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

Milena Milenkovic, Scott T. Jones, Frank Levine, Enio Pineda
International Business Machines Corporation
`{mmilenko, stjones, levinef, enio}@us.ibm.com`

Abstract

The open-source Performance Inspector™ project contains a suite of performance analysis tools for Linux® which can be used to help identify performance problems in Java™ and C/C++ applications, as well as help determine how applications interact with the Linux kernel. One such tool is the Per-Thread Time Facility (PTT); it consists of a kernel module and user-space components which maintain thread statistics for time (cycles) or any of a number of predefined metrics. JProf is a Java/C/C++ profiler which uses PTT to produce reports with per-method/function metrics. Another tool is a Tracing Facility, which may be used for tracing instructions, thread dispatches, and sampling events. In this paper we present the details of the most commonly used Performance Inspector tools, targeting the audience of developers interested in performance fine-tuning.

1 Introduction

"I suggest you count your bees, you may find that one of them is missing."

—Inspector Clouseau,
Pink Panther Strikes Again

With growing software complexity, a performance analyst job is becoming increasingly difficult. The Performance Inspector project (PI) consists of a set of tools that helps with analyzing application and system performance on Linux. It includes a kernel driver module (pitrace) and various user-space applications and libraries. The project is hosted on SourceForge (`http://perfinsp.sourceforge.net`). PI currently includes support for the Intel x86, Intel and AMD x86_64, and IBM PowerPC64 and s390 platforms.

The PI toolset enables analysts to identify the overall processor utilization, and application/thread hardware counter summary information. PI provides support for both application and kernel sample-based profiling or instruction tracing. Sample-based profiling without full context information may not give enough analysis information to tune large applications, so PI provides method (Java) or function/subroutine (C/C++) tracing at the application level, relying on built-in Java Virtual Machine (JVM) support for method entries and exits and gcc compile options to generate entry/exit notifications. The Per Thread Time facility, together with the metrics calibration, allows for accurate per-method counts of stable metrics such as *instruction completed*. Because of the repeatability of this metric, it has been used to assign instruction budgets to application components. Utilizing the PI programmatic control of tracing and the consistency of the measurements, changes in component instruction budgets can be identified.

PI encompasses the following user-space components:

- The *libperfutil* library includes a set of APIs for communication with the pitrace driver and other utilities. The JPerf.jar package includes support for the equivalent Java interfaces: for example, you can turn instruction tracing on and off directly from a Java application, thus enabling fine-grain control of the traced code.
- *JProf (libjprof)* is a Java profiling agent that responds to events from either the JVM Profiler Interface (JVMPI) or the JVM Tool Interface (JVMTI), based on the invocation options. Common usages of JProf are:
  - Capturing execution flow in the form of method call trees or a method trace.
  - Resolving Just-In-Time (JIT) compiled code

```
Type and Length | Major Code | Minor Code | TimeStamp | Variable Data

 16 bit         | 16 bit     | 32 bit     | 32 bit    | variable length
```

Figure 1: Trace record format

addresses to method names to support trace post-processing.

- Capturing the state of the Java Heap that can later be processed by the *hdump* PI application to help locate memory leaks.
- Capturing information about IBM™ JVM usage of locks via the Java Lock Monitor (JLM).

• The *rtdriver* application is a socket-based command interface that enables interactive control of JProf, such as when to start or stop of profiling, or when to dump the contents of the Java Heap.

• The *swtrace* application is used to control the Tracing facility. This application is also used to invoke the AboveIdle tool—a lightweight tool which reports processor utilization (busy, idle, and interrupt time).

• The *post* application is used to convert binary trace files to readable reports using the *liba2n* (A2N—address-to-name) library, which converts addresses to symbolic names. This library may be used to convert addresses to names for dynamically generated code, such as the code generated by Java via the Just-In-Time (JIT) support; along with time-stamped tracing, it provides accurate symbolic resolution even when addresses are reused.

• Other tools include *msr*, used to read and write model-specific registers (MSR); *mpevt*, used to manipulate hardware performance counter events; *ptt*, used to give summary per-thread metric counts; and *cpi*, used to measure cycles per instruction (CPI) for an application or a time interval.

The rest of the paper is organized as follows. Section 2 explains the Tracing facility and what kind of information can be traced with it. Section 3 explains the inner workings of the Per-Thread Time Facility which provides per-thread metric virtualization, and Section 4 explains how metrics are adjusted for instrumentation overhead in the JProf profiler. Section 5 briefly describes how users can visualize some of the PI reports, and the last section gives directions for future project development.

2 Tracing Facility

PI includes support for a software tracing mechanism—the Tracing Facility. Although there are already established Linux tools with somewhat similar functionality, our tracing mechanism accurately captures information necessary for address-to-name symbol resolution of dynamically generated code such as Java JITed code. The main issue with the JITed code is that it can be recompiled and moved around the address space. The tracing mechanism combines kernel knowledge about memory segments for a process with JProf jita2n (JITed code address-to-name) synchronizing records and the corresponding dynamically generated code.

We consider two main groups of trace records: one group consists of Module Table Entry (MTE) records and the other group consists of all the other record types, such as ITrace and tprof. Each group uses a per-cpu buffer, i.e., there is an *MTE buffer* and a non-MTE buffer (*trace buffer*) allocated for each CPU in the pitrace driver. These buffers are pinned (allocated in pitrace kernel module) and memory mapped, so that `libperfutil` can read them directly. All trace records have a similar format, shown in Figure 1. The *Type and Length* field specifies the record length and the type of the *Variable Data* field. The *Major Code* field specifies a trace record type, for example, MTE, tprof, or ITrace. The *Minor Code* field specifies a subtype within a type, e.g., a process name MTE record. The TimeStamp field has the lower 32-bits of the time stamp, and a special trace record is written to indicate a change in the higher 32 bits.

Figure 2 shows a block scheme of PI components and files involved when tracing a Java application.

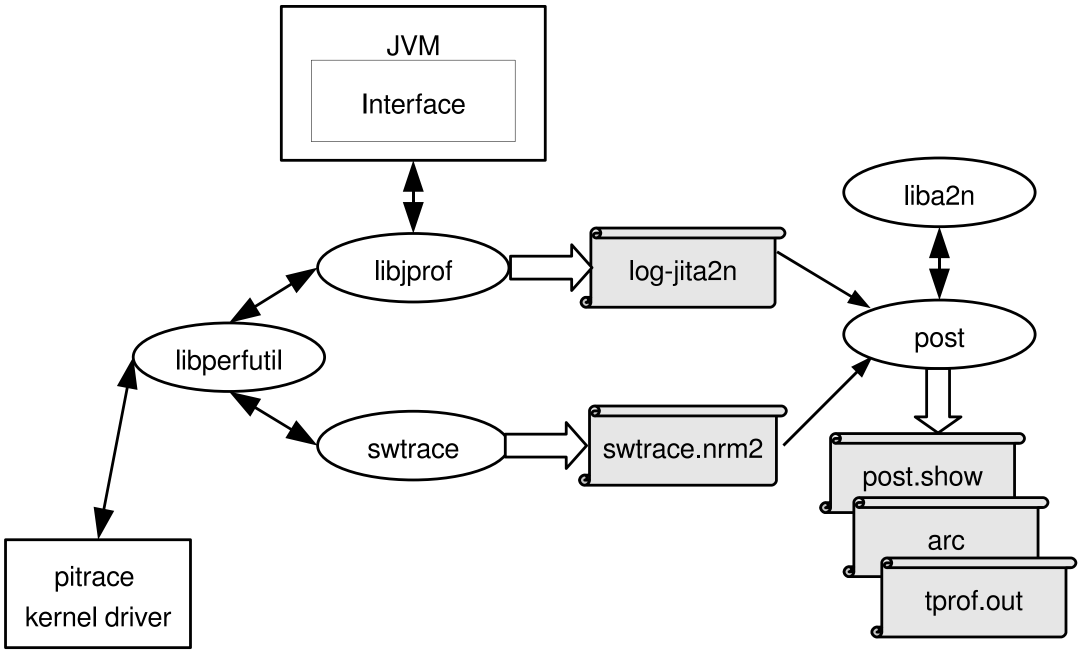

Figure 2: Block scheme for tprof/ITrace tracing of Java application

The `swtrace` application is the front-end which controls the tracing facility via `libperfutil` APIs. It is used to enable tracing of specific record types, specify the size of trace and MTE buffers, turn tracing on and off, write the content of trace buffers to a file, and select the Tracing facility mode. There are three possible modes: *normal*, *wrap-around*, and *continuous*. In the normal mode, tracing automatically stops when either an MTE buffer or a trace buffer becomes full. In the continuous mode, both MTE and trace buffer segments are written to a file when a segment size reaches a given threshold. In the wrap-around mode, meant to be used to analyze application crashes or the most recent application activity, MTE buffers are written continuously in a file, and other trace records wrap around the buffer. The default trace file name is `swtrace.nrm2`.

When initialized and turned on, the Tracing facility gets notifications about each task `exit` and `unmap`, using the existing kernel notification mechanism. When a task exits, we write its parent tree (if not already written), and if the task is not a clone, we also write a trace record for each of its mapped executable memory segments. Similarly, when a memory segment is unmapped, we write the parent tree and executable segment info for the corresponding task. When tracing is turned off, we write previously unwritten MTE data for all tasks still alive.

A trace record write can be initiated from the pitrace module, or from a user application, using the `libperfutil TraceHook()` function. The JProf profiler can use this function to trace start, stop, and name information for each Java thread; it then also writes the same information into a `log-jtnm` file. When a JITed method is loaded, JProf can write trace records with the method start address, current thread, and the current time stamp; it then also writes address to name translation info such as code address, method name, class name, time stamp, and possibly bytes of instructions, into a `log-jita2n` file. This information is used by post to resolve addresses of trace records to the correct Java method, class, and thread name. Post can create an ASCII version of a trace (`post.show`), a *tprof.out* report from `tprof` trace records, and an *arc* report from ITrace records. ITrace and `tprof` tracing mechanisms are explained in more details in the following subsections.

2.1 ITrace

To fully understand a complex performance issue, analysts sometimes need to see a full instruction trace. To get such a trace, we actually need to trace only taken branch instructions, using the underlying hardware support for trap on branch or taken branch. The only issue is that in some earlier 2.6 kernel distributions, trap

```
# arc Field Definition:
#   1: cpu no.
#   2: K(kernel) or U(user)
#   3: last instruction type
#     0=INTRPT, 1=CALL,  2=RETURN, 3=JUMP, 4=IRET, 5=OTHER, 6=UNDEFD, 7=ALLOC
#   4: no. of instructions
#   5: @|? = Call_Flow | pid_tid
#   6: offset (from symbol start)
#   7: symbol:module
#   8: pid_tid_pidname_[threadname]
#   9: last instruction type (string)
#  10: line number (if available)
...
  0 U 3      1 @  120 <plt>:/opt/ibm-java2-i386-50/jre/bin/libj9prt23.so   11c1_11c1_java_main JUMP 0
  0 U 3      2 @    0 __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main JUMP 0
  0 U 1      1 @   2c __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main CALL 0
  0 U 2     19 @    0 __pthread_enable_asynccancel:/lib/libpthread-2.5.so   11c1_11c1_java_main RETURN 0
  0 U 1      7 @   31 __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main CALL 0
  0 K 5      1 @    6 no_singlestep:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @    0 syscall_trace_entry:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @    0 do_syscall_trace:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @   2e do_syscall_trace:vmlinux   11c1_11c1_java_main OTHER 0
...
  0 U 2      1 @   1a java/io/PrintStream.print(Ljava/lang/String;)V:JITCODE 11c1_11c1_java_main RETURN 0
  0 U 3      1 @  19f hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      5 @  1af hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3     11 @   c4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      7 @   d4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      7 @   d4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
...
```

Figure 3: Excerpts from an arc file

on branch flags might not be correctly preserved across interrupts and system calls, so we need to dynamically patch such critical places or to use kprobes.

We call a branch trace *ITrace*. ITrace can include both user- and kernel-space trace records. One ITrace record has addresses of the branch and the branch target, and possibly the number of instructions executed from between the previous and the last branch execution. There are separate major codes for user and kernel addresses. On PowerPC, ITrace records can also include load and store addresses (with different major codes).

The post application can produce an *arc* report from an ITrace and the corresponding `log-jita2n` file. Figure 3 shows excerpts from an arc file obtained from the ITrace of a simple `hellop` application, where `main()` calls `myA()` in a loop and prints a value; `myA()` calls `myC()` which calculates that value. We can follow a write request from JITed code to a JVM library to a system library to the kernel and back. (One arc excerpt in the figure shows the entry to the kernel and the other one shows the exit from `PrintStream.print` to `hellop.main`.)

ITrace can be controlled using the provided `run.itrace` script, or `libperfutil` C or Java interfaces. `run.itrace` is normally used when we do not want to or cannot change the tracing target application; the script asks for the lowest pid to trace. A more controlled ITrace can be obtained by using `ITraceOn()` and `ITraceOff()` interfaces around the section of the code to be traced.

Currently PI does not include support for continuous ITrace. However, we are investigating an algorithm that might enable this feature in future releases.

2.2 Tprof

Tprof trace can be used for system performance analysis. It is based on a sampling technique which encompasses the following steps:

- Interrupt the system periodically if time-based, or when performance-monitoring hardware reaches a given threshold, if event-based.
- Determine the address of the interrupted code along with the process id (pid) and thread id (tid).

```
================================
 )) 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
   SYM     48  1.04       write_chan
   SYM     42  0.91       __copy_to_user_ll
   ...
  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
   SYM     16  0.35       java/lang/Long.getChars(JI[C)V
   SYM     15  0.32       java/io/FileOutputStream.write([BII)V
   SYM     13  0.28       java/lang/StringBuffer.append(Ljava/lang/String;)Ljava/lang/StringBuffer;
   SYM     12  0.26       sun/nio/cs/StreamEncoder.flushBuffer()V
  ...
```

Figure 4: A time-based tprof report excerpt

- Record tprof trace record in a trace buffer.
- Return to the interrupted code.

The detailed steps to obtain a `tprof` trace and a subsequent `tprof.out` report are encapsulated by the *run.tprof* script. This script interacts with the analyst to set up and run necessary steps. Similarly to ITrace, JProf is used to collect the necessary JIT address-to-name information.

The `Tprof.out` report shows percentages of tprof trace records for various granularity, such as for each process, module within a process, or a symbol within a module. Figure 4 shows an excerpt from a time-based `tprof` report for the `hellop` application, for symbols within a module within a thread. Such reports can be used to detect hot-spots in the application or to indicate the resource distribution.

2.3 Other Trace Types

The trace format can easily be used for various types of trace records. In addition to ITrace, tprof, MTE, system information, and time stamp change trace records, the tracing facility currently can produce traces of thread dispatches, and interrupt entries and exits.

3 Per-Thread Time Facility

To accurately determine performance metrics accumulated in an instrumented function, the user-space profiler needs operating system or device driver support for virtualized per-thread metrics (PTM). Such support needs to:

- *Keep separate metrics count for threads of interest.* The PTM code needs to get control when a new thread is about to be dispatched, and to read and save away values of hardware monitoring counters used for metrics.
- *Factor out time spent in external interrupts.* When applications are being monitored, there are some kernel operations that are being done as a direct result of the application code, such as those that require a kernel service. When those services are executed synchronously, it is usually best to include the overhead of the entire code path, including the kernel code path as part of the application because that is how the application will run normally, without instrumentation. For example, we do not want to remove the influence of page faults. However, some events, such as I/O interrupts, tend to occur randomly on a given thread. When trying to produce repeatable measurements, it is helpful to

factor out or separate out the metric counts related to asynchronous interrupts.

- *Make per-thread metric values available to the profiler.* One way to do it is to use a system call or an `ioctl`. However, we can avoid the overhead of system calls using a mapped data area with all necessary information.

The best solution would be to have all these features provided by the operating system. By having the OS monitor the selected threads, we avoid security issues, especially if an application is allowed to monitor only its own threads. The perfmon2 project is an excellent PTM candidate and we eagerly await its full inclusion into the mainstream Linux kernel [1]. In the mean time, we implement the necessary support in the Per-Thread Time Facility (PTT) in the PI driver module, `pitrace`. Note that a more correct name would be Per-Thread Metrics Facility; PTT is a legacy name from the time it supported only per-thread processor cycles. Today PTT can support virtualization of any physical metric provided by the performance monitoring counters.

The `pitrace` module hooks the scheduler close to its return, when the thread to be scheduled is already known. We either use kprobes or dynamically patch the kernel ourselves. To factor out external interrupts, we patch the interrupt entries and exits, so that the time spent in interrupts is accounted for in per-cpu interrupt buckets.

When a request to monitor a thread is made, the driver allocates and maps a thread work area which the application or profiler attached to the application may access directly. The profiler specifies the exact metrics to be monitored and the driver simply reads and accumulates the specified metrics at dispatch and interrupt entry/exit time. Figure 5 shows the PTM state machine. For example, when a previous state was the Dispatch and we are currently entering an interrupt, the metrics delta (difference from the metrics in the last state) should be added to the accumulated thread metrics.

The mapped thread work area contains the accumulated per-thread values and the per-cpu values in the last PTM state. The profiler reads the metrics, calculates the differences from the value of the counters at the time of the last PTM state, and adds those differences to the accumulated values. Since there is a chance that the thread could be dispatched out and back in during the

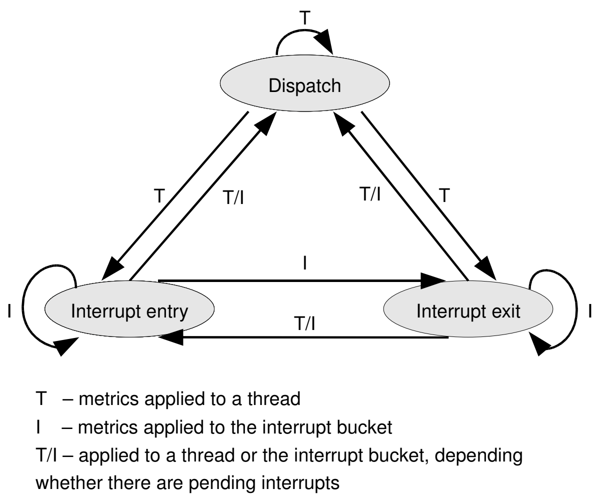
T
Dispatch
T
T/I
T/I
T
I
I
Interrupt entry
T/I
Interrupt exit
I
T – metrics applied to a thread
I – metrics applied to the interrupt bucket
T/I – applied to a thread or the interrupt bucket, depending whether there are pending interrupts

Figure 5: State machine for virtual per-thread metrics

calculations, there should be a simple way for the profiler to determine that this has occurred. One way to provide this feature is to also keep track of the number of dispatches and interrupts in the mapped thread work area. The profiler reads the count of dispatches and interrupts before reading the metrics and reads them again after performing the calculations. If the number of dispatches and interrupts does not change, then the calculated values can be used. If the thread was dispatched or interrupted while reading counters, then the calculations should be repeated until the number of dispatches and interrupts stays the same [2]. Our experiments indicate that this procedure needs to be repeated at most a couple of times.

3.1 PTT Interfaces and `ptt` Application

The `libperfutil` library provides interfaces to the PTT facility in the driver. There are APIs to initialize the PTT facility (`PttInit()`) and to terminate it (`PttTerminate()`). Instead of using a single function to get the current thread metric values, recent packages provide separate functions depending on whether the underlying platform is a uni- or multi-processor system, and on the number and combination of metrics (counters vs. cycles), so that the most frequently used cases of one or two metrics are optimized to reduce overhead. The required function is automatically selected by `libperfutil`, so that the profiler code only needs to set a pointer to it. The interested reader can get more details about available APIs from the package documentation. The maximum number of metrics collected concurrently is eight, regardless of the number of performance monitoring counters available.

One example of PTT facility usage is the `ptt` application, which can turn PTT on and off and dump information about threads for which PTT data is available.

4 JProf Callflow Tracing and Metrics Calibration

Identifying and reporting calling sequences, by receiving notifications on entries and exits to functions or Java methods, is an important methodology that has been shown to be very useful for performance analysis [3].

Based on the invocation options, JProf calls a `libperfutil` function, `PttInit()`, which in turn initializes the PTT facility. JProf receives notifications from the Java Virtual Machine (JVM) about Java method entries and exits, via the JVM Profiler Interface (JVMPI) or the JVM Tool Interface (JVMTI), and it can also query the JVM about the method type and other relevant information.

When an entry or exit event is received, JProf can get the virtualized metrics for the thread on which it is executing. However, the act of observing a metric in a running application almost always changes the behavior of that application in some way. For example, the instructions used to read a metric increase the execution path length of the application and the memory used to store what was read reduces the amount of memory available to the application. That is why the metrics need to be calibrated, that is, adjusted to compensate for the overhead caused by the instrumentation required to observe the metric.

All metrics can be calibrated, but the accuracy of the calibration depends on the stability of the metric being observed. For example, the number of processor cycles required to execute a method is not a stable metric, since it is influenced by a great number of factors such as memory latency, the size of the instruction or data cache, the amount of free memory, asynchronous interrupts, and even the size and complexity of the instructions used by the method. On the other hand, the number of instructions completed is a stable metric, because the number of instructions executed along any given path in the uninstrumented application is fixed. This is why the JProf calibration algorithm is optimized for instructions, although it can be applied to any metric.

The most obvious kind of calibration is performed by merely reading the values of the metrics at entry to and exit from JProf. By doing this, JProf can eliminate its own effects on the metrics between these two reads. We call this *internal calibration*. To maximize the accuracy of the internal calibration, we want to read the metrics as soon as possible after entry to JProf (Early Read) and again at the last possible moment before exit from JProf (Late Read). Another calibration component is *external calibration*—compensation for instrumentation overhead outside of JProf.

In an ideal world, there would be no instructions before the Early Read or after the Late Read, but this is never true. Even the instructions necessary to call the Early Read routine or setup the actual reading of the values are overhead that must be removed.

To achieve successful removal of the JProf portion of external overhead, there are no conditional branches before the Early Read and after the Late Read, to keep the instruction path constant.

4.1 The Calibration Algorithm

The basic assumption on which the calibration algorithm is based is that the overhead which must be removed can be computed from the minimum observed change in the metrics between calls to the profiler. Between any consecutive calls to the profiler, we can compute metric deltas which are the differences in the metric values between those acquired by the Early Read routine from the current call and those acquired by the Late Read routine from the previous call. Each delta includes both the external instrumentation overhead that we want to remove and the actual metric values that we want to keep.

However, the instrumentation overhead may vary depending on the type of the event (entry or exit), type of method (native, interpreted or JITed), and even the transition sequence between methods. The overhead associated with an entry event following an entry event may be different from the overhead associated with an entry event following an exit event, due to optimizations in the so-called *glue code*.

The solution is to maintain an array of minimum observed deltas for each sequence of method types and transition types. We have found that a sequence of three method types and the transitions between them is sufficient for all of the applications we have tested. For example, Interpreted-enters-JITed-enters-Interpreted is one sequence, while JITed-exits-to-JITed-enters-JITed is a different sequence.

Working with the JVM, we have found that there are really three different transition types: entry, exit, and exception-exit. We treat exception-exit (an exit from a routine as a result of an exception) as a unique type, in order to eliminate its influence on statistics gathered for normal exits.

There are also many different method types. We not only consider interpreted and JITed methods, but also native methods. We further distinguish between static and non-static methods for each of these types, since non-static routines require additional glue code to identify the object associated with the method. The last two method types we use are Compiling and Other. Just as we defined a special transition type to isolate the effects of exceptions, we define a special method type for the Java compiler to isolate its effects. Finally, we define an Other type to allow us to isolate the effects of methods whose type can not be accurately identified. This can occur when profiling is started in the middle of executing a method and we lack information about the context in which the method is executing. Thus, we use 8 different method types and 3 different transition types for a total of 8*3*8*3*8 = 4608 different sequences.

Although the categories are still relatively easy to manage, the sheer number of categories introduces other problems. As the number of categories increases, the number of events in each category decreases. This makes it harder to find the true minimum overhead for each category. It also makes it too costly to save counts of all of the different types of sequences with every method.

The solution to both of these problems is to train the profiler by saving the minimum observed values from other profiling runs. This is most effective if the training application generates events in as many valid categories as possible. Some categories will never occur, such as JITed-enters-Native-exits-to-Native, which is invalid because the native method must return to the JITed method which called it. We use a *trainer* Java test case, which is included in the PI package.

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

Table 1: Minimum number of instructions for the most frequently seen transitions. En–method entry, Ex–method exit, lower case–static methods, upper case–non-static methods.

4.2 Environmental Overhead

The calibration algorithm described so far still has one remaining flaw: not all glue code should be associated with instrumentation overhead. Some glue code will be executed even if the application is not being profiled. The calibration algorithm can accurately detect overhead, but it can not determine how much to remove and how much to keep. To do this, the profiler requires specific knowledge about the execution environment when executing applications that have not been instrumented.

The solution to this problem is to execute the trainer application while gathering an instruction trace. By carefully analyzing the results of the instruction trace, a set of minimum values after calibration can be determined. Table 1 shows an example of minimum calibration values for the most frequently seen transitions, for IBM

JVM 5.0 SR5 for 32-bit Linux on Intel platforms. For example, En-jitted-Ex-Jitted means that a method calls a static JITed method which then returns to a non-static JITed method.

Note that we are using only 4 of the types in the 5-type sequences we described. Due to the difficulty of generating every possible combination in the trainer code, we limit the number of the steps to 4. The sequences not covered by data Table 1 will assume a minimum overhead of 1. We do not try to determine values for interpreted methods, because all methods significantly contributing to the overall application profile will be JITed after the initial warm-up.

The calibration algorithm must still be used with these minimum calibration values. The amount of calibration needed can still vary based on the parameters specified during instrumentation, even though the environmental overhead represented by these minimum calibration values remains constant.

By applying the calibration algorithm with an appropriate set of minimum calibration values, profiling accuracy is nearly identical to that achieved by instruction tracing with a fraction of the impact on the execution speed of the application. We validated this approach by comparing a calibrated flow to the instruction trace, for several testcases.

Note that we assume that the instrumentation overhead is constant for a particular transition/type sequence. This is achieved in IBM Java 5. Another concern is that a Java compiler may in-line methods and then may or may not produce entry/exit events for such methods. Disabling in-lining may affect the general overhead of the application. One approach for Java is to simply let in-lining occur as normal and only get the entry/exit events for the methods that are not in-lined. Moreover, a compiler may optimize code and change it in various ways, such as partial in-lining or loop unrolling. These and other optimizations may cause the calibrated metrics to vary from what is expected by examining the code.

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

The same calibration concept can be extended to code written in other programming languages, such as C. JProf needs two additional event categories, for C code entries and exits, so that it can be used for profiling of both standalone C code and code called from Java using the Java Native Interface (JNI). We implemented a prototype profiler library *hookit* which sends entry/exit notifications to JProf. The code to be profiled needs to be compiled using the gcc compile option `-finstrument-functions` and to be statically linked with `libhookit`.

4.4 JProf Callflow Reports

JProf can produce two kinds of callflow reports, depending on the invocation options. More frequently used is a `log-rt` report, which represents methods organized into a call tree, with the number of callers and callees for each method. Figure 6 shows an excerpt of a `log-rt` file for `hellop`. The number of loop iterations was 1000, so both `myA` and `myC` are called 1000 times. The BASE column shows the accumulated number of instructions executed for all 1000 calls.

The other type of callflow report is a `log-gen` file, which has a full callflow trace, with one line for each method entry or exit event, together with the metric value(s) between two successive events. The records in a `log-gen` file are written immediately after a method entry/exit, so the calibration algorithm has to apply whatever is the current minimum delta.

5 Performance Inspector report visualization

Several types of reports produced by PI toolset can be visualized using Visual Performance Analyzer (VPA), which is an Eclipse-based visual performance toolkit [4].

With the help of VPA, users can visualize tprof reports with its Profile Analyzer component, and callflow reports with the Control Flow Analyzer.

6 Conclusion and Future Directions

Although some of PI tools have overlapping functionalities with other Linux utilities or kernel modules, we believe that the project significantly contributes to the always-demanding field of performance analysis, by providing some unique useful features. One such feature is per-thread metrics virtualization which, together

```
   LV CALLS CEE      BASE   DELTA  DS IN  NAME
    2  1000 1000    11888 2378000  1  6  J:hellop.myA()V
    3  1000    0  7004919 1225000  0 12  J:hellop.myC()V
    2  1000 1000     9200 2384000  0  7  J:java/lang/StringBuffer.<init>()V
    3  1000 1000    48529 2382000  0  8  J:java/lang/StringBuffer.<init>(I)V
    4  1000    0     3880 1221000  0  5  J:java/lang/Object.<init>()V
    2  1000 2000    37855 3543000  1  6  J:java/lang/StringBuffer.append(J)Ljava/lang/StringBuffer;
    3  1000 3000   114912 4709000  5 14  J:java/lang/Long.toString(J)Ljava/lang/String;
    4  1000    0   108379 1225000  1  5  J:java/lang/Long.stringSize(J)I
    4  1000    0   178282 1225000  0  4  J:java/lang/Long.getChars(JI[C)V
    4  1000 1000    10800 2384000  0  5  J:java/lang/String.<init>(II[C)V
    5  1000    0     3320 1221000  0  2  J:java/lang/Object.<init>()V

 Column Labels:
  : LV    = Level of nesting       (Call Depth)
  : CALLS = Calls to this method   (Callers)
  : CEE   = Calls from this method (Callees)
  : BASE  = Metrics observed
  : DELTA = BASE adjustment due to calibration
  : DS    = Dispatches observed
  : IN    = Interrupts observed
  : NAME  = Name of Method or Thread
```

Figure 6: A log-rt report excerpt with the instruction completed metric

with the metrics calibration mechanism, enables accurate profiling of Java methods or C functions. Also useful is the combination of the Tracing Facility and address-to-name resolution mechanism, which results in correct trace interpretation for dynamically generated code.

We constantly add support for new hardware platforms, and we will continue to do so in the future. Current tools support a limited set of hardware performance counter events, but this set can be easily extended by adding new events to per-platform event description files.

We also strive to support new Linux releases. The sensitivity to kernel changes would be significantly reduced if we could build on the top of mechanisms integrated with the mainline Linux kernel. In the future, we might be able to use perfmon2 for per-thread performance counter virtualization [1]. It would be nice to merge the Tracing facility with some of the on-going tracing efforts, such as the Driver Tracing Infrastructure [5].

We maintain a long wish list of useful additions to the PI project, such as the capability for continuous ITrace. As always, new ideas and contributions are welcome.

Legal Statement

References

[1] *perfmon2: the hardware-based performance monitoring interface for Linux*, `http://perfmon2.sourceforge.net/`

[2] R.F. Berry, et al., *Method and system for low-overhead measurement of per-thread performance information in a multithreaded environment*, US Patent No. 6658654, 2003.

[3] W.P. Alexander, R.F. Berry, F.E. Levine, R.J. Urquhart, *A unifying approach to performance analysis in the Java environment*, IBM Systems Journal, Vol. 39, Nov. 1, 2000, pp. 118–134.

[4] *Visual Performance Analyzer*, `http://www.alphaworks.ibm.com/tech/vpa`

[5] D. Wilder, *Unified Driver Tracing Infrastructure*, Linux Symposium, Ottawa, Canada, 2007.

Proceedings of the Linux Symposium

Volume Two

July 23rd–26th, 2008
Ottawa, Ontario
Canada