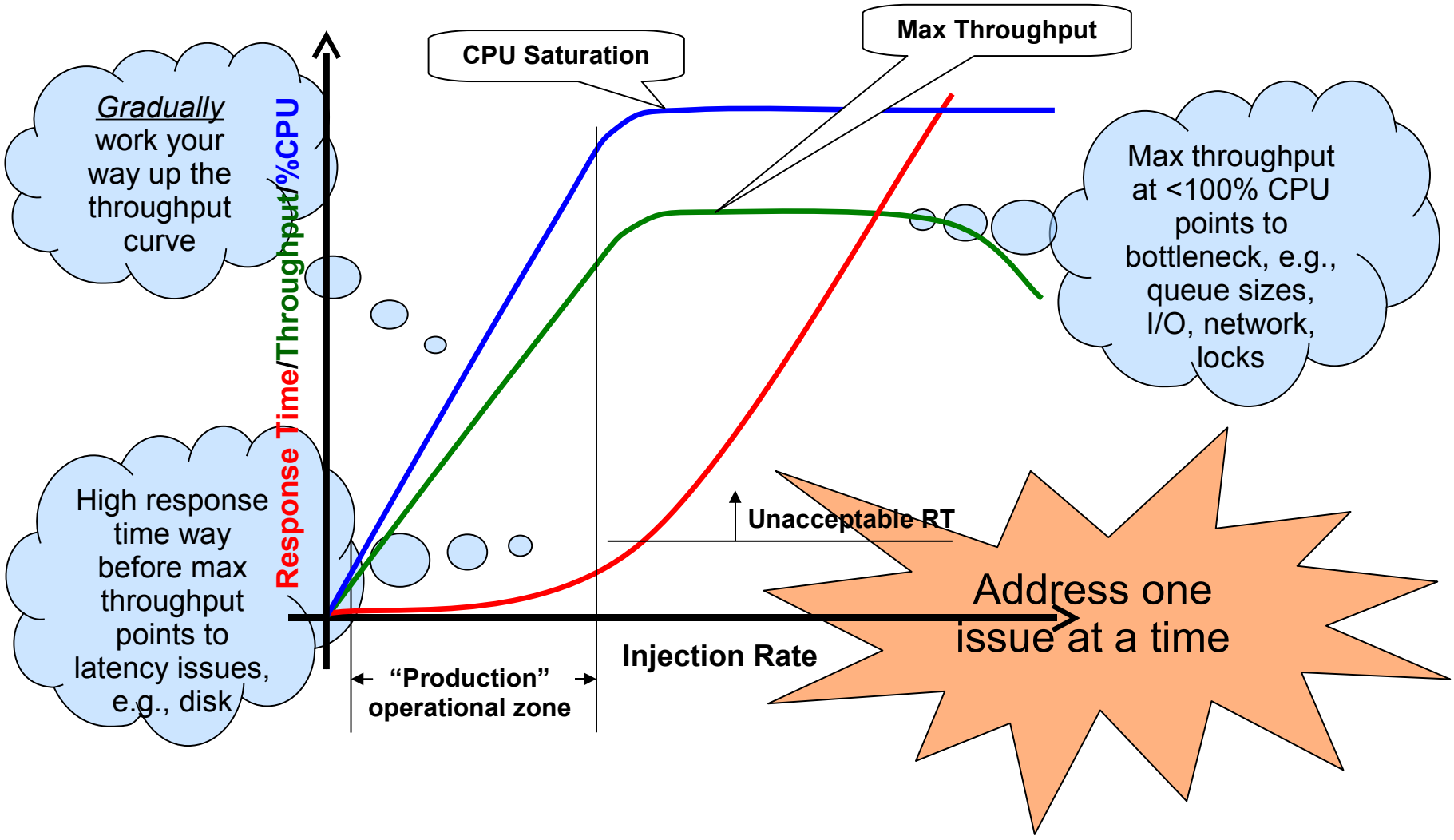
Top-Down Approach



Iterative Approach



Key Tool: A Throughput Curve



Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

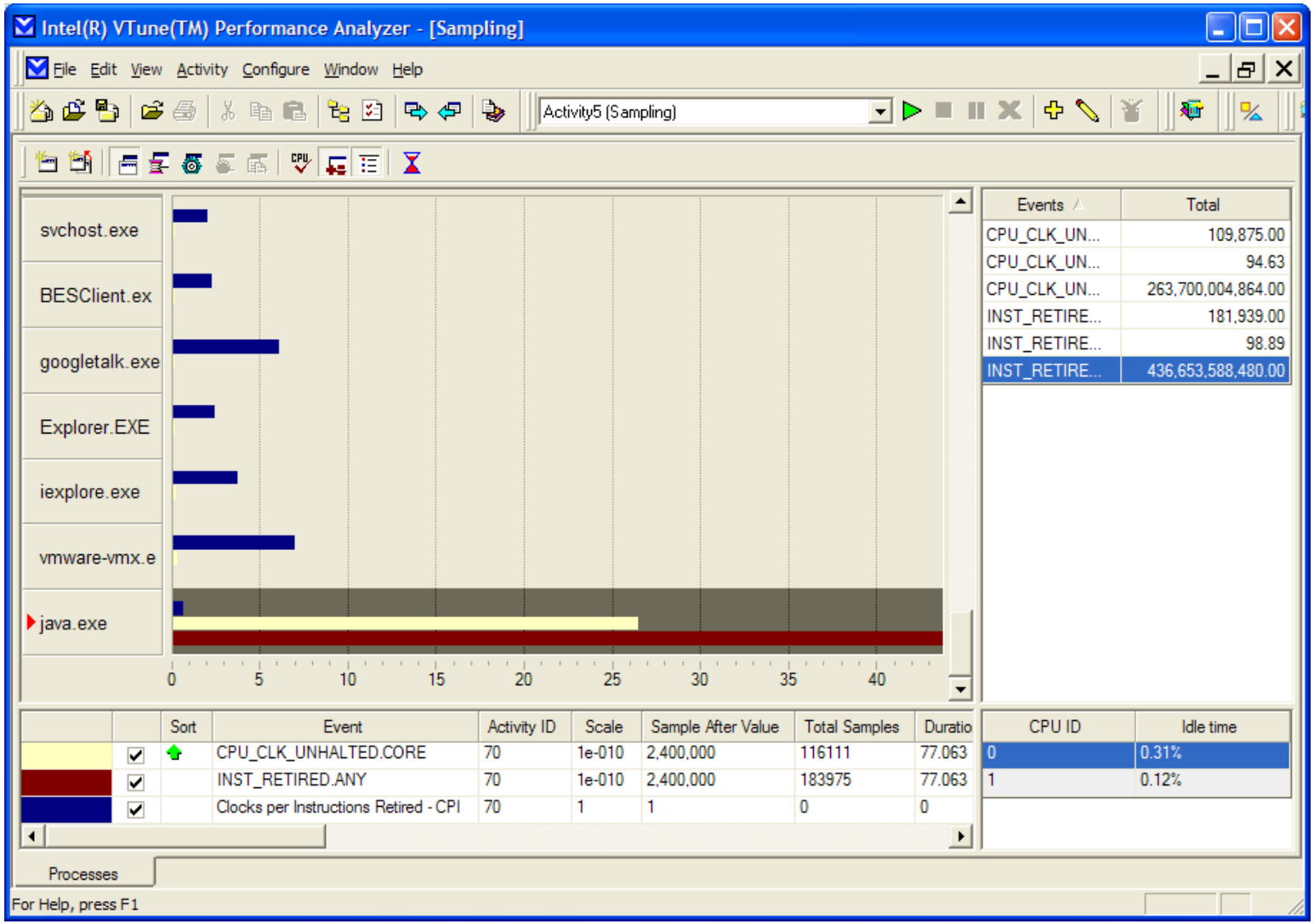
Tools, Tools, Tools

- SAR, iostat, other OS-supplied monitoring tools
- JDump and thread dumps
  - Often overlooked, easy to analyze with targeted scripts
- Dtrace: Solaris™ Operating System, OS X
  - http://www.sun.com/bigadmin/content/dtrace
- NetBeans™ software Profiler: Windows, Linux, Solaris, OS X
  - http://www.netbeans.org
- Project VisualVM: Windows, Linux, Solaris Operating System, Java HotSpot, JRockit™
  - https://visualvm.dev.java.net
- BEA™ Mission Control: Windows, Linux
  - http://dev2dev.bea.com/jrockit/tools.html
- Intel® VTune™ Performance Analyzer: Windows, Linux
  - http://www.intel.com/software/products/vtune
- Sun Studio Analyzer: Linux, Solaris operating system
  - http://developers.sun.com/sunstudio

VTune Performance Analyzer

- Helps identify and characterize performance issues by:
  - Collecting performance data from the system running your application
  - Organizing and displaying the data in variety of interactive views from system-wide down to source code or processor instruction perspective
  - Identifying potential performance issues and suggesting improvements
- Uses profiling APIs exposed by JVM machines
- Can profile pure and mixed managed and unmanaged code

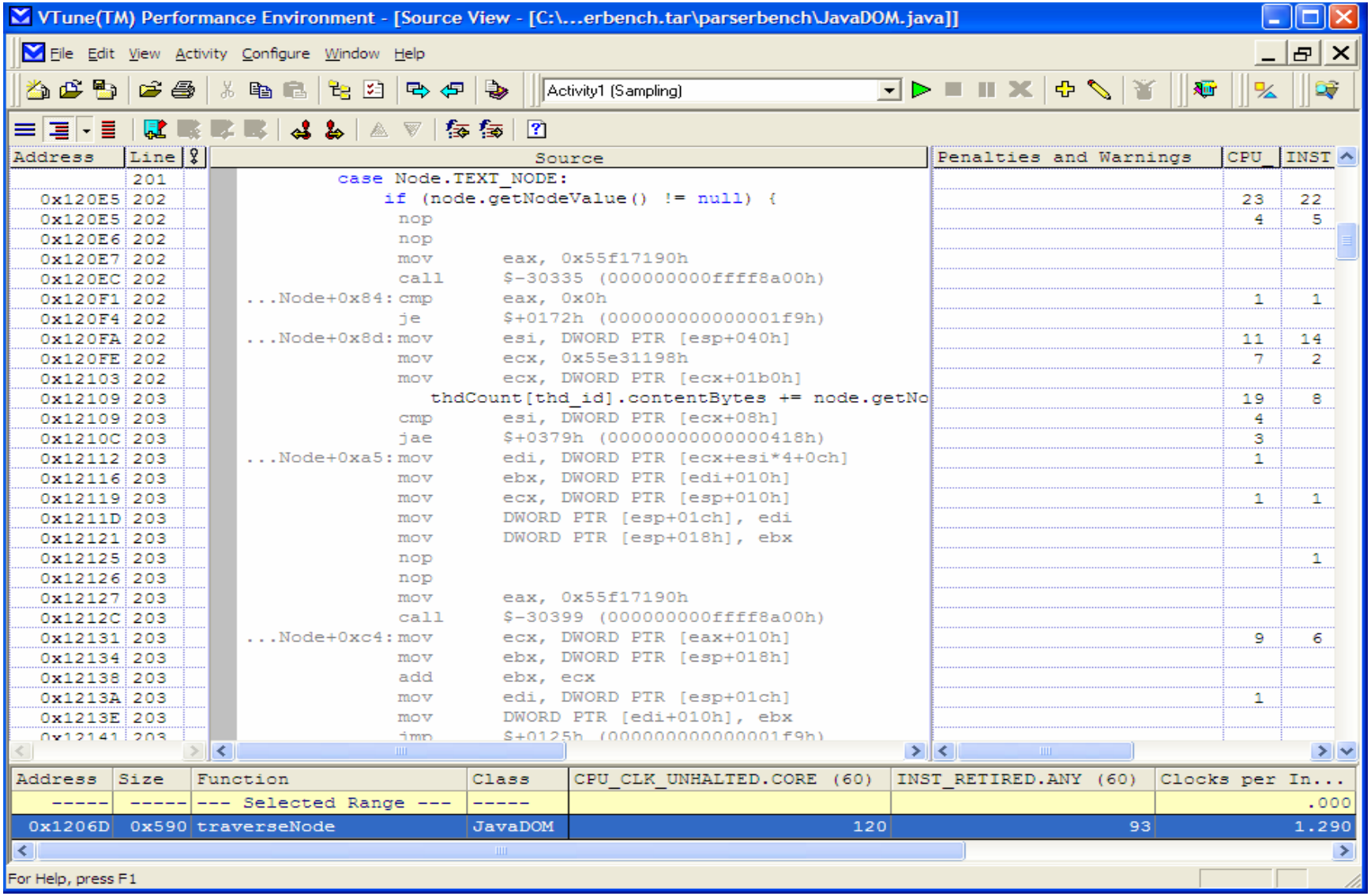
VTune: Process View, XML Parser



VTune: Sampling

- Occasionally interrupts the processor
  - Periodic: typically 1000 times per second
  - Event-Based: Triggered by the occurrence of a user definable number of processor events
- Collects execution context
  - Execution memory address (Instruction Pointer)
  - OS process and thread ID
  - Executable module located at that address
  - If you have symbols for the module, post processing can identify the function or method at the memory address
  - Line numbers from symbol file can direct you to the relevant line of source code
- System wide

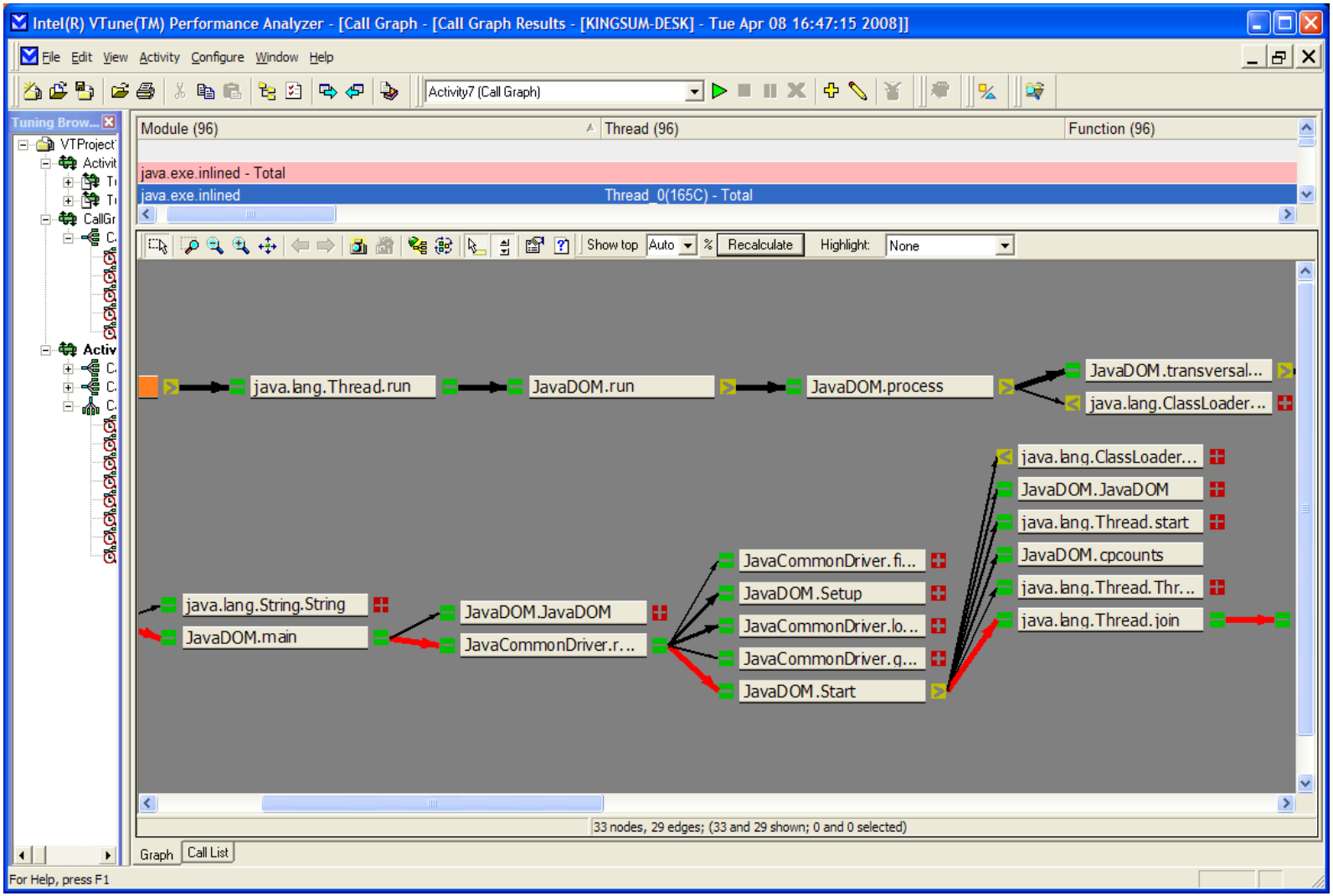
VTune: JIT'ed Code, XML Parser, traverseNode



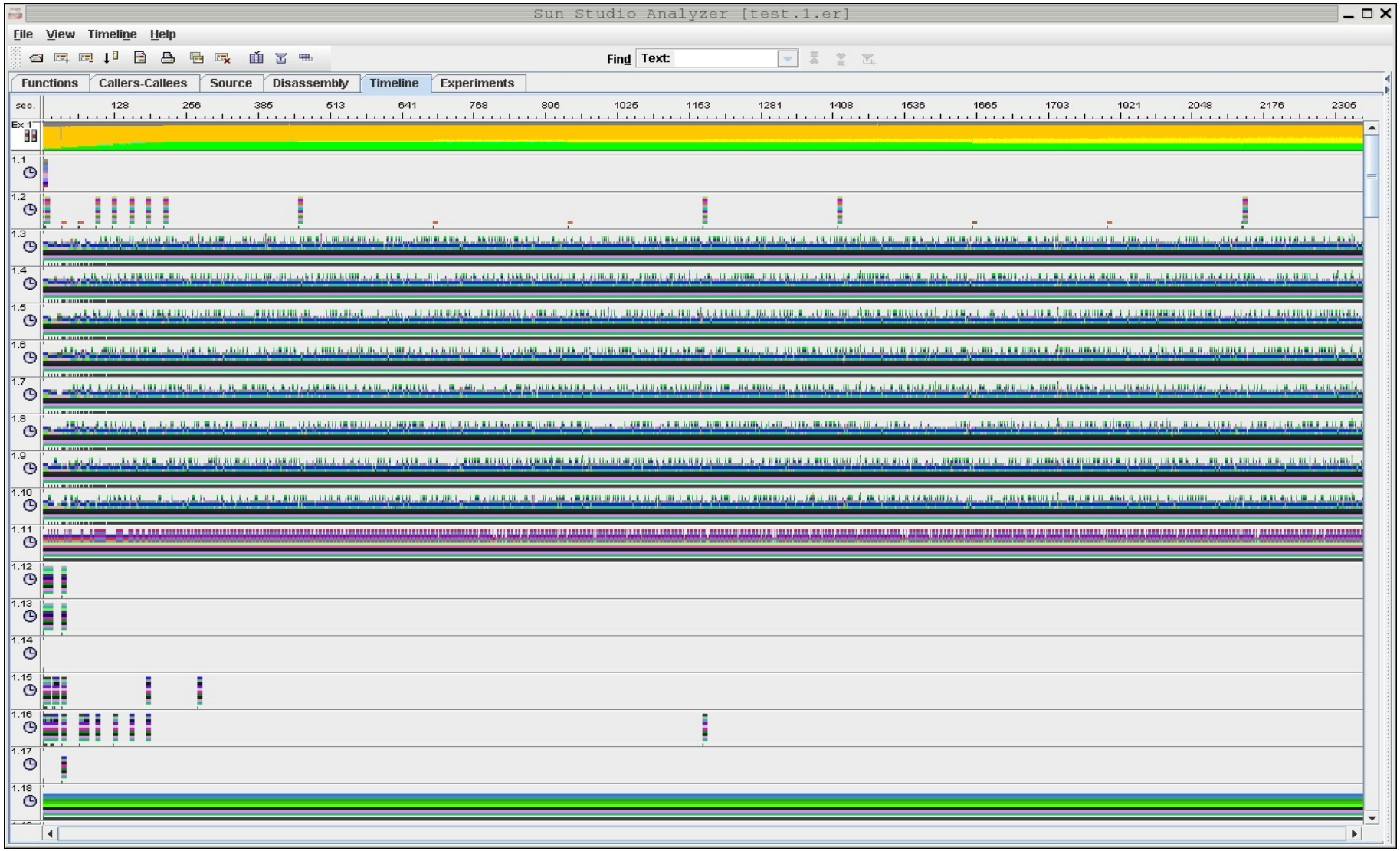
VTune: Call Graph

- Tracks method entry and exit points in your code
  - Periodic: typically 1000 times per second
  - Event-Based: Triggered by the occurrence of a user definable number of processor events
- Displays program flow, critical functions and call sequences
- Can use instrumentation and JVM machine profiling APIs
  - Mixed Mode – shows performance data for Java code and underlying OS APIs
  - Pure – shows performance data only for Java code

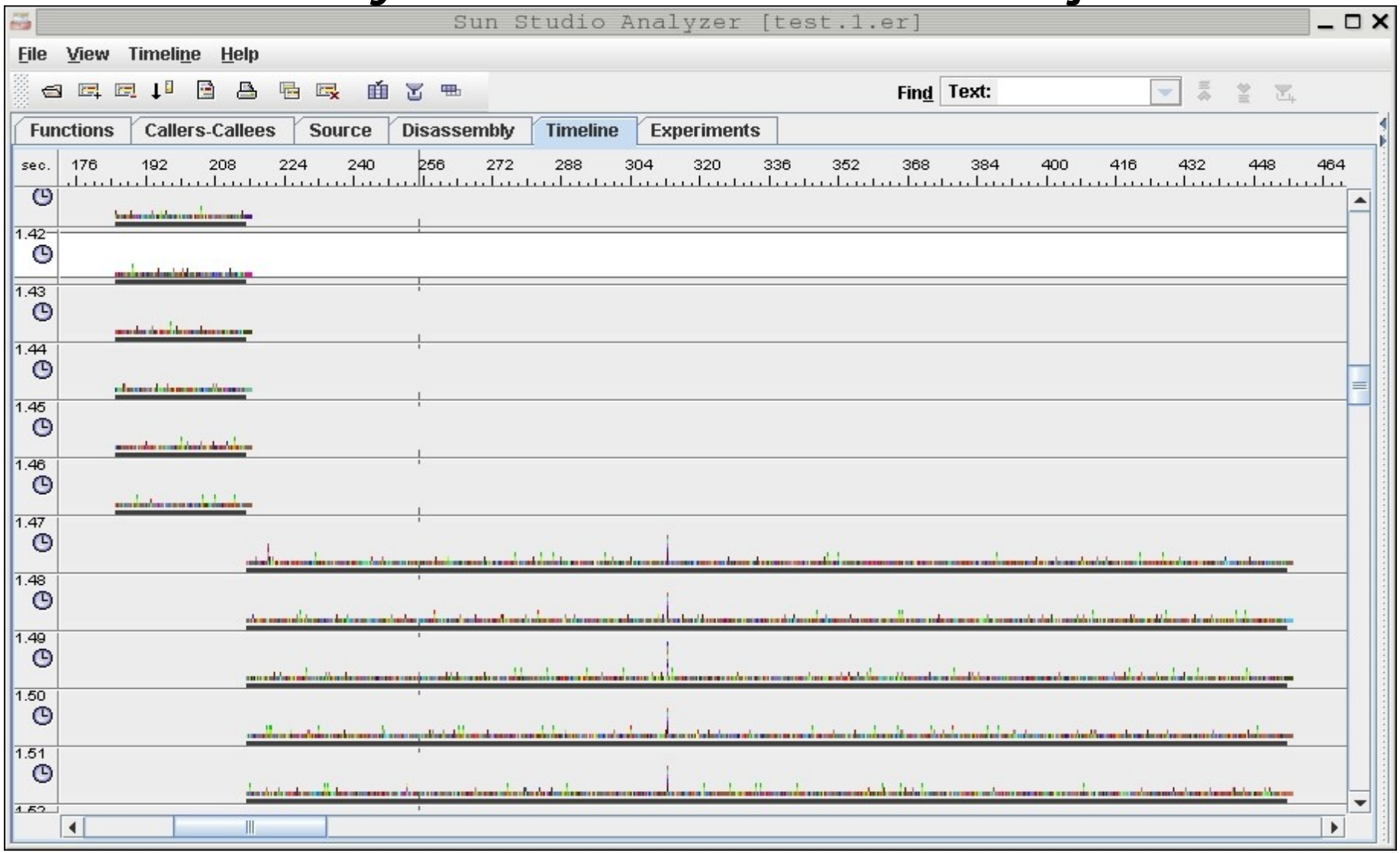
VTune: Call Graph, XML Parser



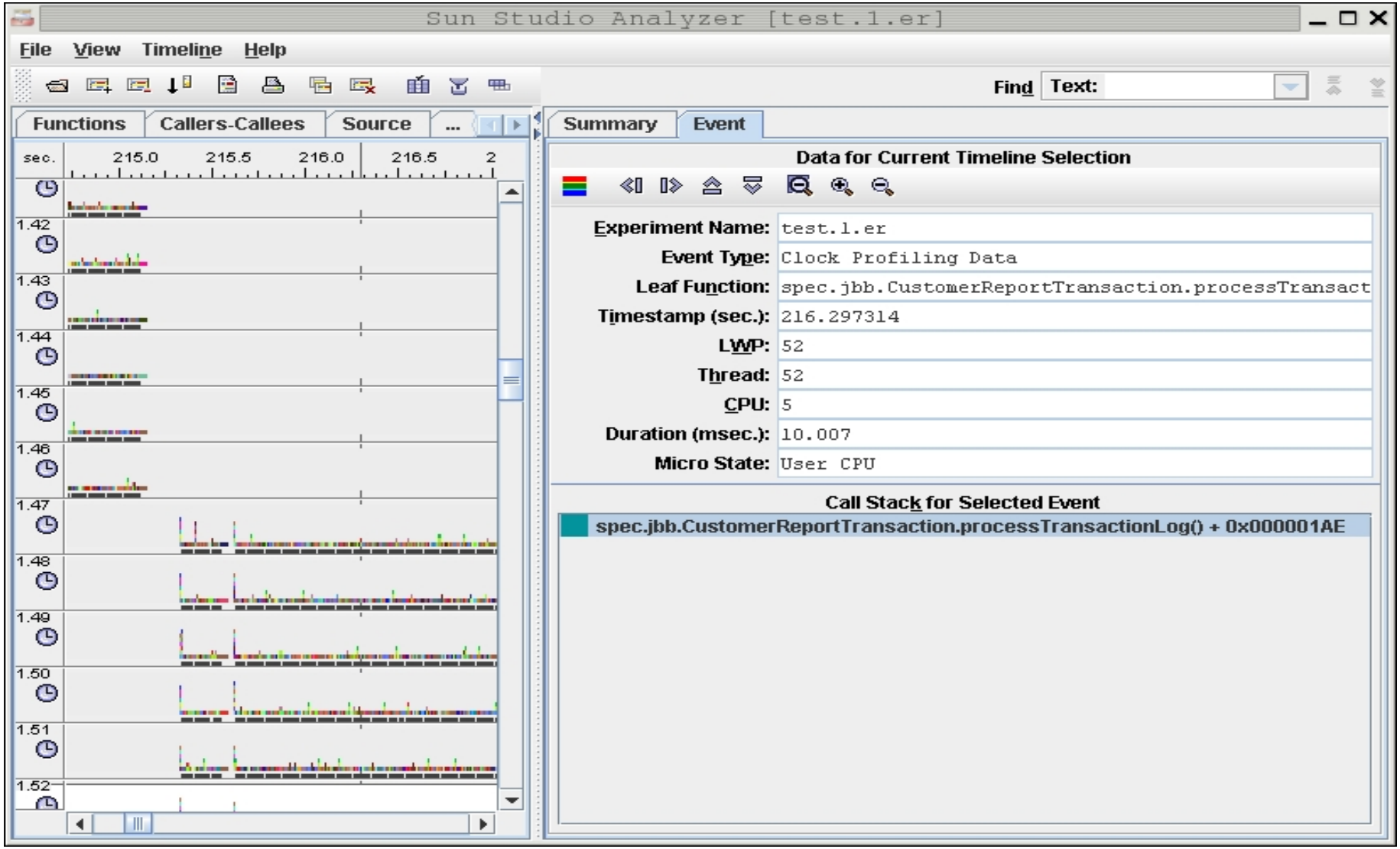
Studio Analyzer: JVM Machine Threads, SPECjbb2005



Studio Analyzer: Transition, SPECjbb2005



Studio Analyzer: Stack Trace, SPECjbb2005



Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

SPECjvm2008: scimark.FFT.large

- scimark is a Java technology-based floating point benchmark from NIST
- FFT is “Fast Fourier Transform”
- One dimensional, in-place algorithm with bit reversal
- n*log(n) complexity
- Large datasets: 2 million double complex values (32mb)
- Data access pattern is in steps of 2^n (n=20,19,18....)
- What's the story?
  - VTune data indicates low Instructions/Clock (IPC)
  - VTune cache miss profiles indicate very large cache miss rate
  - VTune data drill-down shows high cache miss rate inside one tight loop

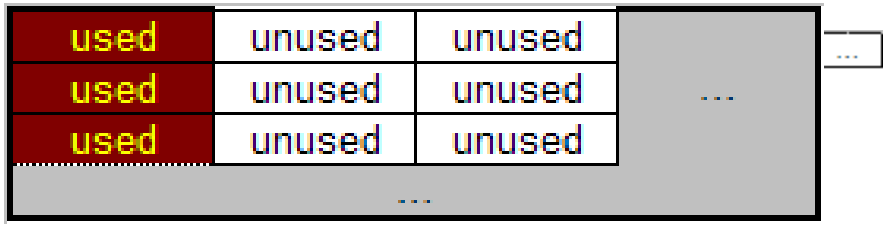
Problem Analysis

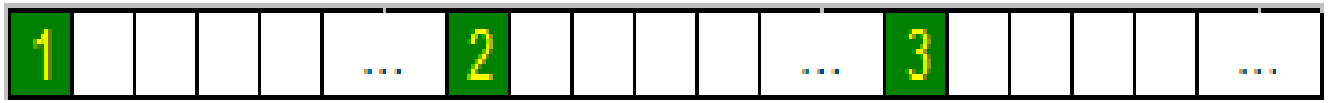
- Reads and writes are in 2^n steps

```
double data = new double[2*1024*1024];
for (int j = 0; j < step; j++) {
    double sum = 0;
    for (int i = 0; i < data_size; i += step) {
        sum += data[i+j]; data[i+j] += sum;
    }
}
```

2^n access stride in the inner loop

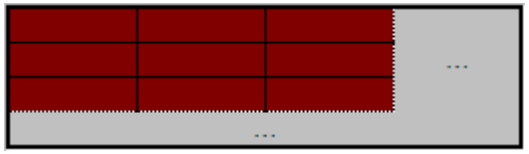
Cache Miss



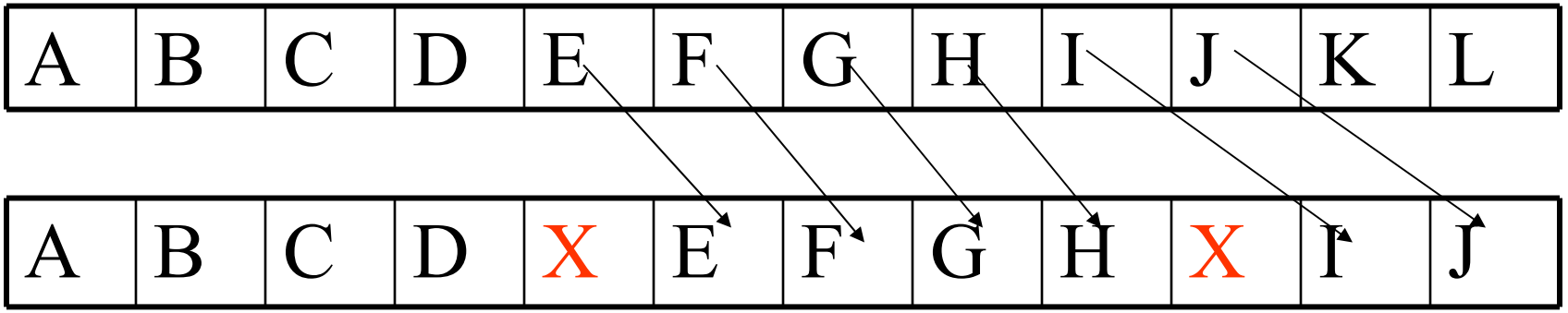


- Set-associative caches; an address must be placed in one specific set
  - Low bits of all addresses in a given set are identical, so a 2^n stride maps all addresses in the sequence onto the same set
  - Only a small fraction of the cache is used

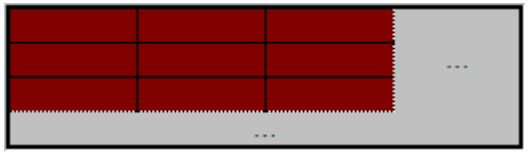
Change the source code



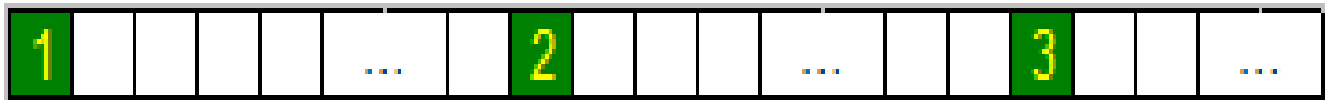
- Option 1: insert dummy element(s) into the array every k elements and change code to read/write from the new array correctly: ABCDEFGH => ABCDxEFGHx
  - new_index = old_index + floor(old_index / k)
  - Increases footprint as well as number of instructions executed per operation but allows entire cache to be used

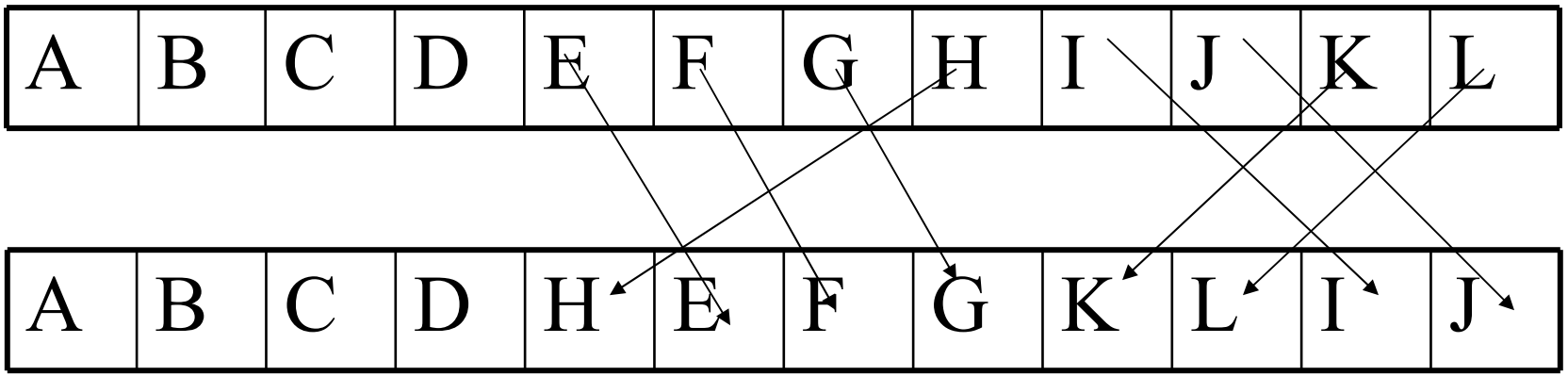


Change the source code



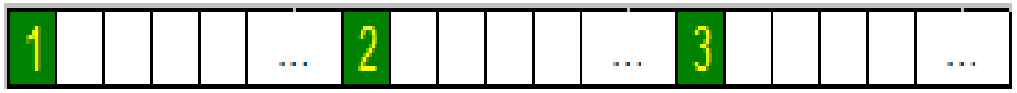
- Option 2: move data items around in some pattern: ABCDEFGHIJKL.... => ABCDHEFGKLIJ....
  - More complex code necessary to correctly access needed element => many more instructions than option 1
  - Footprint unchanged

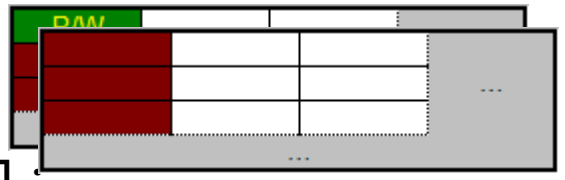




Option 3: Loop Interchange + Scalar Promotion

- From

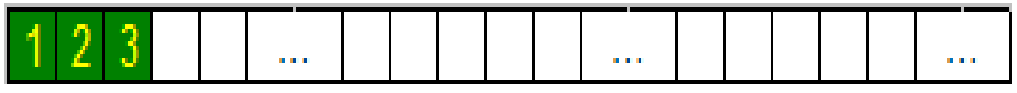


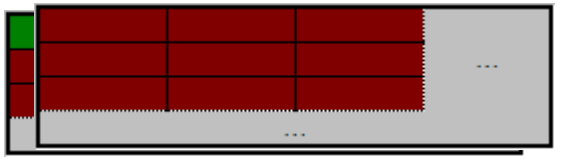


```
double data = new double[2*1024*1024];
for (int j = 0; j < step; j++) {
    double sum = 0;
    for (int i = 0; i < data_size; i += step) {
        sum += data[i+j];
        data[i+j] = sum;
    }
}
```

Cache Miss

- To





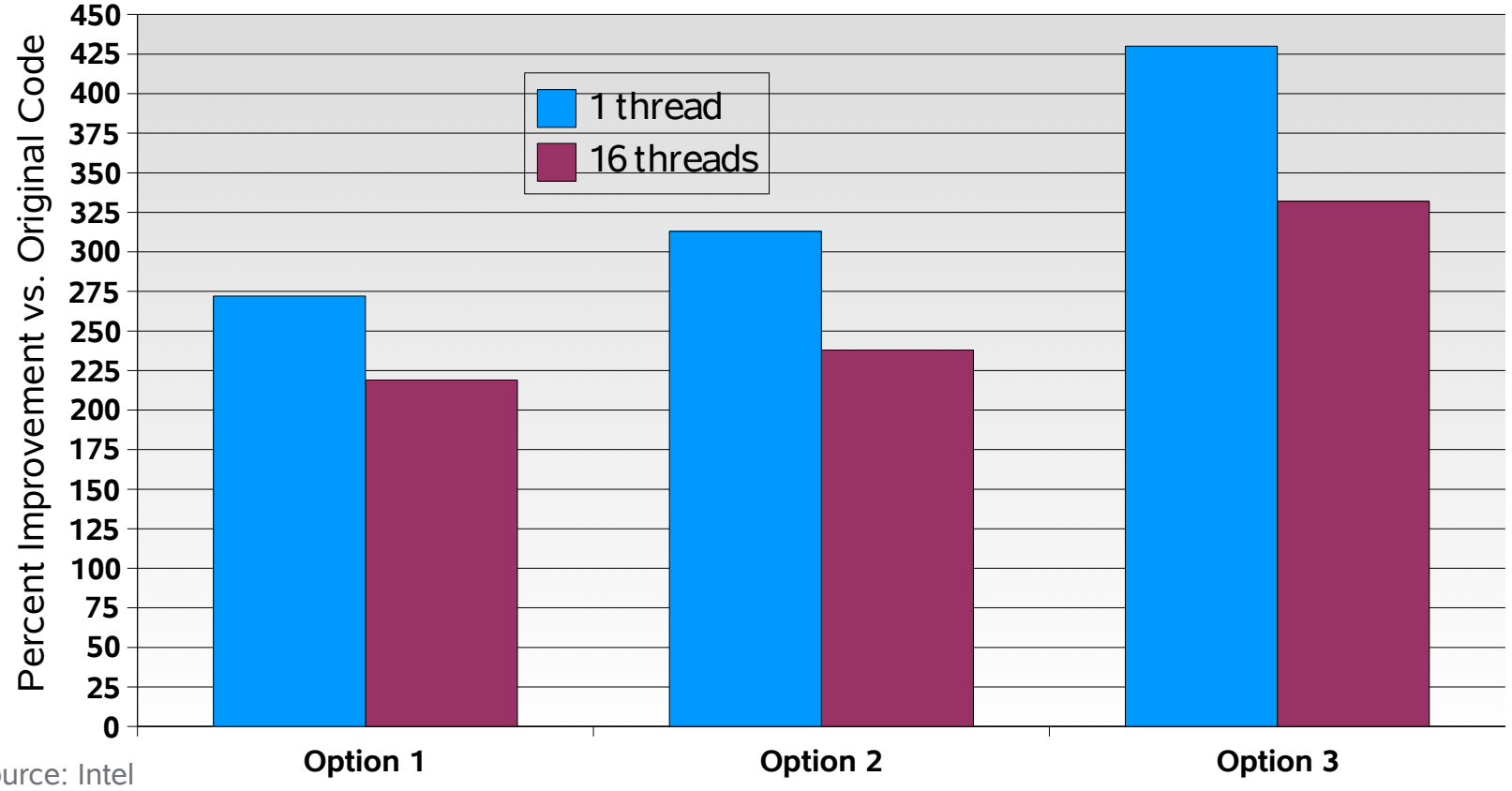
```
double data[] = new double[2*1024*1024];
double sum_array[] = new double[step];
for (int i = 0; i < data_size; i += step) {
    for (int j = 0; j < step; j++) {
        sum_array[j] += data[i+j];
        data[i+j] = sum_array[j];
    }
}
```

Promote sum to an array

Interchange loops

Cache Hit

Closing the Loop



Source: Intel

- 4 chip, 16 core Tigerton 2.93GHz 1066MHz bus
  - Measurements on other platforms show similar improvements
- Option 3 is best
  - Minor footprint increase, smallest code change

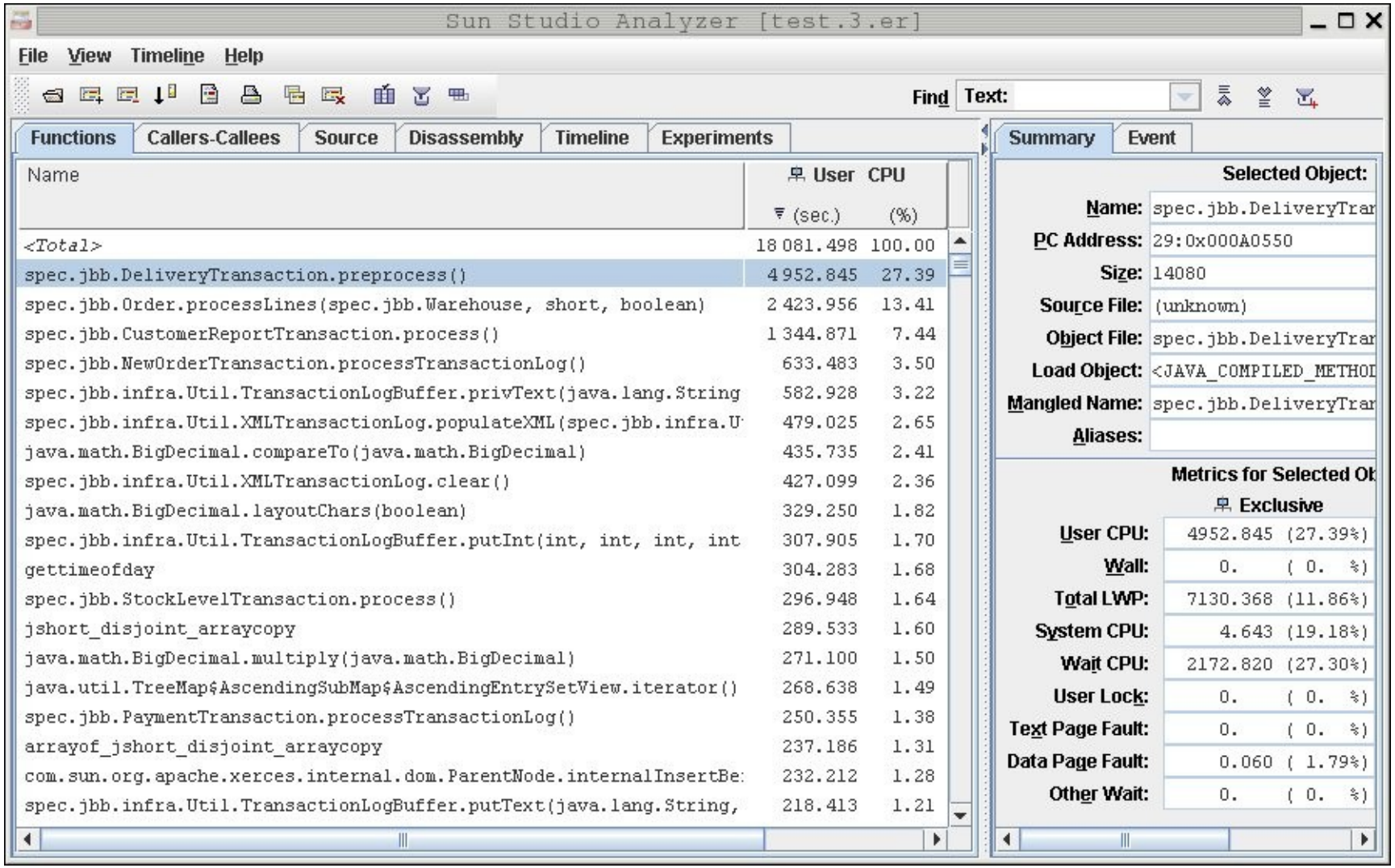
Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

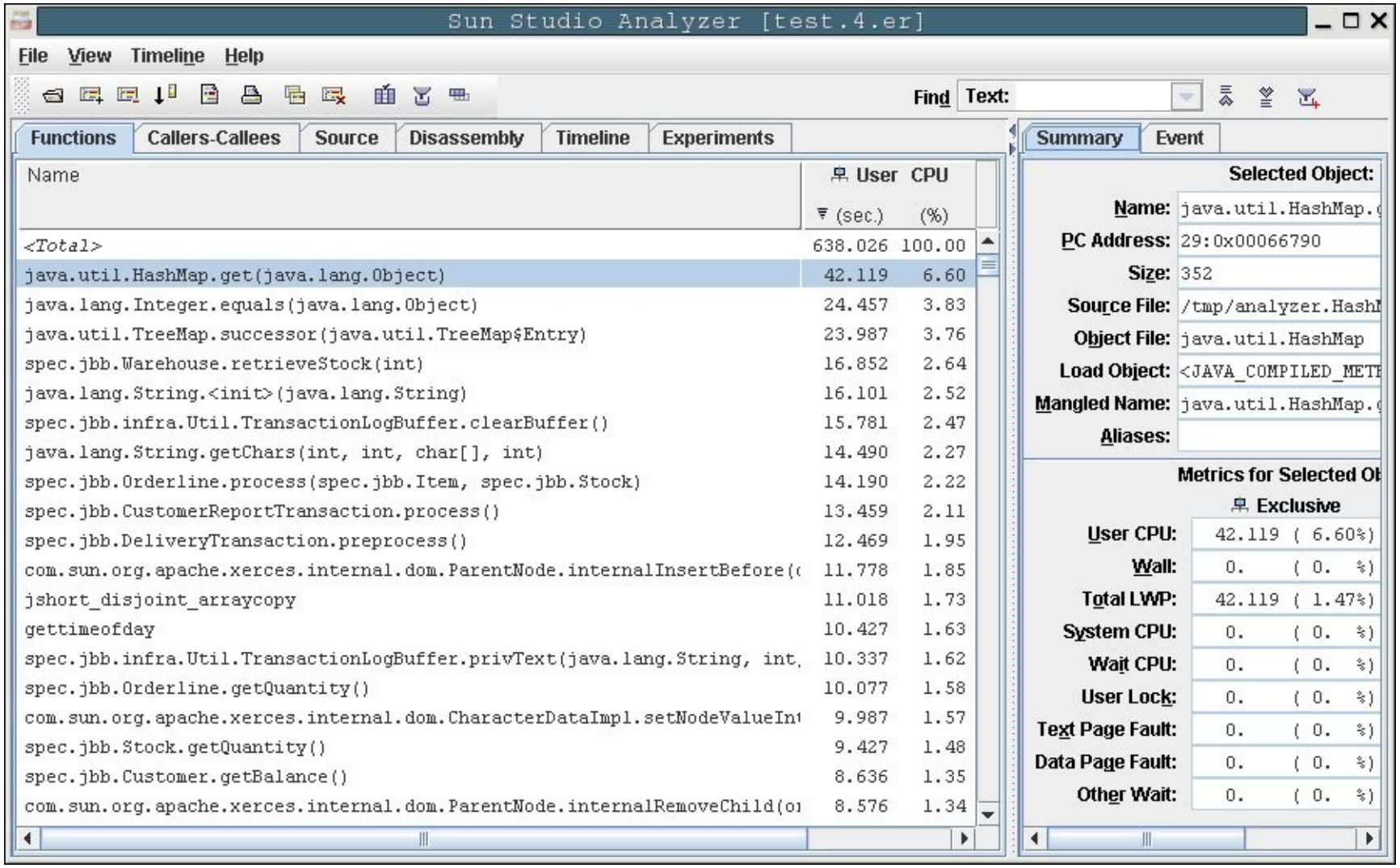
SPECjbb2005

- Emulates a three-tier client/server warehouse management system, emphasizing the middle tier
- Measures warehouse transaction rate, one thread per warehouse
- In theory, “embarrassingly parallel”
  - Measures hardware, JVM machine and OS scalability
  - Single- and multi-JVM machine run modes
  - Single-JVM machine mode is the real test of scalability
- Object allocation intensive
- Spoiler: *very* memory system sensitive
  - The smaller the caches, the more sensitive
  - The longer the main memory latency, the more sensitive
  - The more NUMA the memory system, the more sensitive

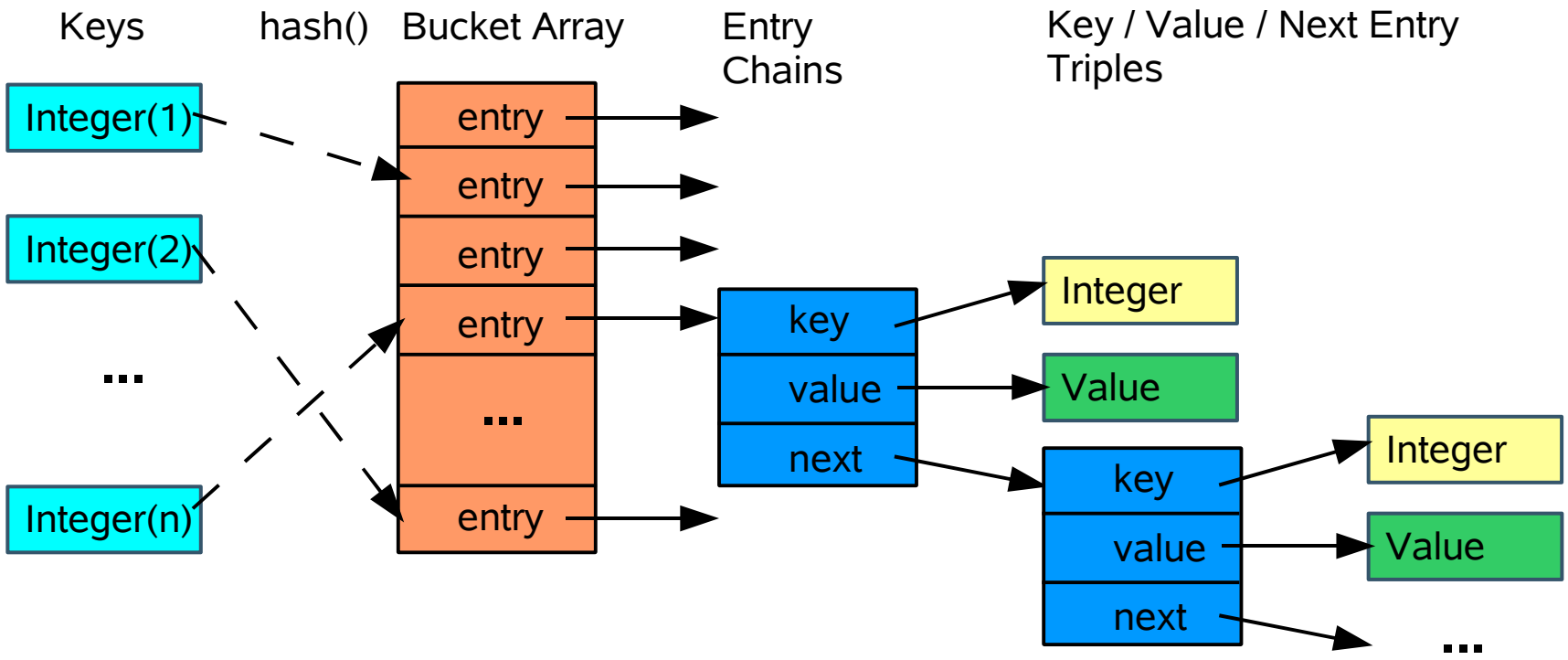
SPECjbb2005: Studio Analyzer, Profile



SPECjbb2005: Studio Analyzer, -XX:-Inline



HashMap Organization: Integer Keys



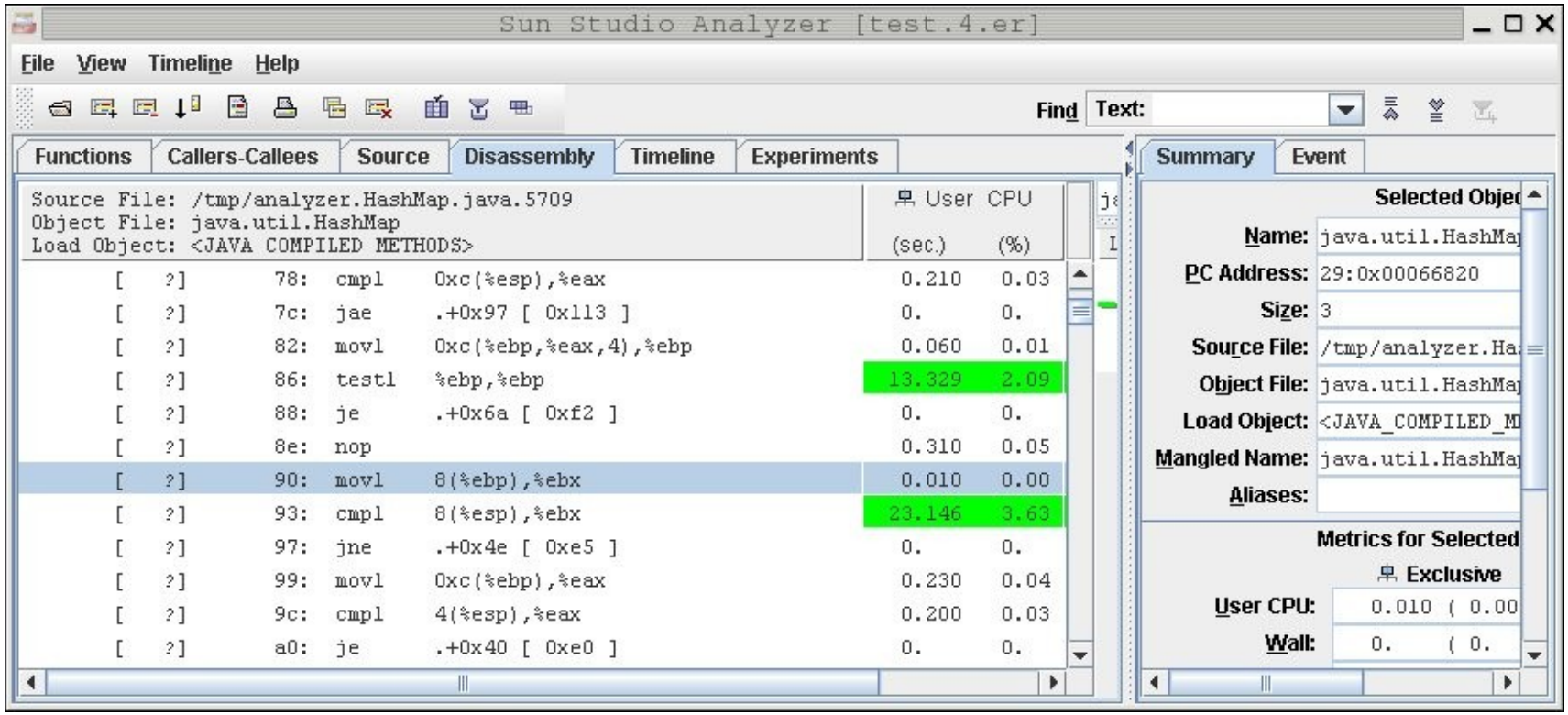
- HashMap is implemented as a chained hash table
- Chained hash tables have poor memory access locality
- Even if average chain length is one, could miss on every access: Bucket Array element, Entry, Key and Value objects

SPECjbb2005: HashMap.get(), Misses in Orange

```
public V get(Object key) {
  int hash = hash(key.hashCode());
  for (Entry<K,V> e = table[indexFor(hash,table.length)];
       e != null; e = e.next) {
    Object k;
    if (e.hash == hash &&
        ((k = e.key) == key || key.equals(k)))
      return e.value; // Possible miss
  }
  return null;
}

int index; HashMap table; ...; // Fill up table
Value value = table.get(index);// Many, many get()s!

// But 'index' is auto-boxed, so the call
// to get() is actually

Value value = table.get(Integer.valueOf(index));
```

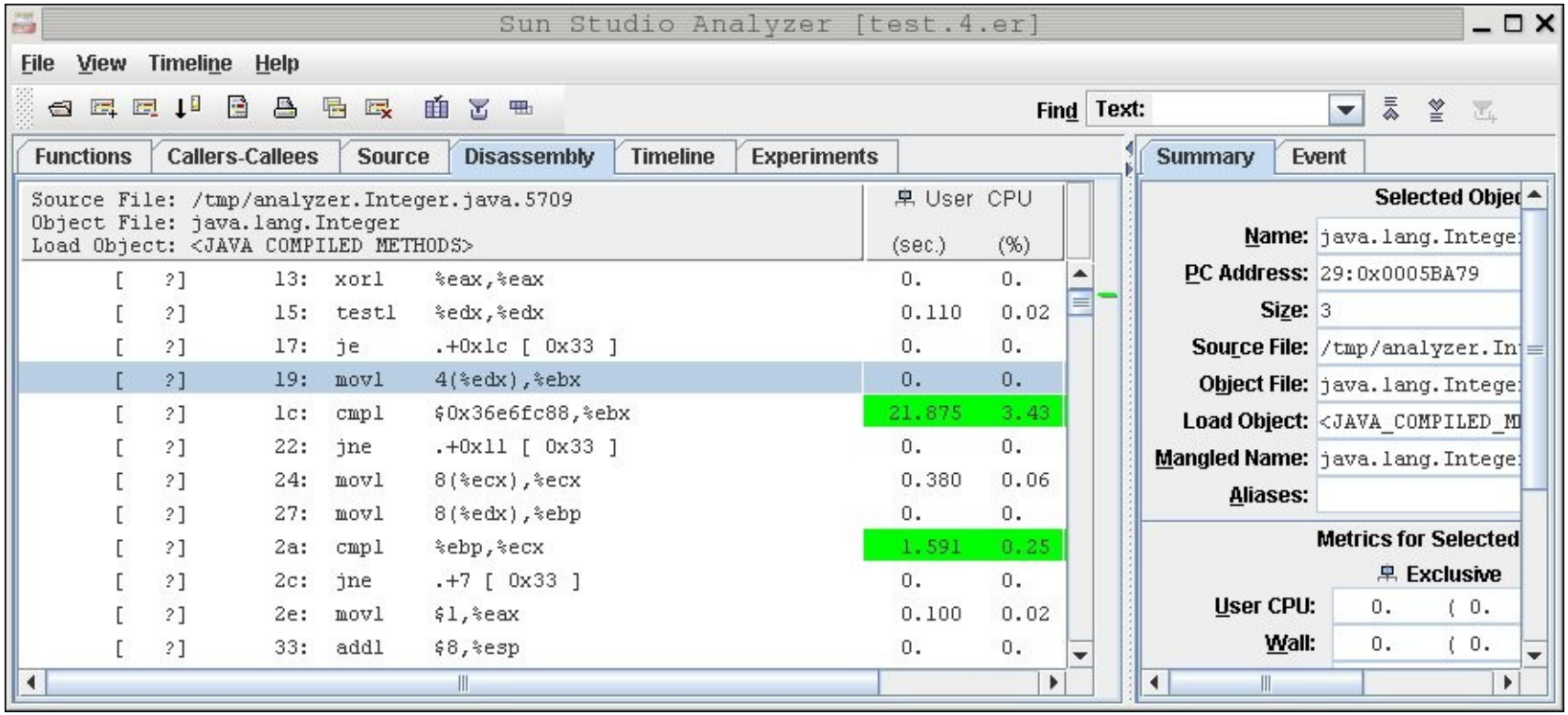
SPECjbb2005: Studio Analyzer, Assembly

- Bucket Array access
  - `table[indexFor(hash,table.length)]`
- Chain Entry access
  - `e.hash`

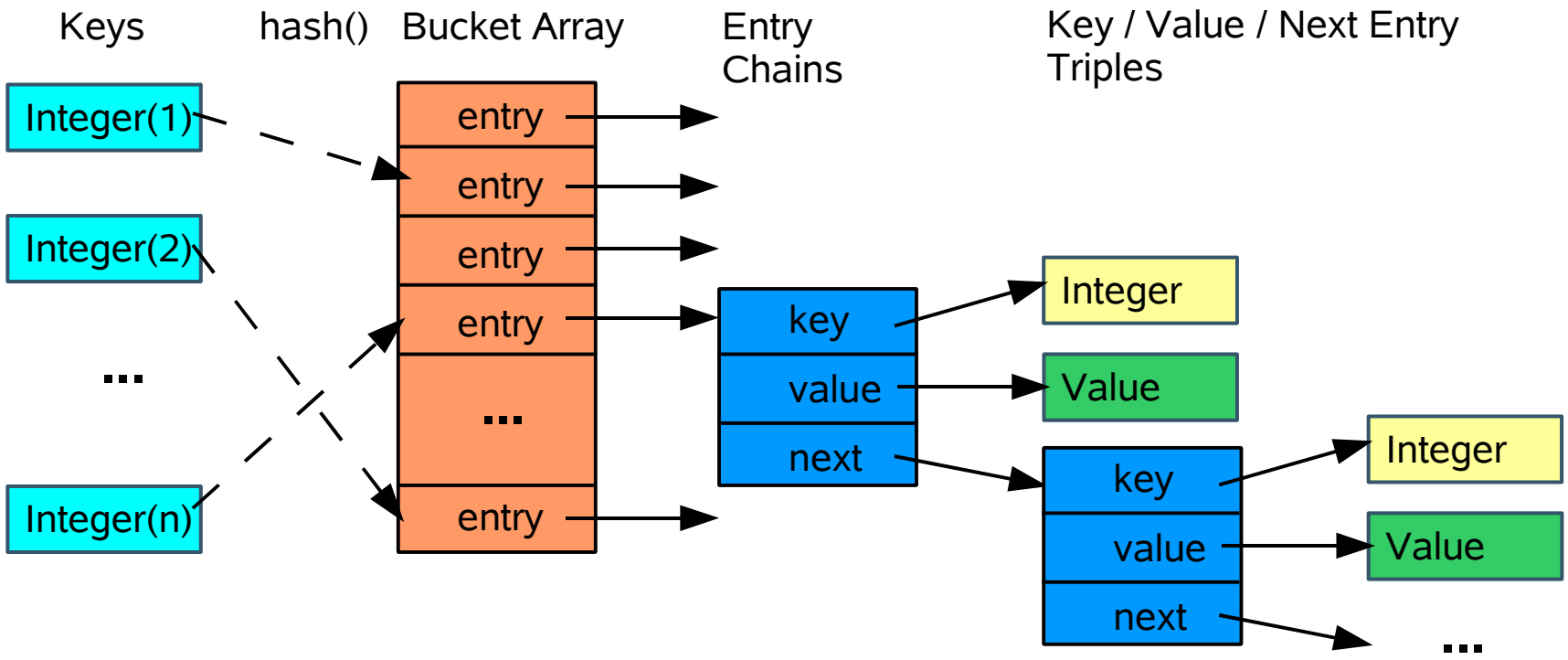


SPECjbb2005: Studio Analyzer, Assembly

- Key access
  - `key.equals(k)`
    - `key.class`
    - `key.value`



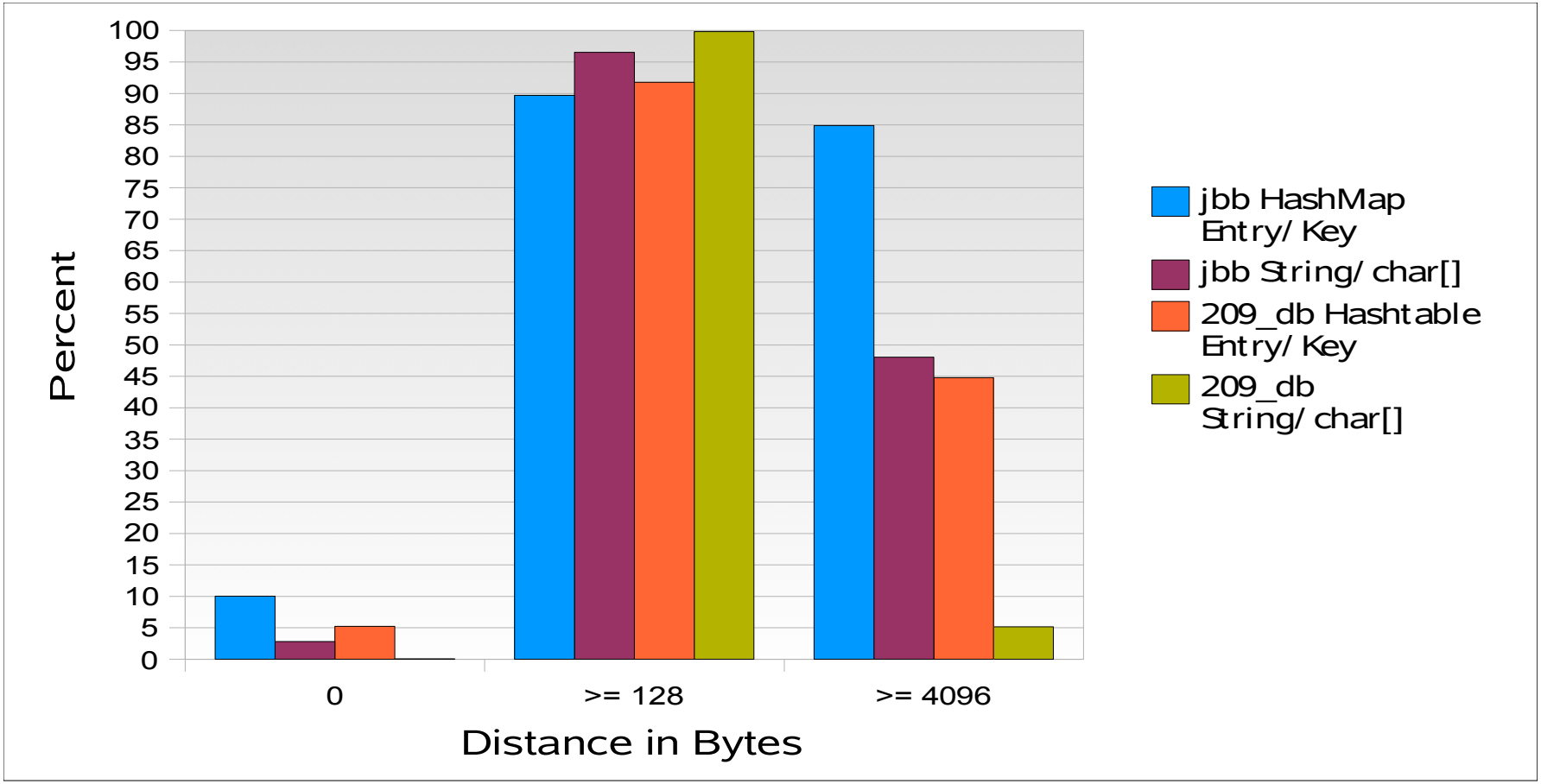
HashMap Organization: Integer Keys



Where in memory are all these guys?

Byte Distance: Parent End to Child Start

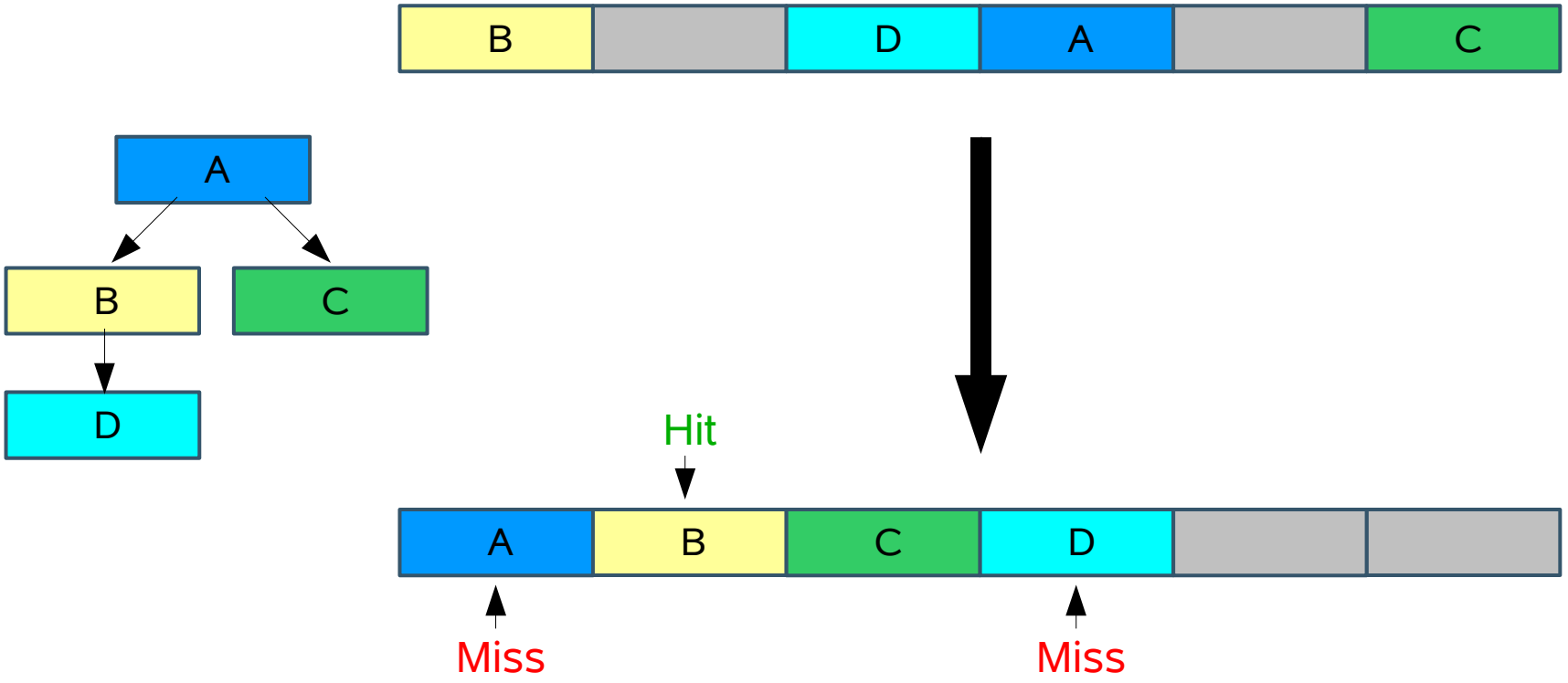
Percent of Object Instance Pairs a Given Distance Apart



Source: Sun Microsystems

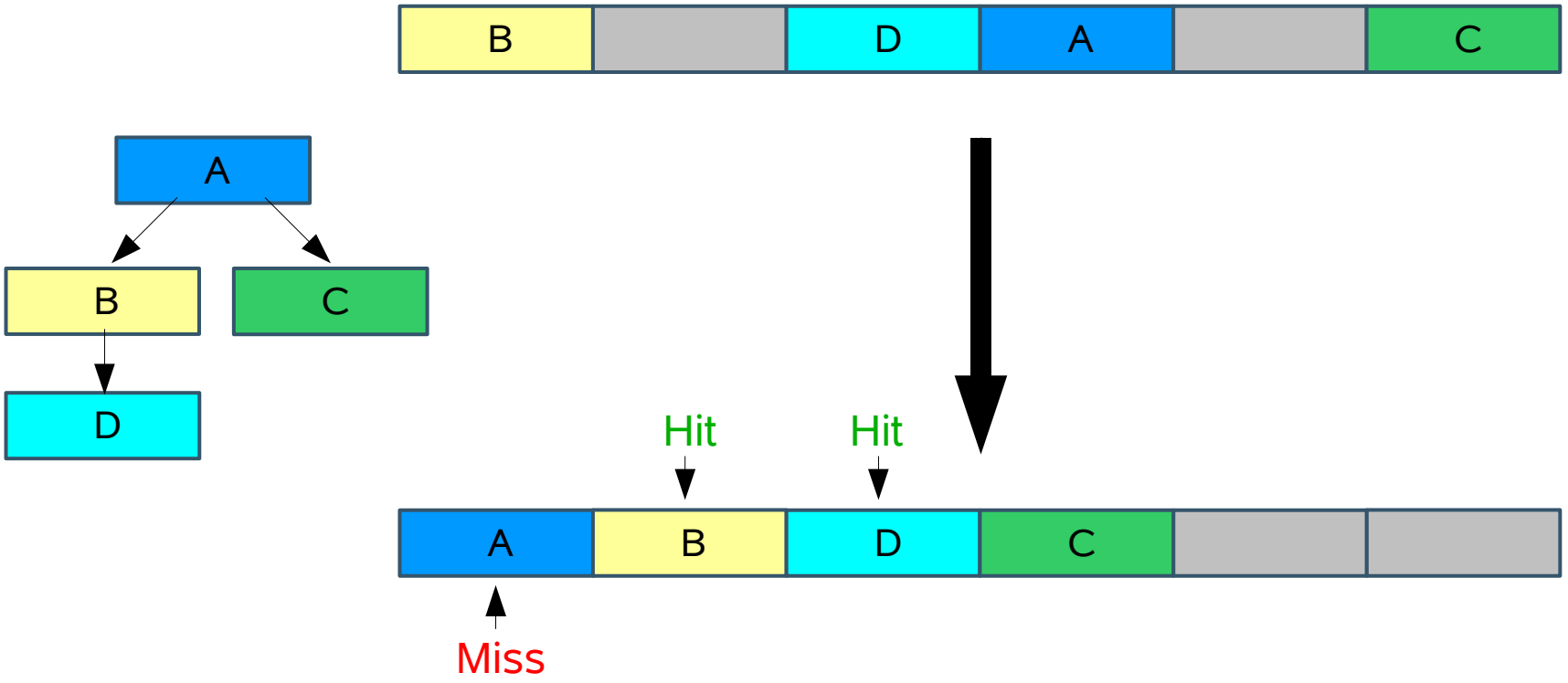
Breadth-First GC Object Copying

- Copy object
- Copy all its children
- For each child, copy all its children, ...

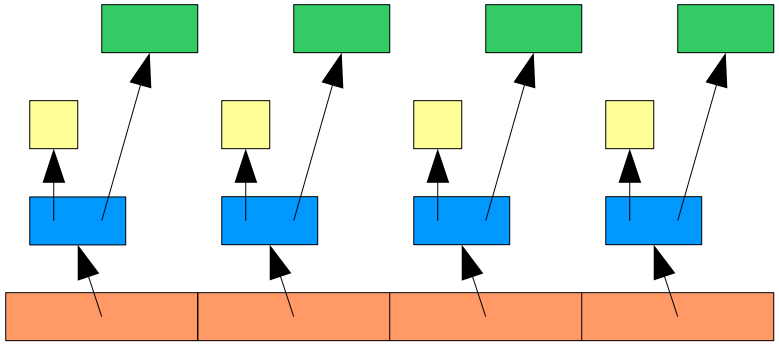


Depth-First GC Object Copying

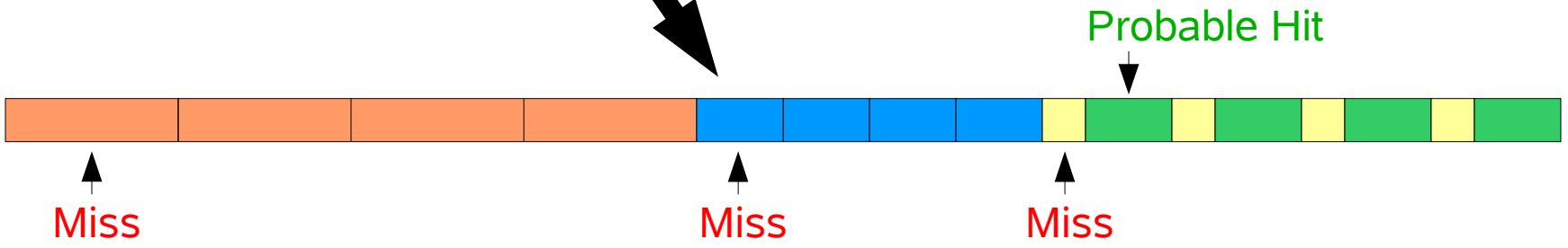
- Copy object
- Copy first child
- Copy first child's first child, ...
- Copy second child, then second child's first child, ...



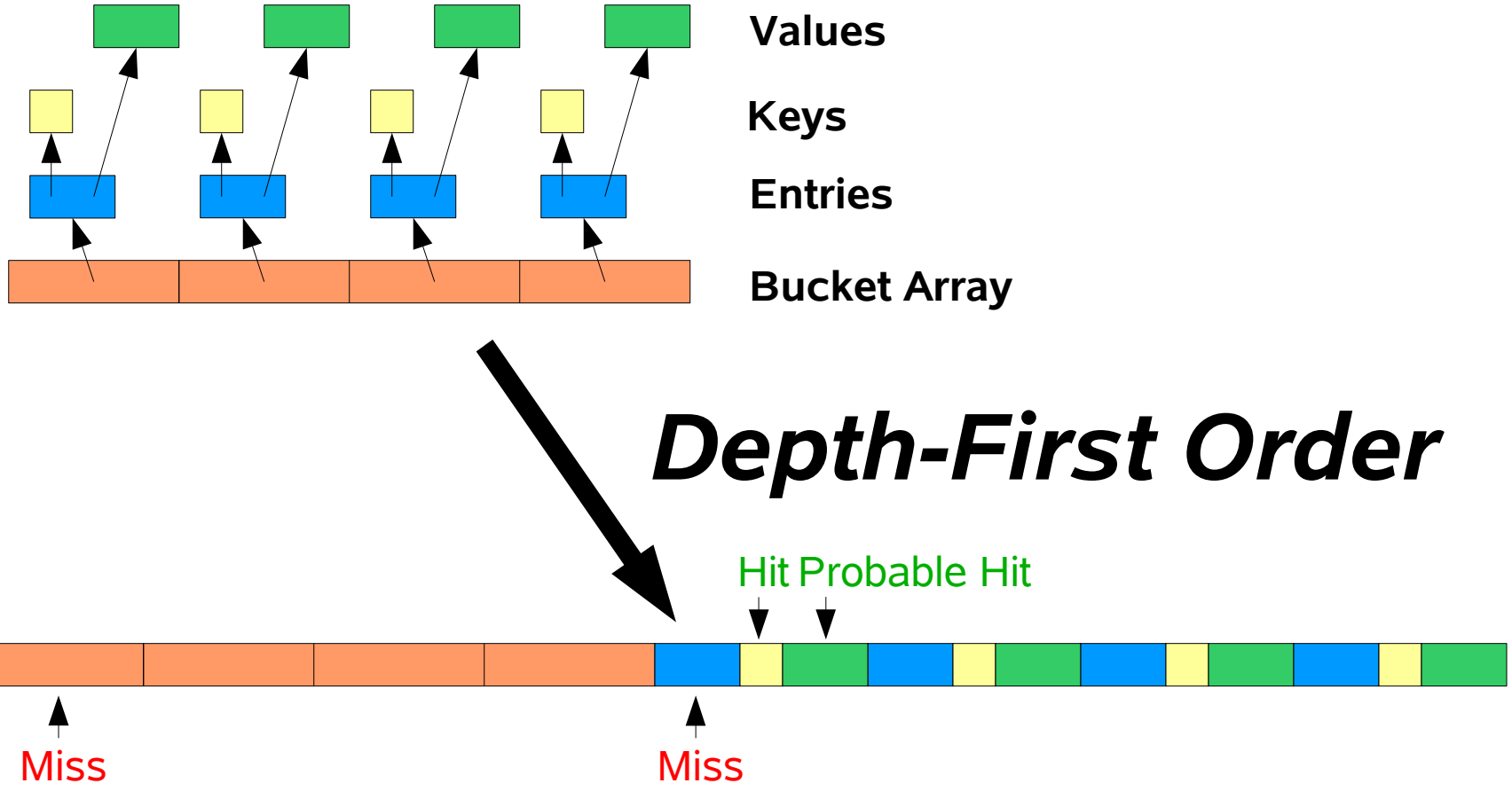
Breadth-First Copying: HashMap



Breadth-First Order

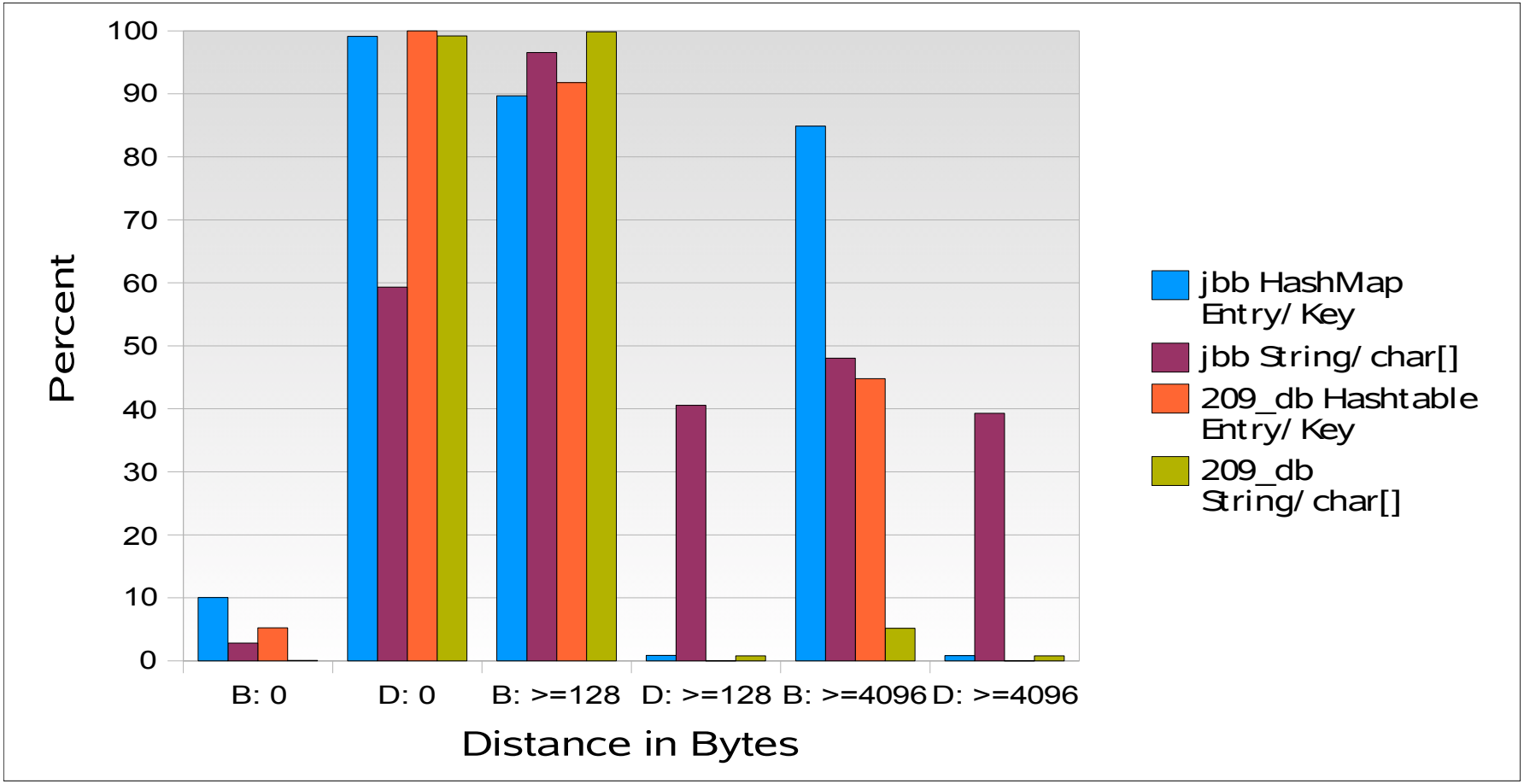


Depth-First Copying: HashMap



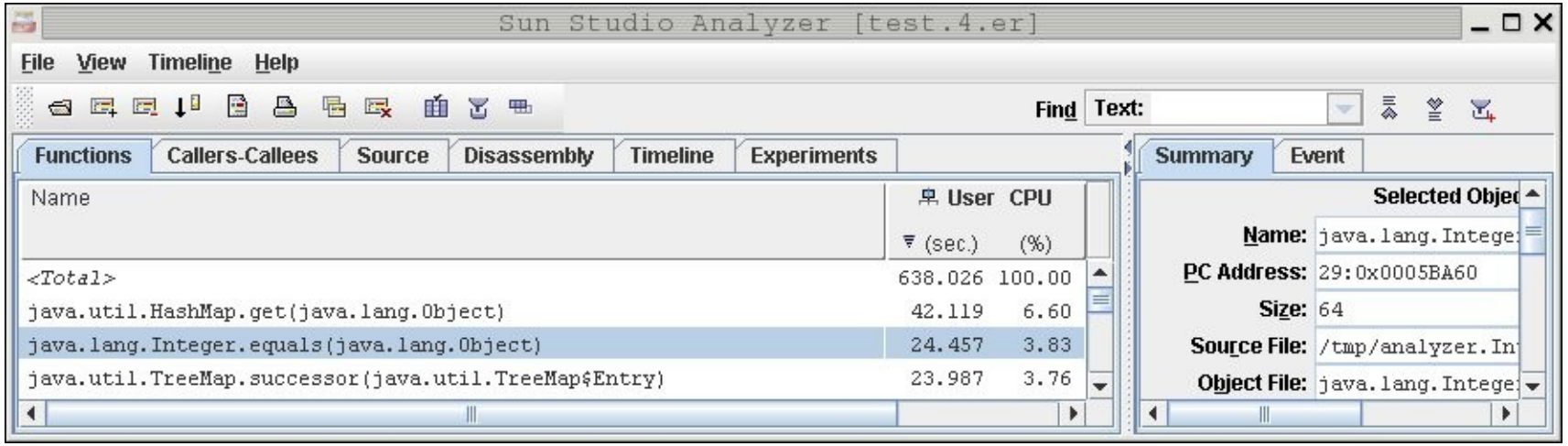
Byte Distance: Parent End to Child Start Breadth-First vs. Depth-First

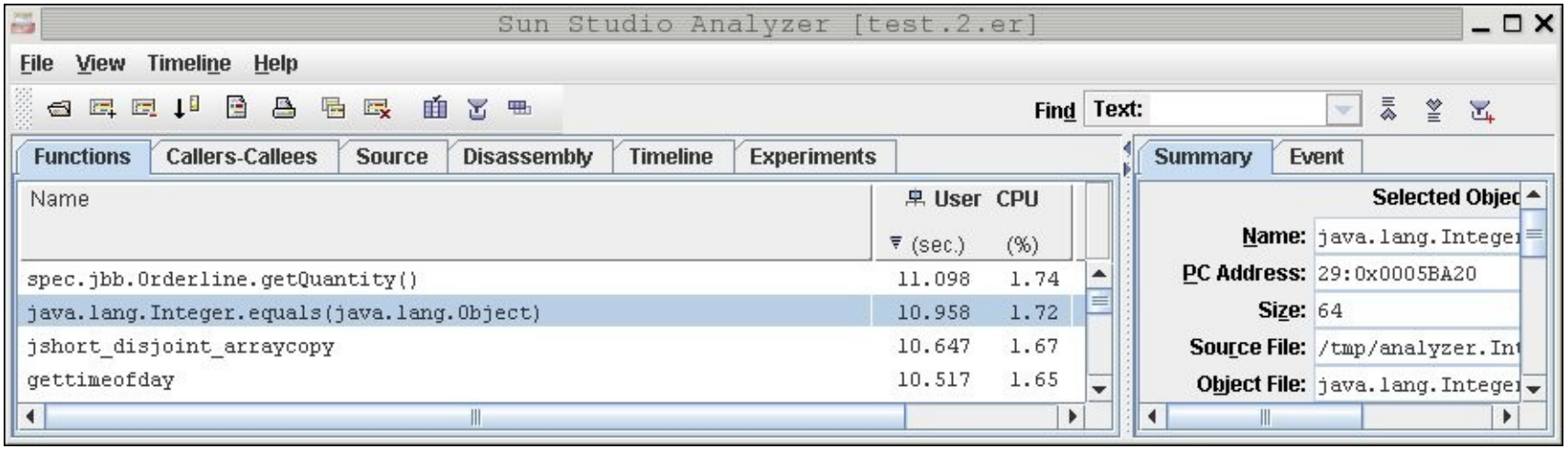
Percent of Object Instance Pairs a Given Distance Apart



Source: Sun Microsystems

Breadth-First vs. Depth-First: SPECjbb2005

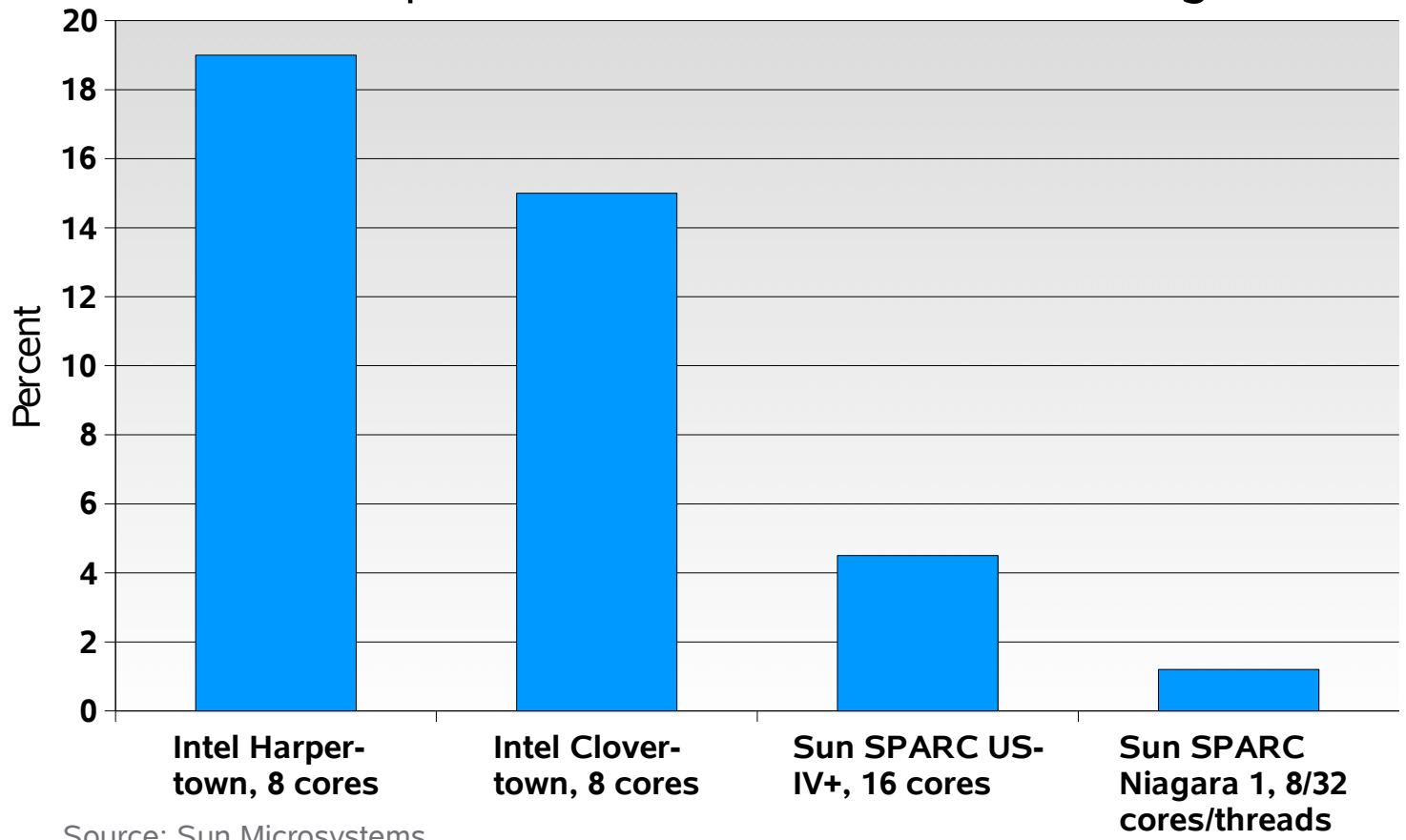




Breadth-First vs. Depth-First: SPECjbb2005

- Relative improvement, *not* absolute: see spec.org for the latter
- Benefit varies by platform
- Not just HashMap! TreeMap, String, etc. also benefit

Depth-First Percent Improvement Over Breadth-First, Single JVM Machine

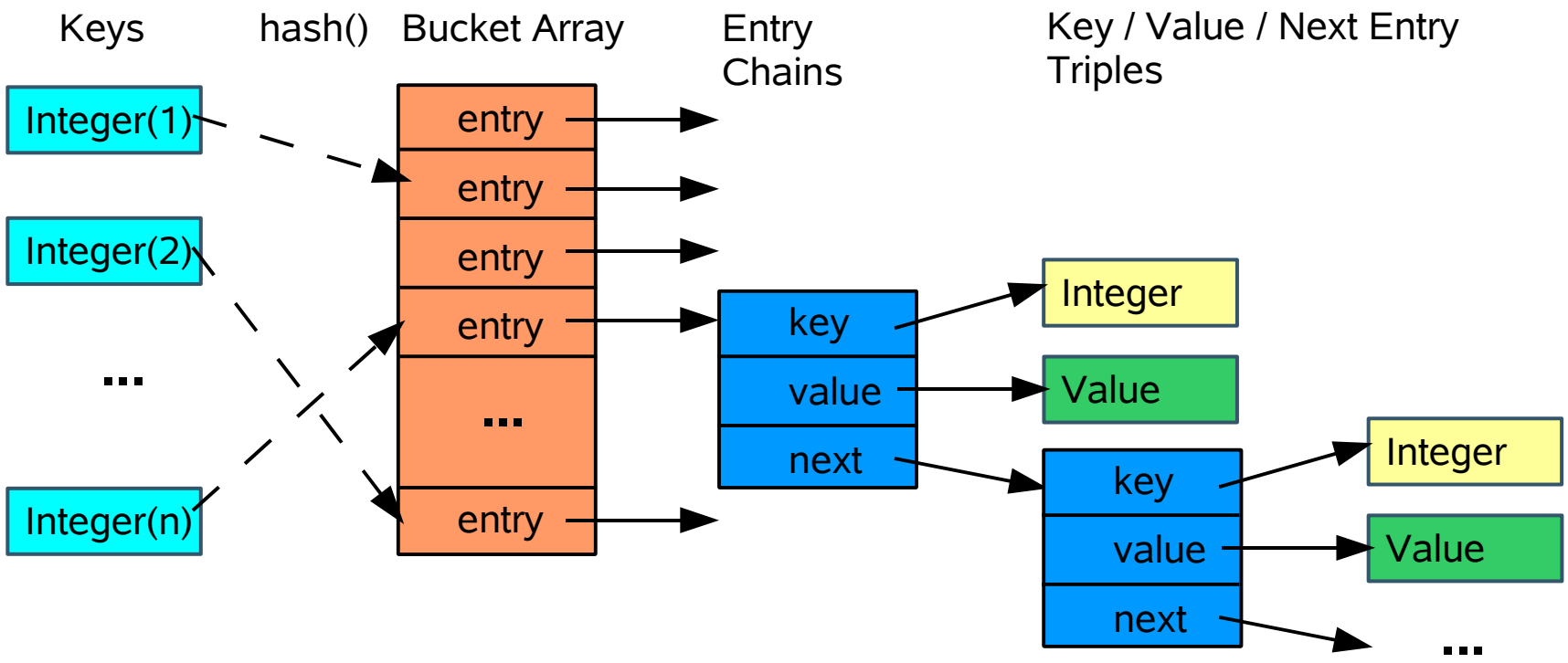


Source: Sun Microsystems

Breadth-First vs. Depth-First

- Any static policy will hurt some applications and help others
- A common pattern in Java technology applications is object arrays pointing to nodes
- Depth-First is better for this pattern
- Little or no degradation with respect to breadth-first on all other performance benchmarks we can get our hands on
- Many, e.g. SPECjvm98's 209_db, are much faster with depth-first
  - Strings and their char[]'s are co-located 100% of the time

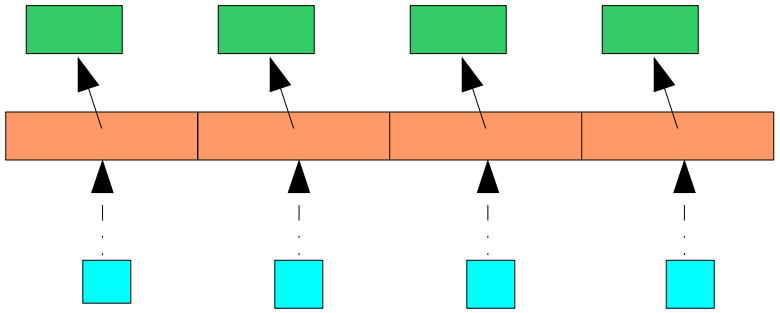
But Wait, There's More! Remember This?



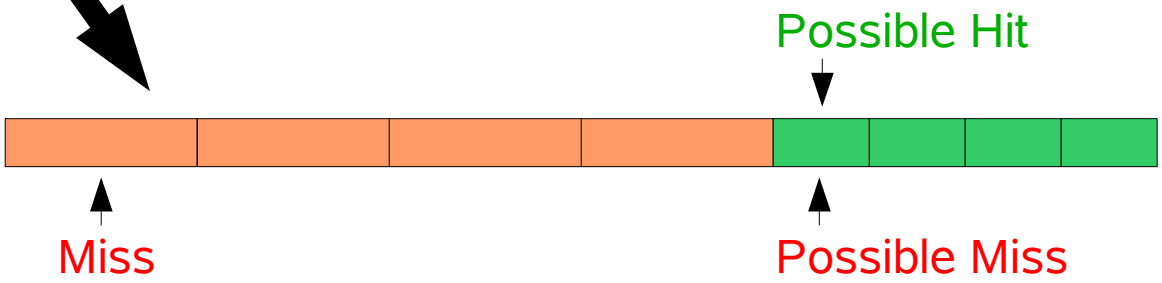
There *must* be a better way!

Yes, There Is: Specialize for Integer Keys

- Front the whole business with an Integer -> Value array cache, punt to backing HashMap on miss
- If front cache is big enough, the backing HashMap will stay out of processor cache: copying order doesn't matter



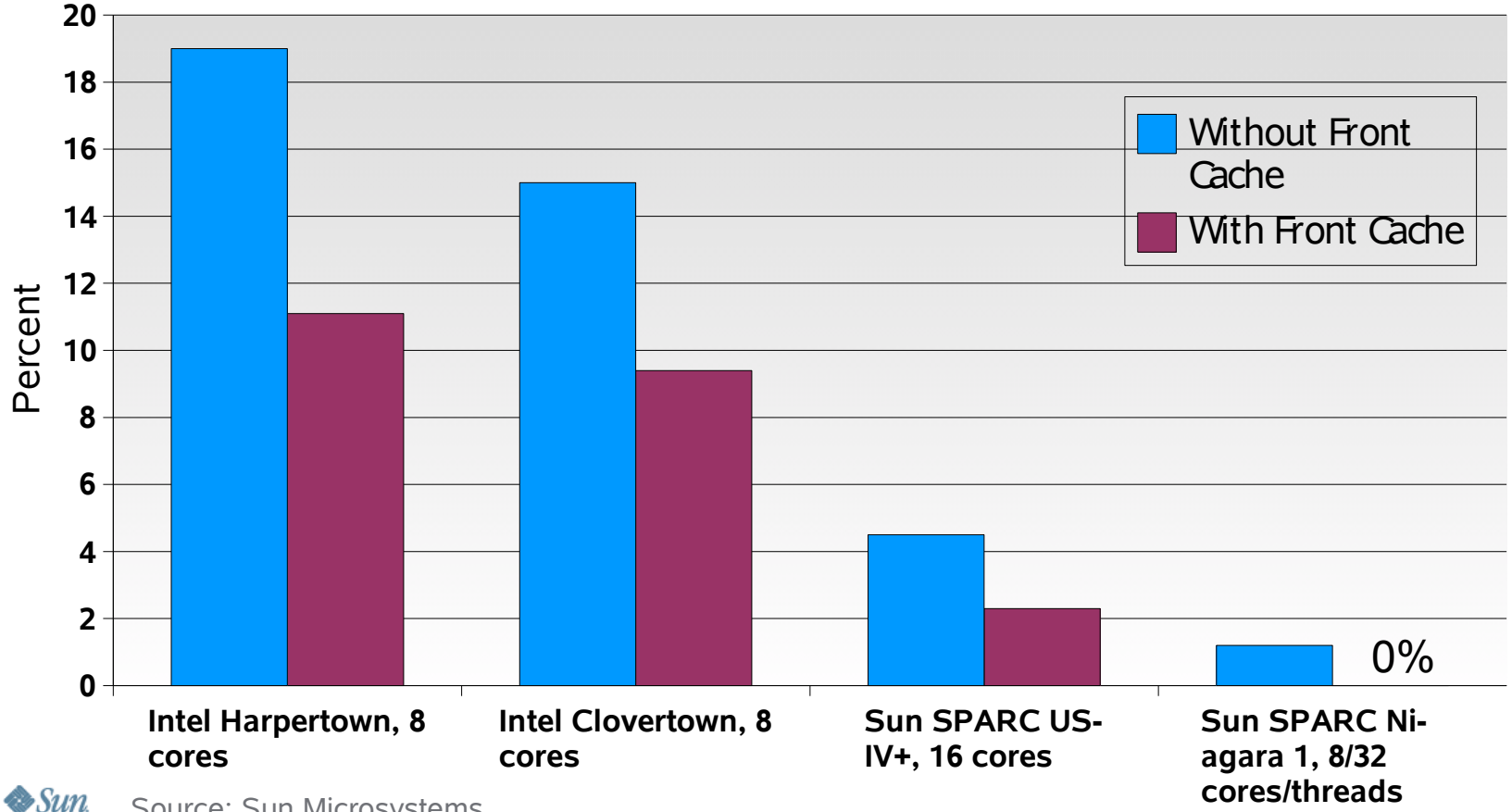
Breadth or Depth-First Order



Breadth-First vs. Depth-First: SPECjbb2005

- Relative improvement, *not* absolute: see spec.org for the latter
- Benefit varies by platform
- Not just HashMap! TreeMap, String, etc. also benefit

Depth-First Percent Improvement Over Breadth-First, Single JVM Machine



Source: Sun Microsystems

Demo

Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

Performance Analysis Takeaways

- Incorporate performance monitoring and analysis into the development cycle
  - Including, to the extent possible, production
- Use the available tools: they work really well
- Learn and apply basic hardware design knowledge
- The memory system is not your friend
  - Most performance problems can be traced to its limitations
- Results vary by platform
  - Optimizing cache utilization matters most when average memory access time is high relative to processor speed
  - Test on as many platforms as possible
  - WORA => do your best to eliminate platform to platform variability

Agenda

- Methodology: Hardware, JVM Machine, Class Libraries, Applications
- Tools, Tools, Tools
- Case Study 1: SPECjvm2008, FFT Large
- Case Study 2: SPECjbb2005
- Demo
- Performance Analysis Takeaways
- Java Library and JVM Machine Performance Frontiers

Java Library and JVM Machine Performance Frontiers

- Completely dynamic and adaptive execution
  - Hardware-assisted always-on dynamic profiling
  - Adaptive garbage collection
  - Predictability and throughput goals
  - Real-time
- Ergonomics writ large
  - Execution history recording and feedback
- High performance computing
  - Photo-real graphics generation, speech analysis and generation, financial analysis
  - Auto-vectorization and concurrentization
  - Massively parallel Java platform
- Startup and footprint: minimize class loading overhead
  - Eliminate static variables and initializers, horizontal class dependencies
  - Minimize object allocation during startup
- Memory-optimal data structures: wide-node TreeMap, B+Trees

THANK YOU

Kumar Shiv, Intel
Paul Hohensee, Sun Microsystems

TS-6434