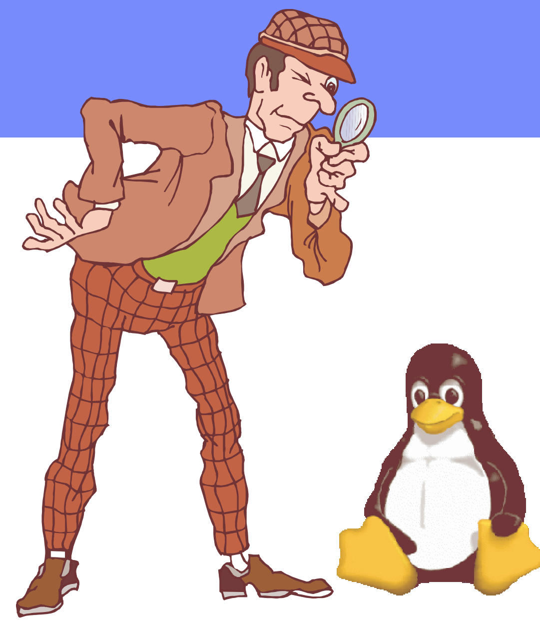
Performance Inspector Tools with Instruction Tracing and Per-Thread/Function Profiling



Milena Milenkovic, Scott T. Jones, Frank Levine, Enio Pineda

IBM

Outline

- **Introduction**
- Time/Event Profiling
- Callflow Profiling with Calibration
- Java Callstack Sampling: A Hybrid Approach
- Report Visualization with VPA
- Conclusion

Introduction

- Growing software complexity → more to analyze
- The Performance Inspector™ (PI) project:

  mature set of cross platform tools to help analyzing

  application and system performance

  – Small tools team dedicated to supporting and improving the tools
- Hosted on SourceForge (http://perfinsp.sourceforge.net)
  – Support for Intel x86, Intel/AMD x86_64, IBM PowerPC64 and System z

PI Capabilities: Comprehensive Analysis Reports

- Processor utilization
- Where time is spent;

  where cache misses or other events occur
  - Sample-based profiler tuned to minimize impact on system
  - Supports dynamically generated code and address space reuse
- Program flow analysis
- Efficient counter virtualization by thread

PI Capabilities: Java Analysis Reports

- Heap and objects information
- Lock contention
- Time or instructions executed by method
  - Callflow profiling tuned for performance and accurate calibration
  - Results using instructions completed comparable to ITrace
  - Accurately estimates performance without instrumentation
- Context analysis - Java callstack sampling

PI Customers

- Performance analysts and developers across IBM
  - Over 1500 users
  - Used to analyze complex applications and predict improvements of proposed code changes
- Open source community

Outline

- Introduction
- **Time/Event Profiling**
- Callflow Profiling with Calibration
- Java Callstack Sampling: A Hybrid Approach
- Report Visualization with VPA
- Conclusion

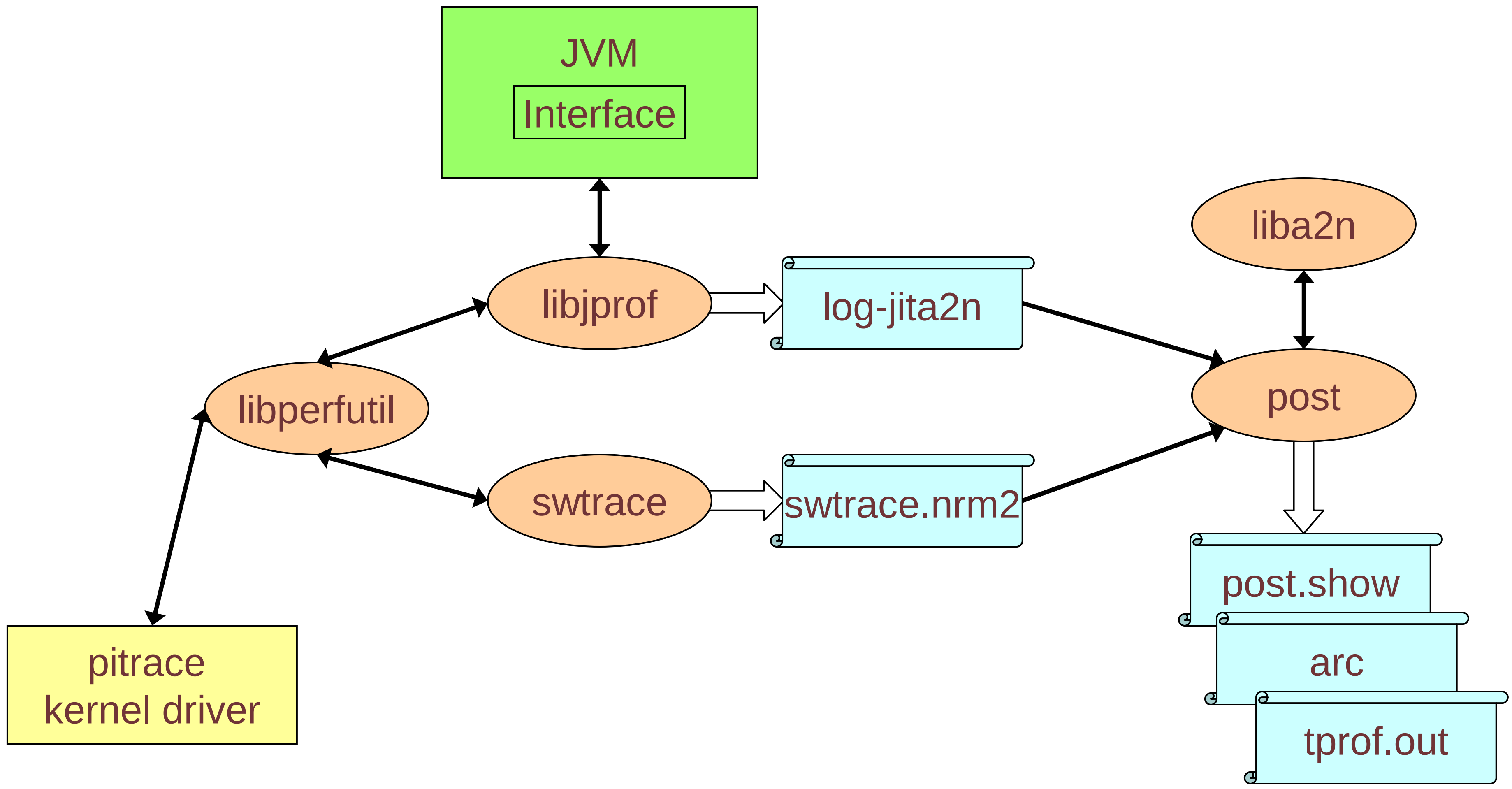
tprof

- Used for system performance analysis
  - To detect hot-spots in an application
  - To indicate the resource distribution
- Based on sampling
  - Interrupt the system periodically, with a given rate (time-based), or when a performance monitoring counter reaches a given threshold (event-based)
  - Record the address of the interrupted code and its pid/tid

tprof

- run.tprof script
- Post-processor generates tprof.out report
- Shows sample distribution for
  - Each process
  - Module within a process
  - A symbol within a module
  - Same as above for threads

Tracing Facility: Java Application Tracing



hellop.java

```
class hellop
{  static long ii;

   public static void main(String args[])
   {  int    i;
      int    j = 0;
      String eol;

      int iter = Integer.parseInt(args[0]);

      eol = ( args.length > 1 ) ? "\r" : "\n";
      System.out.println("\nHello, World!\n");

      for (i = 0; i < iter; i++) {
         myA();
         System.out.print( ii + eol );
      }
      System.out.println( ii );
   }

static void myA()
  {
     myC();
  }

  static void myC()
  {
     for (int j = 0; j < 1000; j++) {
        ii++;
     }
  }
}
```

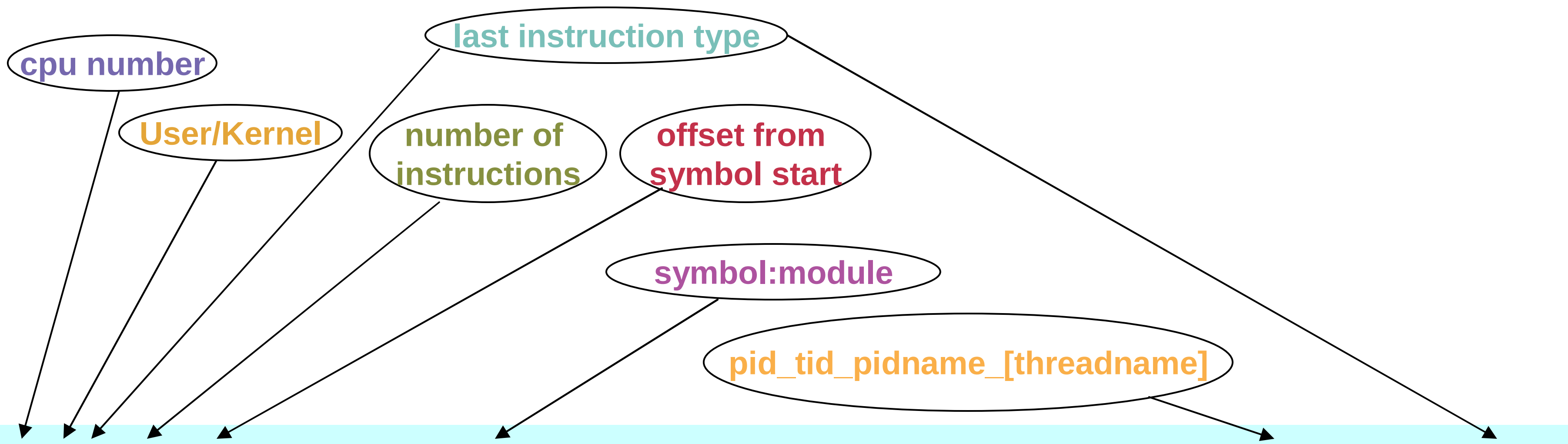

tprof - A tprof.out Excerpt

```
================================
)) Process_Thread_Module_Symbol
================================
LAB TKS %%% NAMES
PID 2372 51.25 java_103c
  TID 1704 36.82 tid_main_103c
    MOD 721 15.58 vmlinux
      SYM 123 2.66 _spin_unlock_irqrestore
      SYM 88 1.90 system_call
...
    MOD 338 7.30 JITCODE
      SYM 81 1.75 hellop.myC()V
      SYM 32 0.69 hellop.main([Ljava/lang/String;)V
      SYM 17 0.37 java/lang/String.indexOf(II)I
      SYM 17 0.37 java/io/PrintStream.write(Ljava/lang/String;)V
```

ITrace

- ITrace is branch trace, recorded using the underlying hardware support for trap on branch/taken branch
- Both user- and kernel-space branches
- ITrace record: branch address, target address, # of instr
- On PowerPC, also load/store address + branch trace
- ITrace control: run.itrace script or libperfutil C/Java APIs
- The post application can produce an arc file

ITrace – An arc File Excerpt




```
0 U 3  1 @ 120 <plt>:/opt/ibm-java2-i386-50/jre/bin/libj9prt23.so 11c1_11c1_java_main JUMP
0 U 3  2 @   0 __libc_write:/lib/libpthread-2.5.so 11c1_11c1_java_main JUMP
0 U 1  1 @  2c __libc_write:/lib/libpthread-2.5.so 11c1_11c1_java_main CALL
0 U 2 19 @   0 __pthread_enable_asynccancel:/lib/libpthread-2.5.so 11c1_11c1_java_main RETURN
0 U 1  7 @  31 __libc_write:/lib/libpthread-2.5.so 11c1_11c1_java_main CALL
0 K 5  1 @   6 no_singlestep:vmlinux 11c1_11c1_java_main OTHER
0 K 5  0 @   0 syscall_trace_entry:vmlinux 11c1_11c1_java_main OTHER
0 K 5  0 @   0 do_syscall_trace:vmlinux 11c1_11c1_java_main OTHER
…
0 U 3  1 @ 19f hellop.main([Ljava/lang/String;)V:JITCODE 11c1_11c1_java_main JUMP
```


Outline

- Introduction
- Time/Event Profiling
- **Callflow Profiling with Calibration**
- Java Callstack Sampling: A Hybrid Approach
- Report Visualization with VPA
- Conclusion

Callflow Profiling

- Why we need callflow profiles?
  - tprof sampling has a very low overhead, but no context info
  - To fine-tune applications, we want to know the value of a metric for a method/function when in a particular calling sequence
- JProf profiler notified by JVM about method entries/exits
  - Via JVMPI or JVMTI
- Gets virtualized thread metrics

  from the Per-Thread Metric Facility (PTM)

Callflow Profiling Issues

- Problem:

  the act of observing a metric in a running application

  almost always changes the behavior of that application

- Solution:

  metric calibration to compensate for overhead

JProf Metrics Calibration

- Calibration is the removal of instrumentation overhead
- "Stable" metrics such as executed instructions

  can be better calibrated
- Internal calibration
  - Eliminates the effect of JProf itself
  - Read metrics after JProf entry (Early Read) and just before the exit (Late Read)
- External calibration
  - Eliminates effects outside of Early/Late Read

JProf Metrics Calibration

- Overhead which must be removed can be computed from the minimum observed change (delta) in the metrics between calls to the profiler
- Each delta: the instrumentation overhead + metric value
- Instrumentation overhead may vary depending on
  - The type of the event (entry or exit)
  - Type of method (native, interpreted or JITed), static or non-static
  - Transition sequence between methods

JProf Metrics Calibration

- We train the profiler by saving the minimum observed values from other profiling runs (*trainer* testcase)
- Minimum observed values – removing too much
- Solution: use ITrace to determine real minimum values
- With calibration, profiling accuracy is nearly identical to that achieved by instruction tracing

Min Number of Instructions Example (J9 JVM)

En-jitted-En-jitted	3	En-Native-Ex-jitted	23
En-jitted-En-Jitted	6	En-Native-Ex-Jitted	4
En-jitted-En-native	28	Ex-jitted-En-jitted	1
En-jitted-En-Native	35	Ex-jitted-En-Jitted	4
En-jitted-Ex-jitted	3	Ex-jitted-En-native	38
En-jitted-Ex-Jitted	3	Ex-jitted-En-Native	29
En-Jitted-En-jitted	3	Ex-jitted-Ex-jitted	1
En-Jitted-En-Jitted	4	Ex-jitted-Ex-Jitted	1
En-Jitted-En-native	28	Ex-Jitted-En-jitted	1
En-Jitted-En-Native	29	Ex-Jitted-En-Jitted	2
En-Jitted-Ex-jitted	3	Ex-Jitted-En-native	38
En-Jitted-Ex-Jitted	3	Ex-Jitted-En-Native	39
En-native-Ex-jitted	4	Ex-Jitted-Ex-jitted	1
En-native-Ex-Jitted	4	Ex-Jitted-Ex-Jitted	1

Profiling Exit/Entry Events in C/C++ Code

- Profiler library hookit sends entry/exit notifications to JProf
- The code to be profiled:
  - Statically linked with libhookit
  - Compiled using the gcc compile option -finstrument-functions
- Currently supported only on x86

Callflow Tracing With JProf: log-gen report

- Produces a trace of method entry/exits,

  with the metric value between the two calls

```
                        …
   1 < J:java/io/OutputStreamWriter.write([CII)V
   3 < J:java/io/BufferedWriter.flushBuffer()V
   2 > J:java/io/OutputStreamWriter.flushBuffer()V
   4 > J:sun/nio/cs/StreamEncoder.flushBuffer()V
  29 > N:java/io/FileOutputStream.writeBytes([BIILjava/io/FileDescriptor;)V
1255 < N:java/io/FileOutputStream.writeBytes([BIILjava/io/FileDescriptor;)V
   1 < J:sun/nio/cs/StreamEncoder.flushBuffer()V
   2 < J:java/io/OutputStreamWriter.flushBuffer()V
 211 < J:java/io/PrintStream.write(Ljava/lang/String;)V
                        …
```

Callflow Tracing With JProf: log-rt Report

- Callflow trees

```
LV CALLS  CEE    BASE DS IN  NAME
 2  1000 1000   11888  1  6  J:hellop.myA()V
 3  1000    0 7004919  0 12  J:hellop.myC()V
 2  1000 1000    9200  0  7  J:java/lang/StringBuffer.<init>()V
 3  1000 1000   48529  0  8  J:java/lang/StringBuffer.<init>(I)V
 4  1000    0    3880  0  5  J:java/lang/Object.<init>()V
 2  1000 2000   37855  1  6  J:java/lang/StringBuffer.append(J)Ljava/lang/StringBuffer;
 3  1000 3000  114912  5 14  J:java/lang/Long.toString(J)Ljava/lang/String;
 4  1000    0  108379  1  5  J:java/lang/Long.stringSize(J)I
 4  1000    0  178282  0  4  J:java/lang/Long.getChars(JI[C)V
 4  1000 1000   10800  0  5  J:java/lang/String.<init>(II[C)V
 5  1000    0    3320  0  2  J:java/lang/Object.<init>()V
```

Object Callflow Tracing With JProf

- Also in log-rt reports

```
LV CALLS    CEE    BASE   DS  IN     AO      AB      LO    LB     NAME
1       1 600006 600038   0   0 100010 2400400 1901 45664 J:hellop.main([Ljava/lang/String;)V
  --                                   5     160     2    64 java/lang/String
  --                                   5     240     2    72 CHAR[]
  --                              100000 2400000 1897 45528 java/lang/StringBuffer
```

Requirements for Per Thread Metrics Support

- Keep separate metrics count for threads of interest
  - Get control just before a thread dispatch,
    read hardware monitoring counters and save values
- Factor out time spent in asynchronous interrupts
  - Want to count synchronous kernel services such as page faults
  - Want to factor out random events on a thread,
    such as I/O interrupts – more repeatable measurements
- Make per-thread metric values available to the profiler
  - ioctl, system calls, or a mapped data area
  - Mapped data – best performance

Per-Thread Metrics Facility

- Per-Thread Metrics Facility in pitrace driver
  - We hook the scheduler and interrupt entries/exits
  - Map a thread work area which contains accumulated metrics, count of dispatches and interrupts
- Better solution: OS provides PTM support
  - Avoids security issues (e.g., an application may monitor only its own threads)
  - perfmon2: an excellent candidate

Outline

- Introduction
- Time/Event Profiling
- Callflow Profiling with Calibration
- **Java Callstack Sampling: A Hybrid Approach**
- Report Visualization with VPA
- Conclusion

Java Callstack Sampling (SCS): A Hybrid Approach

- Java callstack sampling combines the best of:
  - tprof: **low overhead**, but no context
  - Callflow profiling: **full context**, but a large overhead
- tprof + abstract Java stack
  - At a tprof event, notify JProf to take a Java callstack

How Does SCS Compare to tprof?

- Get tree of callstacks
  - tprof gives flat profile
- Advantages
  - Shows how you got to where you sampled
  - Gives visibility to Java code that hasn't been JITed
- Disadvantages
  - Much heavier weight
  - Sampling overhead is not constant

How Does SCS Compare to Callflow Profiling?

- Get tree of callstacks
  - Just like callflow profiling
- Advantages
  - Much lighter weight
  - Deeper stacks (includes in-lined methods)
- Disadvantages
  - Sampling rather than tracing
  - Can’t count calls
  - Lose relative order of method calls

Java SCS: Other Callstack Triggers

- ALLOCATED_BYTES
- MONITOR_WAIT
- MONITOR_WAITED
- MONITOR_CONTENDED_ENTER
- MONITOR_CONTENDED_ENTERED
- IDLE (sample all idle threads every N ms, default 10)

Outline

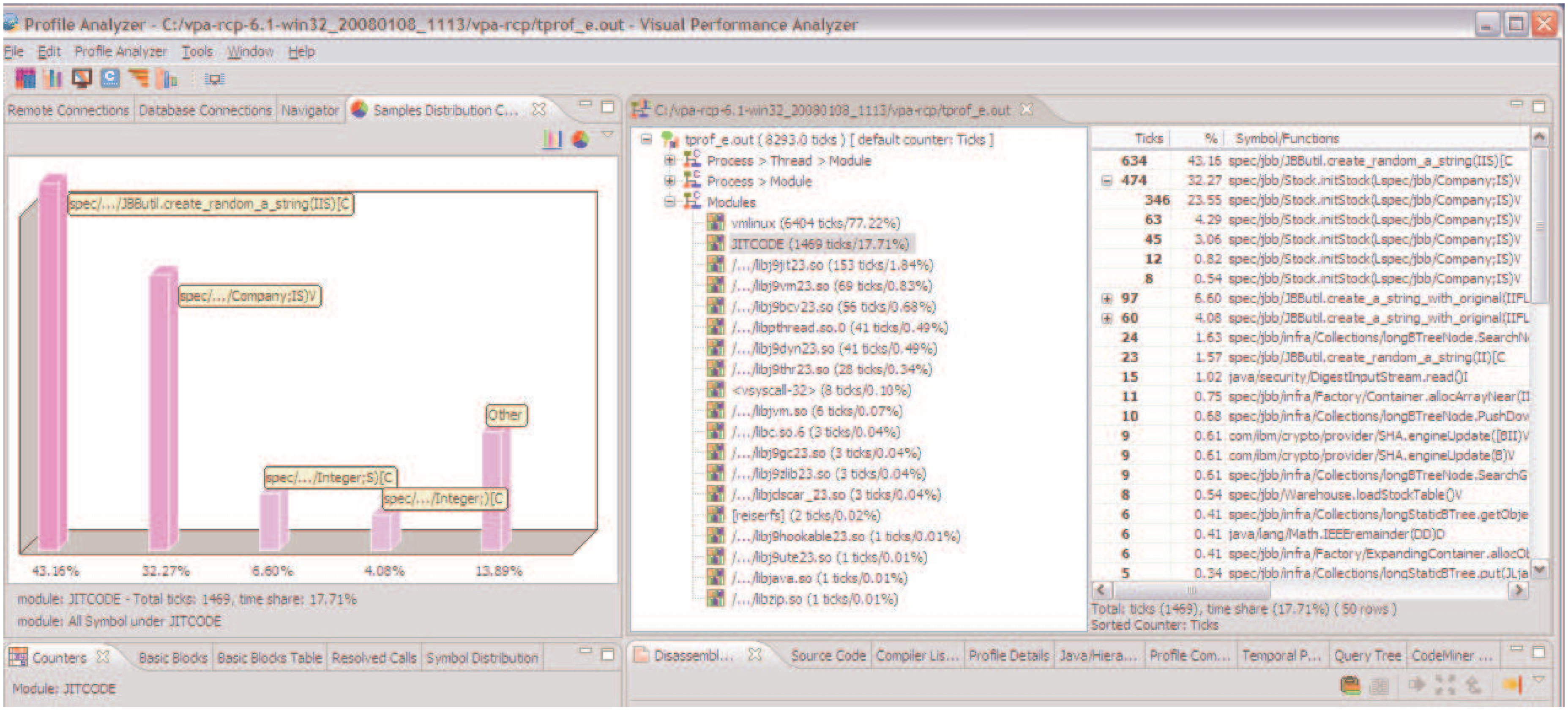
- Introduction
- Time/Event Profiling
- Callflow Profiling with Calibration
- Java Callstack Sampling: A Hybrid Approach
- **Report Visualization with VPA**
- Conclusion

Report Visualization

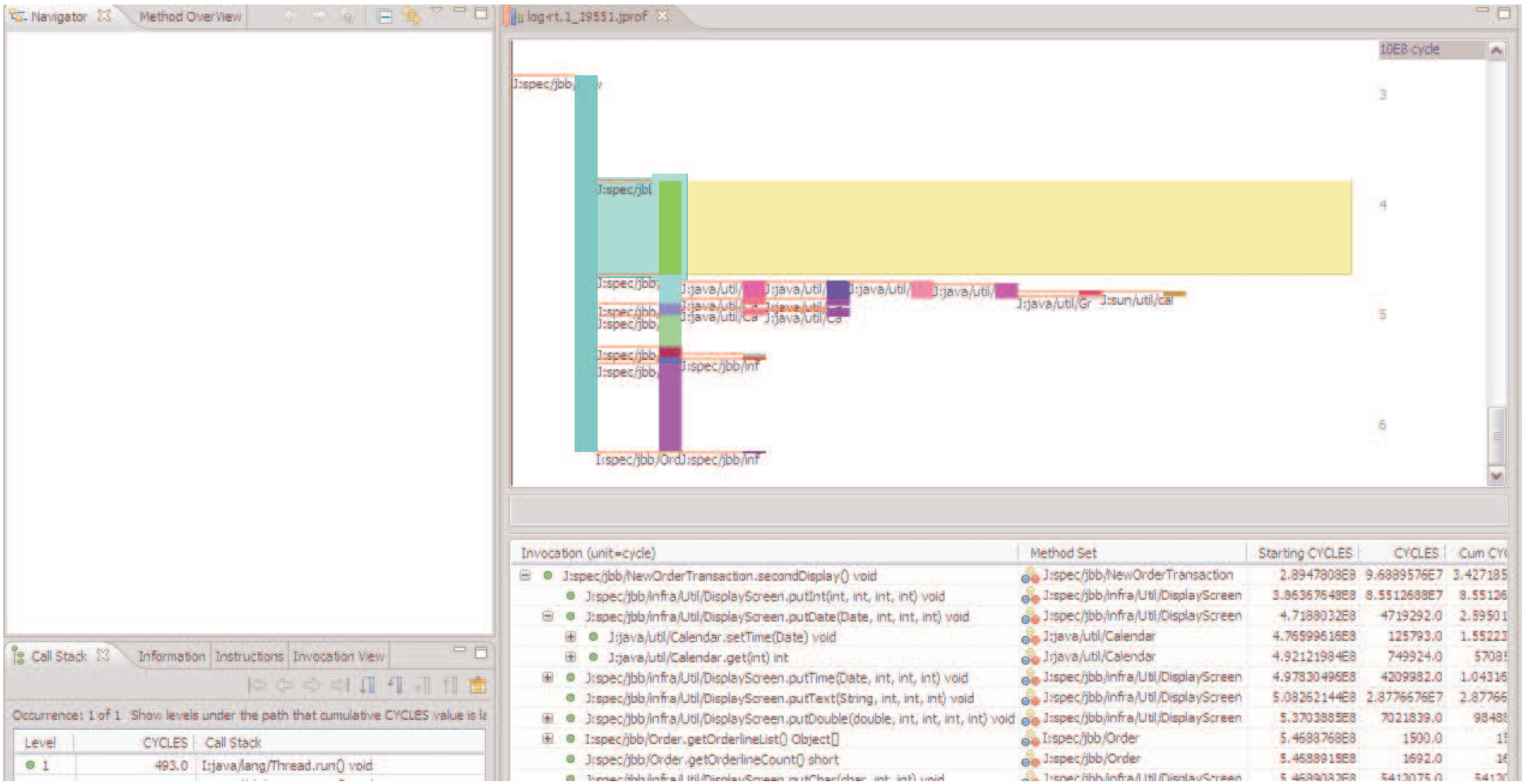
- Visual Performance Analyzer (VPA),

  an Eclipse-based visual performance toolkit
  - Profile Analyzer for tprof reports
  - Call Tree Analyzer for callflow (log-rt) reports

Report Visualization with VPA: tprof



Report Visualization with VPA: Callflow



Outline

- Introduction
- Time/Event Profiling
- Callflow Profiling with Calibration
- Java Callstack Sampling: A Hybrid Approach
- Report Visualization with VPA
- **Conclusion**

Conclusion

- PI provides many unique and useful features
- Accurate profiling of Java/C functions
  - Per-thread metrics virtualization + calibration
- Tracing of dynamically generated code
  - Tracing Facility + symbol resolution
- In the future, try to merge with new in-kernel capabilities
- New ideas & contributions are welcome!

Backup Slides

PI Components

- Kernel driver module – pitrace
- User-space applications & libraries
  - perfutil (libperfutil)
    - A set of APIs for communication with the driver and other utilities
  - JProf (libjprof) – Java profiling agent
    - Capturing execution flow (method call trees or trace)
    - Capturing the state of the Java heap
    - Capturing information about IBM JVM usage of locks via the Java Lock Monitor
    - Resolving Just-In-Time compiled addresses to method names
    - …
  - hookit (libhookit) – enables C/C++ execution flow profiling
  - rtdriver – socket-based interactive control of JProf

PI Components cont'd

- swtrace
  - Control of the Tracing Facility
  - Processor utilization reports
- post & a2n (liba2n)
  - Conversion of binary trace files to readable reports
- ptt - summary per-thread metric counts
- cpi - cycles per instruction for an application/time interval
- msr – access to model-specific registers
- mpevt – manipulation of hardware performance counter events
- hdump – Java Heap analysis
- …

Tracing Facility

- Why yet another tracing mechanism?
  - Provides accurate address-to-name (a2n) symbol resolution of dynamically generated code, such as Java JITed code (kernel knowledge about memory segments + JProf knowledge)
- Two groups of trace records:
  - MTE records (Module Table Entry), needed for a2n resolution
  - All other types (e.g., ITrace, tprof)
- Per-cpu pinned mte and trace buffers
- Tracing modes: normal, wrap-around, continuous

Tracing Facility: How It Works

- Tracing Facility gets task exit and unmap notifications
  - Record the parent tree & mapped executable segments
  - When tracing is turned off, write the rest of MTE data
- JProf writes Java-related information to the trace & log
  - Start, stop, and name for each Java thread
  - Method start address, current thread, and time stamp
  - log-jita2n: address, method name, class name, time stamp, and possibly bytes of instructions

Trace Format

Type/Length	Major Code	Minor Code	Timestamp	Variable Data

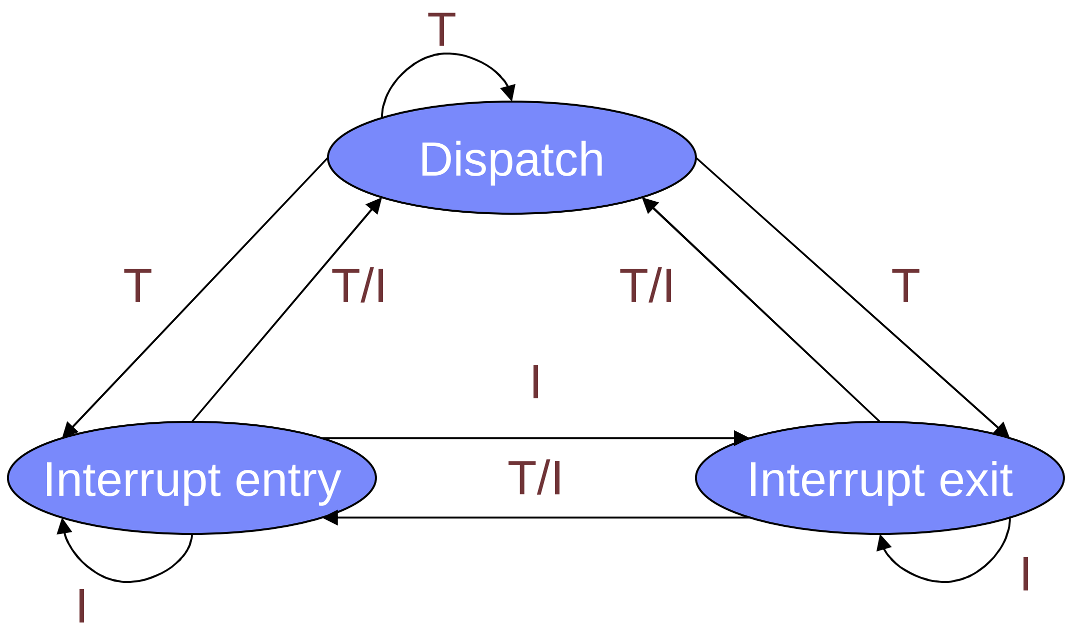
Other Types of Traces

- Trace format can be used for other types of traces
  - Trace of thread dispatches
    (hook at the end of schedule() and ret_from_fork())
  - Trace of interrupt entries and exits
  - Various user-generated trace records

PTT Interfaces

- PttInit() – initializes & starts the PTT Facility, with given metrics
- PttTerminate() – terminates the PTT Facility
- Optimized functions for most frequently used per-thread cases (1 – 2 metrics)
- Functions to get overall stats

PTM State Machine



T – metrics applied to a thread

I – metrics applied to the interrupt bucket

T/I – applied to a thread or the interrupt bucket, depending on interrupt nesting

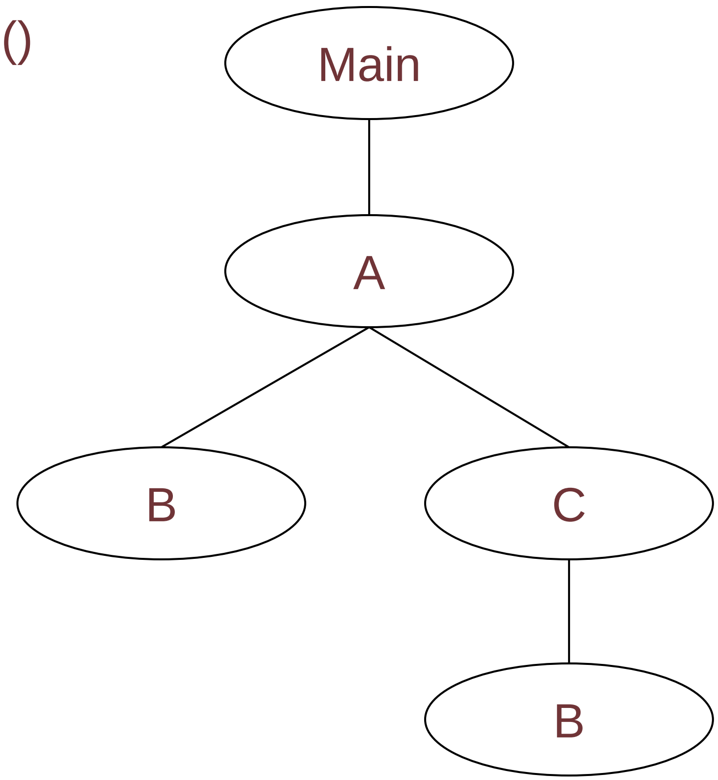
The ptt Application

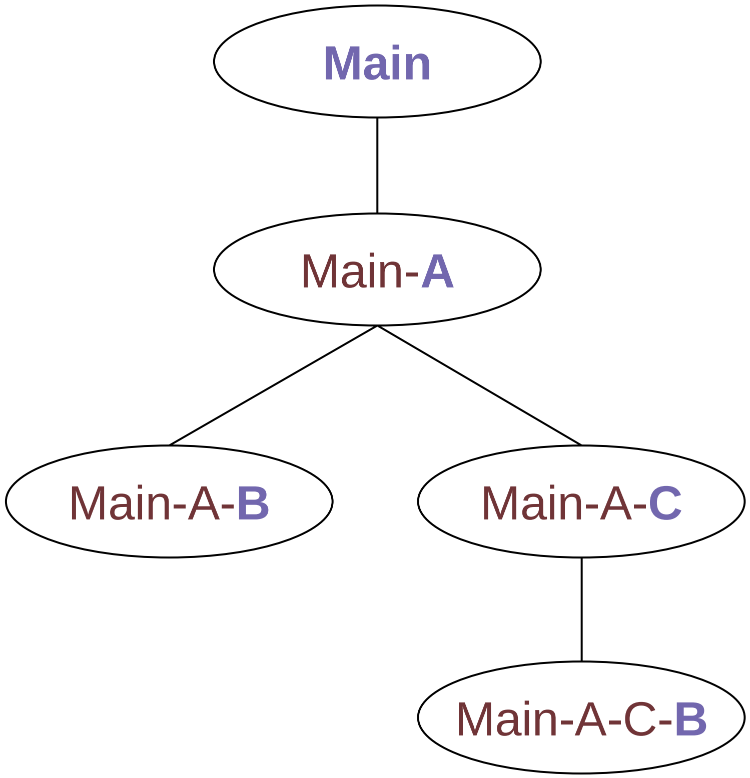
- Turn PTT Facility on/off, read summary information, dump information about threads (all or subsets)

```
  PID    TID      Disp       Intr           INSTR
-----  -----  --------   --------  ----------------
18101  18101       660     283956         698277816  main
18101  18103      1652       2106         306994052  JIT_Compilation_Thread
18101  18113        40         37           2799131  Gc_Slave_Thread
18101  18107       270         67           1038827  **JVM**
18101  18115        53          0            882651  Finalizer_thread
18101  18111        46         34            646395  Gc_Slave_Thread
18101  18112        43        109            271437  Gc_Slave_Thread
18101  18109         1          0            183248  **JVM**
18101  18110         8          9             65003  **JVM**
18101  18114         1          0             38508  RtdriverListenerThread
18101  18108         1         78             32283  Signal_Dispatcher
18101  18102         2         21              9705  **JVM**
                                   ----------------
                                         1011239056
```

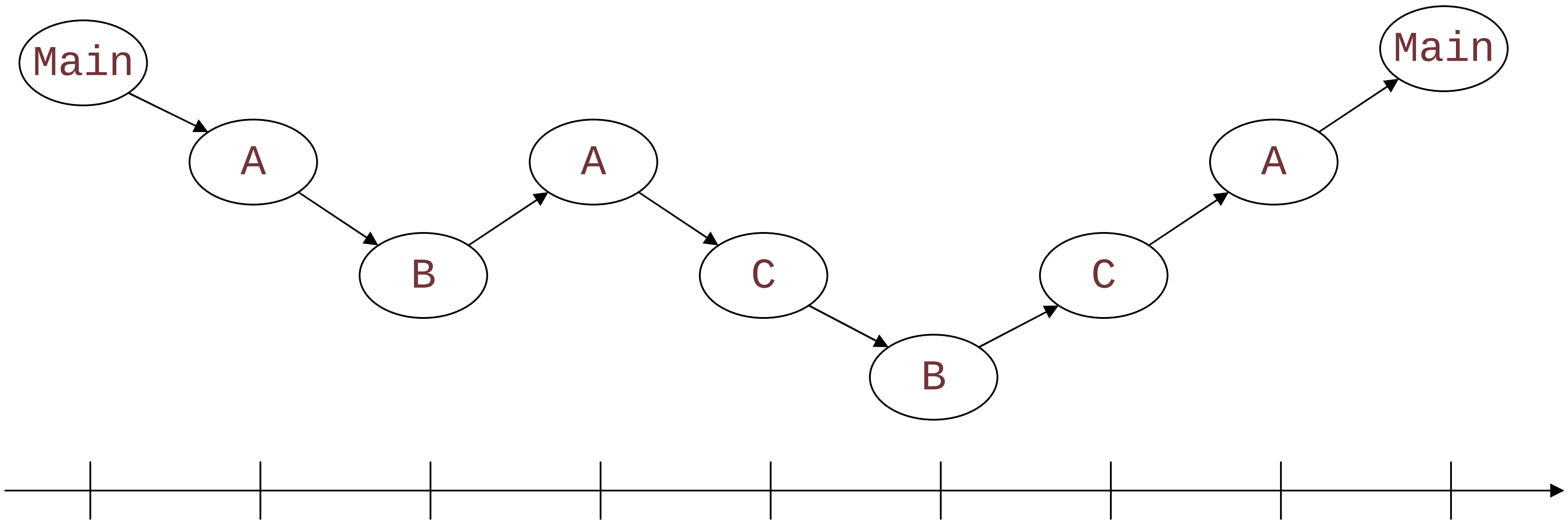
JProf Calibration Details: A Call Tree Example

```
void Main()
{
    A();
}
void A()
{
    B();
    C();
}
void B()
{
}
void C()
{
    B();
}
```





JProf Calibration: Event Timeline & Deltas



SnM	MnA	AnB	BxA	AnC	CnB	BxC	CxA	AxM
-	9	6	2	7	5	3	4	8
-	5	2	1	3	1	2	3	7

Legal Statement

Thanks to Larry Ewing (lewing@isc.tamu.edu) & GIMP for the Tux.

This work represents the view of the authors and does not necessarily represent the view of IBM.

