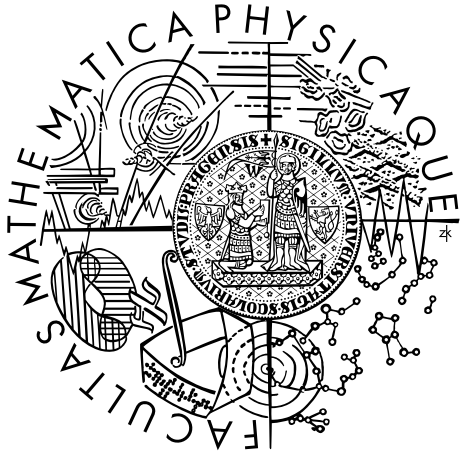
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Peter Libič

Integrating Profiler Data

Department of Software Engineering

Supervisor: Doc. Ing. Petr Tůma, Dr.

Study Program: Computer Science, Software Systems

2008

I would like to thank my supervisor, *Doc. Ing. Petr Tůma, Dr.*, for his help with elaborating this work, for his time, advices, feedback and great amount of his sufferancy of my unannounced visits that usualy catched him just before deadlines. Next I would like to thank *Ing. Lubomír Bulej, Ph.D.* for sacrifice of his desktop for tests of the black magic produced in this work. I am grateful to all my friends for their help and support in the times when the work used to do nothing but crashing the computer. I also want to thank my parents and family for their endless support. Without that this work could never even start.

I declare that I have elaborated this thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prague, 18th of April, 2008 Peter Libič

Contents

Název práce: Integrování profilovacích dat
Autor: Peter Libič
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@dsrg.mff.cuni.cz

Abstrakt: Výkon je jeden z důležitých aspektů softwarových aplikací. Vyhledávání problematických oblastí kódu se stává stále náročnější, protože složitost různých částí počítačových systémů rychle roste. Na usnadnění tohoto hledání existují nástroje nazývané profilery. Profiler by měl ideálně poskytovat data ze všech úrovní systému - od hardwaru přes virtuální stroje interpretovaných nebo částečně kompilovaných jazyků až po samotnou aplikaci. Nástroje pro jazyky, které nejsou kompilované do nativního kódu, jako je Java, bohužel neposkytují možnosti pro zjišťování dat o výkonu z nižších vrstev. Cílem této práce je vytvořit profiler schopný měřit údaje jak z nativních tak i z Java aplikací s podporou vlastností moderních profilerů nativních programů, jako je možnost inicializace získávání dat na základě konfigurovatelných hardwarových udalostí.

Klíčová slova: profiler, Java, hardwarové události

Title: Integrating Profiler Data
Author: Peter Libič
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@dsrg.mff.cuni.cz

Abstract: Performance is one of the important aspects of software applications. With growing complexity of different parts of computer systems, the search for problematic code areas is getting more and more difficult. To help in this task, tools called profilers are available. Ideally, a profiler would provide data from all layers of the system - from hardware through virtual machines of interpreted or partially compiled languages to the application itself. Unfortunately, profilers for languages that are not compiled to native code, such as Java, do not provide facilities to read data from lower levels. The goal of this work is to devise a profiler capable of profiling both Java and native code that would support modern features of native profilers such as triggering on configurable hardware events.

Keywords: profiler, Java, hardware events

Chapter 1

Introduction

System performance is an important aspect of many software applications. With the growing complexity of software, it is becoming increasingly difficult to investigate performance issues. One of the tools that help investigating performance issues are *profilers*. They measure performance data from running applications. They can also help identify the reason why the code is running slowly. As mentioned in [4], it is important to have data from all levels of system – from application, virtual machine, operating system and hardware. Unfortunately, the support for profiling all levels of a system is often not available when partially interpreted or just in time compiled environments and virtual machines need to be considered.

The goal of the thesis is to provide a tool that is capable of profiling both interpreted and native applications (C++, Java), and that would support modern features of native profilers such as triggering on configurable hardware events.

1.1 Structure of the thesis

We will discuss selected profilers and tools for performance analysis in *Chapter 2*. We will also describe some aspects that are not easily available for these technologies. In the *Chapter 3* we will describe our approach to the solution of problems and in the *Chapter 4* we will provide technical details about implementation of tool that was created as a part of this work. We will measure the overhead of the tool in the *Chapter 5* and summarize the work in the *Chapter 6*. The appendix provides installation and usage manual.

Chapter 2

Existing technologies

An overview of existing technologies involving performance analysis is introduced in this chapter. This overview has two goals – to provide outline to the methods of performance analysis and to summarize our studies of that technologies, because some information about them are not available. First, we discuss profilers, then software that can be used to create profiling tools. In the end of the chapter we pu our mind to the technologies available for data integration.

2.1 Profiling

In performance analysis of applications the most common tools are profilers. Their name comes from the result of their work - they generate a profile of the application (or trace of events). It usually contains code position at which occurred observed events and their count. The developer can determine problematic places in his application from that data.

The profilers can be separated into two basic groups:

- Statistical sampling profilers – they use some source of regular events and records data valid in the time the of the event occurrence.
- Event-based profilers – they usually instrument some important areas of application to generate events. Then if the event is triggered the tool adds data to trace.

A brief overview of the existing profiling tools is provided in this section, with technical details on the profiling mechanisms. We do not provide information about many

profilers that are commonly used, for example Intel V-Tune [7], that uses mechanism similar to OProfile and HPROF [12] that uses the feature provided by JVMTI library (detailed description is in *Section 2.2.3*) to profile Java applications. It uses both statistical sampling and sometimes event-based profiling that uses JVMTI callback feature.

This section begins with overview of CPU performance counters, since they are used in the tools described next and we make an extensive use of them. Then we provide overview of OProfile profiler, with detailed description of some parts of its internals. Then we describe an available solution to a problem similar to ours and we mention a representative of profilers using instrumentation – GProf.

2.1.1 CPU hardware performance monitoring counters

All modern microprocessor implementations provide performance measurement support [6, 1]. This support is implemented using performance monitoring counters. A processor typically provides a list of many processor events, such as instruction completion, cache miss, memory access, etc. It also provides few (2-4) registers where these events can be counted. The counted events can be set to generate interrupt if the counter value overflows. The performance counters are essential for these work, therefore we will provide overview of the performance counters system for particular processor family and overview of the APIC part that is involved in generating counter overflow interrupts here.

2.1.1.1 Intel P6 processors family performance counters

We decided to describe the P6 family of the Intel processors, because they are the most common nowadays. The processors Pentium Pro, Pentium II, Pentium III, Core and Core 2 belongs here. The architecture of the performance counters in AMD processors is very similar.

Intel P6 processors provide 2 counters (4 for AMD). The mechanism is separated into two parts, controlling machine-specific registers (MSRs), called *PerfEvtSel0* and *PerfEvtSel1* and performance counters MSRs (*PerfCtr0* and *PerfCtr1*). These registers can be read and write by RDMSR and WRMSR instructions. These instructions are available only in privilege level 0, kernelspace. The performance counters can be read from userspace by RDPMC instruction. This instruction can be restricted to the privilege level 0 by clearing PCE bit in CR4 control register.

In the *Figure 2.1* the layout of the *PerfEvtSel0* and *PerfEvtSel1* MSRs is shown. The description of interesting fields and flags follows:

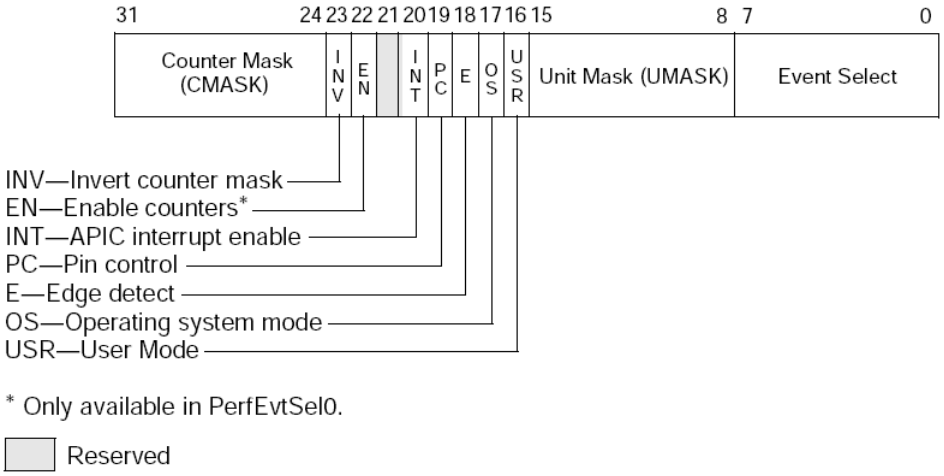


Figure 2.1: Event selection registers, from [6, p. 117 (18-115)]

- Event select field – identification of the event to measure. Events are specified in processors' documentation.
- Unit mask field – enables to select which sub-events of the event increments the counter.
- User mode flag – enables measurement in privilege levels 1, 2 and 3.
- Operating system mode flag – enables measurement in privilege level 0.
- APIC interrupt enable flag – enables the interrupts on counter overflows.
- Enable counters flag – available only in *PerfEvtSel0*, enables and disables measurement in both counters. The second counter only can be disabled by clearing *PerfEvtSel1* MSR.

The performance counters are 40 bits (48 bits on AMD) wide MSRs. The WRMSR instruction can write any value to the lower-order 32 bits, and the higher order 8 bits are sign-extended according to the value of bit 31. This means that both positive and negative values can be written to the performance counter. This is used with advantage for generating performance traces using overflow interrupt feature. The driver writes the negative number to the counter and after triggering absolute value of that number of events, the counter overflows and generates overflow interrupt (if set).

2.1.1.2 APIC and performance counter overflow interrupts

In order to receive interrupts at counter overflows the counters must be configured to count events and APIC must be set up to send interrupts. The performance counter overflow interrupt is a *local interrupt source*. Local interrupt sources have special registers in APIC (mapped to memory) to configure them. For all local sources, the structure called *Local Vector Table* (LVT), is present. It consists of five 32-bit registers. The table with individual fields is displayed in the *Figure 2.2*. For the performance counter interrupt the following fields are available:

- Vector – the number of the interrupt vector that will be generated at the event.
- Delivery mode – the type of the interrupt that will be created – for performance counter interrupts are meaningfull *fixed* (000) and *NMI* (100) modes.
- Delivery status (read only) – indicates if no interrupt activity is present (0), or the interrupt was sent to processor core, but was not accepted yet (1).
- Mask – enables (0) or inhibits (1) the interrupt. Some processors set the mask bit after every performance counter interrupt and the bit must be reset by software.

The writes to the table are executed as memory writes, at specified address that can be configured. The default value for performance counter interrupt table entry is 0xFEE00340. In the Linux kernel the writes to the APIC registers can be done using apic_write() function. To enable NMI interrupt for performance counter overflows, following code can be used:

```
apic_write(APIC_LVTPC, APIC_DM_NMI);
```

Full detail about APIC can be found in [5, Chapter 8].

2.1.2 OProfile

OProfile [14] is an open source system-wide profiling tool that runs on Linux systems. It belongs to the family of statistical sampling profilers. For sampling it uses hardware timer or CPU performance counters, which generate interrupts. It stores the actual position in executing code, yielding set of statistical samples on observed locations in executing program. Important feature of OProfile is the ability to store position not only in single application, but in the entire executing system. It has low overhead and it is not intrusive in the sense that it does not require any code instrumentation. It uses the hardware performance counters as default option on the

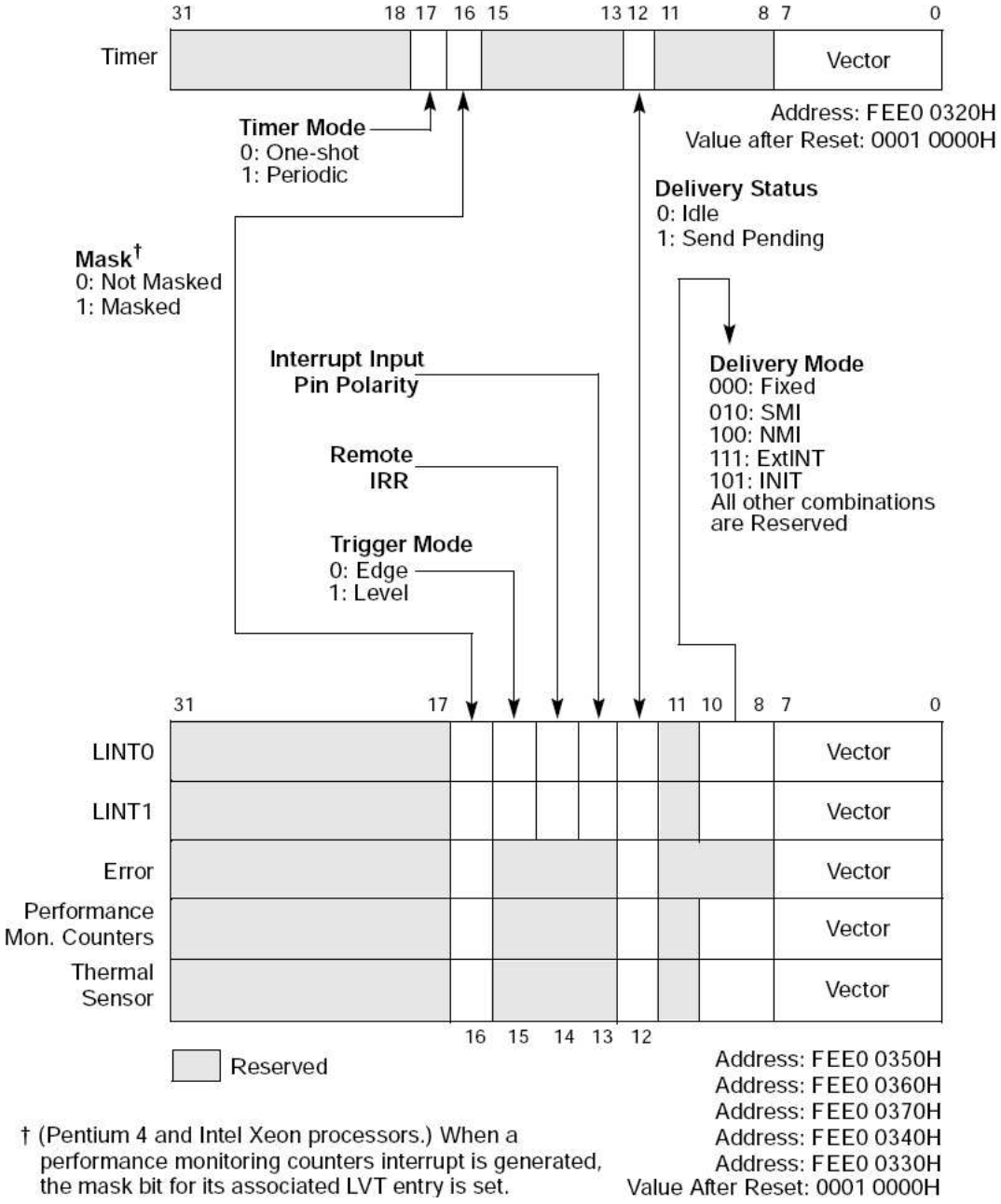


Figure 2.2: Intel APIC Local Vector Table, from [5, p. 347 (8-17)]

systems that support them. In this chapter, we will first provide a basic overview of the tool. It will be followes with a more detailed description of the machine-specific part for the x86 architecture, since the OProfile documentation does not provide detailed information about this component of the system.

2.1.2.1 Overview

The profiler consists of two parts: a kernel driver and a userspace daemon. The kernel driver is responsible for generating and storing performance events, the userspace daemon is responsible for persistent data storage. The profiler also includes some post-profiling tools to generate output.

OProfile has two modes of operation. The first is the timer interrupt mode, where the performance events are generated regularly by the system timer interrupt. In the second mode, the CPU performance counters are used. In this mode, the performance events are generated from the counter overflows. After interception of the performance event, the kernel driver recognizes the instruction that was interrupted (program counter – PC) and stores its address in a special per-processor buffer. In this buffer, information about task switches is also stored, which means that for every event, an information on what instruction address it happened and in which task it was is available. Later, these buffers are processed, and from the absolute PC and the task, the profiler determines into which binary the instruction belongs. This uses the fact that binaries are mmap-ed to the memory. Then the offset in the binary is determined and the pair <binary,offset> is stored to a larger buffer that can be passed to the userspace. The post-profiling tools may then easily determine the function where the event was triggered from the binary symbol table (if present) .

In the result, OProfile is a statistical profiler with very accurate profiles at low overhead. The overhead is between 1–8%, depending on the event frequency [14].

To use the oprofile, the `opcontrol` and `opreport` programs are available. `opcontrol` starts and stops profiling, selects performance events to measure and manages the userspace daemon. The `opreport` program generates the output for the user. See section “Docs” in [14] for additional information.

2.1.2.2 x86 performance counters backend

OProfile has very good internals documentation in [10]. However, this does not cover the machine-specific parts of the kernel driver very thoroughly. Later in this work, we use and modify this part, so we decided to provide a more detailed description

for the x86 architecture backend of the OProfile kernel driver.

OProfile defines structure oprofile_operations , that is filled by the architecture-specific implementation with functions that control the data collection and event generation, like start, stop, setup and shutdown. In the x86 backend these functions are for the performance counter events defined in the arch/x86/oprofile/nmi_int.c file.

In the x86 systems the OProfile utilizes the non-maskable interrupts (NMI) for performance counter overflow notification. This has the advantage of producing more exact traces, because the events can be generated in places where the normal (fixed) interrupt is masked. Such places are spinlocks in the kernel, for example. The Linux provides some so called notifier chains, where the driver can register its callback for the specified event. For NMI, the die_chain is available – the callback can be registered with register_die_notifier () function. OProfile has registered its own notifier with profile_exceptions_notify () callback. This calls the model-specific handler check_ctrs() that has to determine the overflowed counter and add new sample to the OProfile buffers, if successful.

Since performance counters implementation differs between processor families, OProfile has a generic structure, op_x86_model_spec, that encapsulates the counter operations as setup the counter, check for overflow, enable and disable the counter. There are three different implementations for different processors, in the arch/x86/oprofile/op_model_ppro.c file the one for Intel processors except Pentium 4 based, in the arch/x86/oprofile/op_model_p4.c file for Intel Pentium 4 based chips and arch/x86/oprofile/op_athlon_athlon.c for all AMD processors. In the context of this work, the check_ctrs operation is the most important. For all models, it traverses the enabled events and checks if the counter is overflowed. In that case, it calls the oprofile_add_sample() function that creates the trace data and resets the counter to get ready for next event.

Since the performance counters are implemented as MSRs, the model implementation must manage the access to these registers. The Linux provides functions rdmsr() and wrmsr() to read and write to these registers. They require the addresses which are generated in model operation fill_in_addresses . Every model implementation also provides set of macros to read or write counter values and controlling options. They use the rdmsr() and wrmsr() functions. For the counter reset values, OProfile uses the unsigned long integers. However, the counter overflow limit can be set only as the maximum integer value, since the writes to the registers are only 32 bits wide operations. The counter writing macros (CTR_WRITE usually) also manipulates the values to be appropriate for the counter (makes the negative from it). It also provides CTR_READ macros for reading the values.

The Intel Pentium 4 based processors have the performance counter model much

more complex and this work does not cover that chips, but the basic principles of the OProfile model implementation are the same.

2.1.3 Java vertical profiling

In [19] the description how to get the hardware performance monitoring counter values to the profile of the Java is introduced. The authors use approach that modifies the Jikes RVM virtual machine [8] in its thread scheduler. There it reads the values of the processor counters and writes them to the file. A visualisation tool that can interpret the measured data is also availale. They also described some performance anomalies and hypothesised the reason for that anomalies. In [4] they extended the infrastructure for tracing events from the VM and proofed their hypothesis. This shows that vertical profiling may be helpful in identifying performance issues.

2.1.4 GNU gprof

The GNU gprof [3] is a profiler developed by GNU as a part of the GNU compiler suite. We mention it here as a representant of profiler family that uses instrumentation of code. It requires the compiler cooperation, the modules must be compiled with the `-pg` and `-g` options. It is not necessary to compile all modules with `-pg` option, but such modules will not have any profiling information, except the number of calls to that module. In addition, the program can be linked with libc_p.a library by writing `-lc_p` to the linker, instead of standard `-lc`, to count calls to the standard C library. In addition to the `-pg` option the modules can be compiled with `-a` option to count basic-blocks executions. That means if-statement branches, loop iterations, etc.

Then if the program instrumented by the `-pg` options runs, it creates the profile file gmon.out where it stores function (and basic blocks with `-a`) call count and statistical profile – every 0.01 second it records the current executing function. The profile file is generated only on clean exit from the program. After the program execution the profile can be inspected using the `gprof` utility.

A significant disadvantage of the tool is that it is intrusive and requires compiler support. It is very difficult to build an instrumented version of the program if the Makefile is not ready for that option. This is probably the reason of its minimal use.

2.2 Profiling tools

Since none of the presented profilers (even none that we would know) support Java in a way described in *Chapter 1* and it is obvious that implementing a new profiling mechanism will be needed, tools that would be useful in such an implementation – querying system state, querying performance counters, triggering on performance counters, querying virtual machine state – are presented.

2.2.1 perfctr linux kernel extension

The Linux performance monitoring counters kernel extension, called also perfctr patch [16] is an extension to the Linux kernel that makes the CPU hardware performance monitoring counters available to the userspace applications. It can be configured to measure the global values, and per-thread counter values. It can also send signal at the overflow of the counters.

The patch adds new fields for performance monitoring to the kernel task structure. If the counters are enabled for the thread, it initializes the data structures and at every context switch to and from that task it updates the values for the counters according to the values in the hardware. The user can read the values by system call, implemented by special files and ioctl function calls. The patch also introduces the new interrupt handler to the Linux kernel. It is used for performance counter overflow. In this case it sends the signal to the userspace thread that was interrupted.

The patch has nice features, however it lacks documentation. There is no documentation available for the userspace library, except examples in the distribution. This make it very difficult to use, anyway some ideas in the kernel part proved to be useful for this work.

2.2.2 PAPI library

The Performance Application Programming Interface, PAPI for short, is a portable library that exposes a consistent interface for access to the hardware performance counters in various processors [15]. It provides two function layers: high level and low level functions. The high level interface provides simple-to-use functions for measurement of instructions per cycle (IPC), Mflops/s (floating point operation rate), Mflips/s (floating point instruction rate) and simple reading and accumulating of the counters. The low level interface provides more detailed access to the counters and other features. The most important are the functions to inspect the available events, set-up a callback for the overflow, create a program counter histogram, make

available the multiplexing of the performance counters and system state information. The counter multiplexing means using one counter for more events, with accuracy drop. To the state information belongs memory information, executable and library information, etc.

On the Linux/x86 architecture, the library is implemented with the perfctr patch. The problem of the library is that it only counts events in the threads that were registered explicitly with PAPI and were created after initialization of the library.

2.2.3 JVM tool interface

The Java Virtual Machine Tool Interface (JVMTI) is an API provided by JVM to allow using and creating third party tools for debugging, profiling and monitoring. The interface provides means for the tool to be notified of events and status changes in the JVM (callback) and means for inspecting and controlling the virtual machine. The tools take the form of agents written in native (C/C++) code that can be attached to the virtual machine from the command line at JVM start-up. The complete reference to the interface can be found in [9].

An agent is a standalone shared library that exports one required symbol – the Agent_OnLoad() function that is invoked by the execution environment when the shared library is loaded. The agent may optionally export the Agent_OnUnLoad function that is invoked immediately before the library is removed from memory.

There are 5 execution phases that determine the functions and events available for the agent:

- OnLoad phase – while in the Agent_OnLoad() function
- Primordial phase – after OnLoad phase and before the VM start-up
- Start phase – while in VM start-up and before VM initialisation
- Live phase – while the VM is running a program
- Dead phase – after the program termination or after start-up failure

The interface defines the set of capabilities which must be set to make particular functions or events available. This must be done during the OnLoad phase. Besides this, the Agent_OnLoad() function is suitable for only minimal initialisation, because very few functions are available in the OnLoad phase – only capabilities setting, parameter parsing, event notification setup and JVMTI environment creation is usually done here.

Since JVMTI is the only method of getting relevant information on program state from JVM, we make an extensive use of it. We will therefore describe the event callbacks mechanism, important functions and basic agent usage in this chapter.

2.2.3.1 Event callbacks

The agent can be notified of many events. This is the mean to get informed about actions that are important for the agent. To enable an event notification, the agent must fill the set of callbacks to the jvmtiEventCallbacks structure and pass it to the SetEventCallbacks() function. By default, all events are disabled. To enable them, or to disable the already enabled events, the agent calls the SetEventNotificationMode() function.

When the VM encounters an action that matches a selected event, it calls the callback function. There is no event queuing so the callback functions should be re-entrant. To some events, parameters that are the references to the JVM objects, are passed. They are *local references* and they are not valid after the return from the callback.

The events are delivered on the thread that caused them. The events do not change the execution status of the thread. Every event has specified phases in which it can be triggered. If the code runs to a point where the event would be generated, but the VM is in a phase where the event is not allowed, it is ignored.

The most important events for this work are the Method Entry and Method Exit events that are triggered on every method entry and exit, Thread Start and Thread End that notifies about thread creation and termination and VM Init and VM Death that are triggered at the Java program start-up or termination. Other interesting events are Fields Access, Exception, Monitor Wait and Monitor Waited.

2.2.3.2 Control and inspection functions

Along with event notification, JVMTI provides a set of functions that can inspect the VM state or alter it. In the context of this work, the thread management and inspection functions, the stack frame information functions and the method functions are interesting. Thread functions allow to get the information about the thread, such as its name and execution status, to monitor information and change the status by actions such as thread suspension and resume. A function that starts a daemon thread also belongs to this group of functions.. The agent should use this function to start its threads. The stack frame functions allow to inspect the position of the Java code currently running in the VM. The method functions provide information

about a method such as its name, arguments, local variables and modifiers.

Other functions in JVMTI allow to inspect and alter the local variables, enforce garbage collection and get information about it, classes information, bytecode manipulation, heap status etc.

2.2.3.3 Usage

A comprehensive introduction to writing JVMTI agents can be found in [17]. Here, we will describe the very basics of agent programming and instructions how to run Java program with the agent attached.

To run the program with the agent, the user adds the `-agentlib:<library_name>` or `-agentpath:<path_to_library>` command line parameter to the `java` command. With the `-agentlib` argument, the agent is searched for in the system library search path, and the system library naming conventions are used. This means that on Linux, the `-agentlib:example` argument searches for shared library named libexample.so in the directories specified by the `LD_LIBRARY_PATH` environment variable. The `-agentpath` argument searches for the filename specified. In this case, `-agentpath:libexample.so` will use the agent named libexample.so from the current directory.

On the Linux system an agent is a standard shared object that must have the Agent_OnLoad() function exported. In this function, the JVMTI environment should be created. In the C++ language, this is done by the following code:

```
jvmtiEnv * new_jvmti = 0;
jint res = jvm->GetEnv((void **)&new_jvmti, JVMTI_VERSION_1_0 );
```

Since multiple agents can be active, every one has its own JVMTI environment. After environment initialization, the required capabilities must be set:

```
static jvmtiCapabilities capabilities;
memset( &capabilities, 0, sizeof(jvmtiCapabilities));
capabilities.can_signal_thread = 1;
capabilities.can_tag_objects = 1;
capabilities.can_suspend = 1;
error = new_jvmti->AddCapabilities(&capabilities);
```

And finally the event callbacks can be set up:

```
jvmtiEventCallbacks callbacks;
(void)memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMInit = &callbackVMInit; /* JVMTI_EVENT_VM_INIT */
callbacks.VMDeath = &callbackVMDeath; /* JVMTI_EVENT_VM_DEATH */
```

```
error = new_jvmti->SetEventCallbacks(&callbacks, (jint)sizeof(callbacks
    ));
check_jvmti_error( new_jvmti, error, "Cannot set jvmti callbacks");

error = new_jvmti->SetEventNotificationMode(
 JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, (jthread)NULL);
error = new_jvmti->SetEventNotificationMode(
 JVMTI_ENABLE, JVMTI_EVENT_VM_DEATH, (jthread)NULL);
```

The function returns the value JNI_OK at success. The callbacks can be set anywhere in the agent, the capabilities must be set in the Agent_OnLoad() function.

When the agent needs to use its own threads, they should be created as daemon threads by the RunAgentThread() function from JVMTI. It requires the instance of the java.lang.Thread (or descendant) class as argument. This must be created using JNI as follows:

```
jclass thread_class;
jmethodID cid;
jthread newthr;

thread_class = Thread::jni->FindClass( JAVA_THREAD_CLASS );
if (thread_class == NULL) { error() }

//Get the method ID for the  constructor
cid = Thread::jni->GetMethodID( thread_class, "<init>", "()V");
if (cid == NULL) { error() }

//Construct a java.lang.Thread object
newthr = Thread::jni->NewObject( thread_class, cid );

//Free local references
Thread::jni->DeleteLocalRef(thread_class);

jvmtiError error;
error = Thread::jvmti->RunAgentThread(
 newthr, thread_entry, this, JVMTI_THREAD_NORM_PRIORITY );
check_error( error );
```

This code snippet starts a new thread with thread_entry() function as the entry point. The standard threads (pthreads) cannot be used in agent because the JVM doesn't stops until all non-daemon threads are finished. It means that if the agent would use normal thread, the Java program would never exit, and the user would need to terminate it manually.

2.3 Data integration

In performance analysis of the applications a plain profile containing information about executing code positions may not be sufficient. This profile measures the utilisation of only one resource in computer system – processor (in general). But the performance of application is also affected by other resources and their availability. For example, if system lacks free memory, the application will probably run much slower. To observe phenomena like this data integration is usefull. Here we provide brief description of two technologies available.

2.3.1 Measurement infrastructure

In [2] an infrastructure for performance data measurement is proposed. We use the concepts of this infrastructure later in this work, so we will provide a brief description in the following section. The infrastructure was designed with Java language in mind This has some drawbacks when using it in C++. we offer some improvement suggestions in the next section.

2.3.1.1 Infrastructure architecture

The infrastructure is based on idea that the execution of a computer program is a sequence of many events. Events can be like method entry, memory access, write to a file or instruction execution. The infrastructure allows to define a set of events that are important for the user. For every event, a set of performance data that should be measured at all event occurrences can be specified. Here we will provide description of the infrastructure's architecture (its major components).

As shown in *Figure 2.3*, six components (or subsystems) are defined: Event Sources, Performance Data Access, Event Processing, Data Storage, Infrastructure Management and Data Delivery. The first four are meant to run in the context of tested application, the latter two can be separated. Now we will provide the description of these six components.

Event Sources: The objects that belong to the Event Sources are responsible for emitting performance events, that trigger the collection of the performance data. The component gets timing information from the Performance Data Access component and the Event Processing component is used to notify the infrastructure about the event.

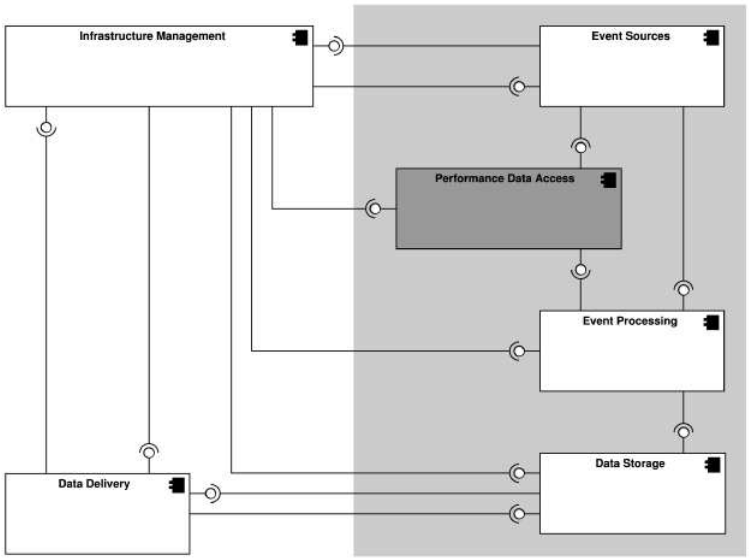


Figure 2.3: Infrastructure overview, from [2, p. 56]

Performance Data Access: The component provides unified access to different sources of performance data. It creates and manages measurement contexts, which are the selections of performance data associated with particular performance event. These contexts are used by the Event Processing component to collect data.

Event Processing: The component is responsible for processing performance events generated by the Event Sources. It includes collecting data and recording them (along with event data) to the memory using the Data Storage component.

Data Delivery: The subsystem is responsible for delivering the recorded data to the consumer. It may be writing files with traces, sending data over the network, etc. It reads data from the Data Storage. It may poll the storage for new data or read the data at user requests only.

Infrastructure Management: This part controls all other components, configures and coordinates them. One of other functions is the registration of event sources.

2.3.1.2 Improvement suggestions

Here, we will provide two suggestions that could improve the design of the infrastructure. First, the sensors that measure data, cannot be configured to accept parameters from events that triggered the data sampling. The example could be the sensor, that should measure the performance counter values from CPU on which the event occurred. In the design of infrastructure, this could be only solved by creating a separate measurement context for every CPU and add logic to the event processors that would select the correct context. We think, that the sensors should have option to read information from event data (or some other means) to configure the actual measurement.

Other suggestion involves the processing and recording of events. The infrastructure defines interfaces, that requires the definition of separate processor and recorders for all event types. Sometimes, this means duplicating code, if the event types require same or almost same handling. In this work, this case is represented by performance counter overflow events for Java and for native code. Their handling differs only in the need to inspect Java code position for Java events. In the original infrastructure interfaces, this would require two recorders and two processors for data measurement, even if the only difference in handler would be in the parameter that informs about event type passed to data storage. We think event data should be extended by the parameter representing the event type that triggered the event or pass event parameters to all methods in event handling procedure.

2.3.2 DTrace

A new subsystem for system-wide profiling support, called the Solaris Dynamic Tracing Framework (DTrace for short), was introduced in the Solaris 10 operating system [11]. It consists of dynamic instrumentation support and special script language (the "D" language) with its interpreter.

Many (tens of thousands) probes are in the operating system, including entry and exit points of almost every kernel function, function entry and return point in application, entry and return from the syscalls, etc. All of them are disabled when the DTrace is not running and the probes have no effect on the performance of the system or the applications. The DTrace subsystem can enable selected probes as necessary using the DTrace scripts - when a DTrace script is started, the system dynamically instruments the kernel or the applications at the places that correspond to the probes specified by the script.

The D language scripts are generally sets of callbacks associated with sets of probes. When the instruction flow passes a probe from the set, the specified action in the

script is executed. In the callback the script may gather data of interest. The script can aggregate data from many probes and provide complex information about the running system and applications.

For example, the following script measures the time spent on CPU by the threads of the specific user:

```
#!/usr/sbin/dtrace -s

sched:::on-cpu
/uid == 1001/
{
self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
@time[execname] = sum( timestamp - self->ts );
self->ts = 0;
}
```

After running the script, the result can look like this:

```
# ./sched.d
dtrace: script './sched.d' matched 6 probes
^C

  pt_chmod                                            517438
  gnome-pty-helper                                    855193
  utmp_update                                        1342610
  awk                                                1545419
  ls                                                 1707776
  basename                                           2202486
  sshd                                               2912672
  iiimd                                              4564425
  run-mozilla.sh                                     7782820
  ...                                                    ...
  nautilus                                        1095535707
  gnome-netstatus-                                1329197887
  java                                            3084636258
  gnome-terminal                                  5804337603
  firefox-bin                                    65677521752
  Xorg                                           77324524853
```

Chapter 3

Solution options

Our ideas and techniques that lead us to the solution of the problems proposed in *Chapter 1* are discussed here. First, we provide detailed analysis of requirements and goals, then we describe our ideas briefly as we were experimenting with them. The major part of this chapter provides detailed description of ideas and techniques.

3.1 Requirements and goals

Our goal is to develop a profiler based on infrastructure [2]. It must work with the JVM from Sun and C++ code as two examples of commonly used environments.

Pursuing the advantage of constant overhead associated with the sampling profilers, our profiler should be based on sampling and should not use instrumentation of code (except where the programmer decides to create sampling points manually). For the choice of events associated with sampling, our profiler should be able to use the common source of events that is the overflow of processor performance counters.

Since our profiler has to deal with Java, statistical sampling must be able to read the code position in Java from the interrupt handler that delivers the event. This brings problems with re-entrancy because the interrupt handler can be invoked at a time when the virtual machine is not ready for code position inspection.

Next requirement is that the profiler can measure performance data from hardware or operating system during event processing. In this case, we must be careful again, because some data may not be available from every caller context or at every moment.

The last goal is to propose universal data format capable of storing various events and performance data. The format must support fast seeking in the data and be

ready for small and large data associated with events.

In following sections, we provide our solution possibilities. We think all of them could lead to successfull result, but in many cases technical problems prevent them to be used.

3.2 Ideas overview

As mentioned before, two major problems are to be solved: notification about CPU performance counter overflow and inspection of code position in Java. One elegant solution that would solve both problems at once is a solution that would allow querying current position of the executing Java code from pretty much anywhere is storing the current position in a globally accessible variable within the executing Java process. This option is discussed in the *Section 3.3: Code position storing*. It turned out to have too big overhead.

After failure of the option that would solve both problems, we addressed them separately. First, we discuss the options that allow us to notify userspace about a performance counter overflow. In the *Section 3.4: PAPI library* we deal with options that provides the library. We found out that it does not provide facilities we need. The perfctr patch option is discussed in the *Section 3.5: perfctr Linux kernel extension*. It almost suits our requirements, but it has no documentation and the required modifications would be too difficult. In the Linux kernel one subsystem that deals with CPU performance counters is already included – OProfile. Natural experiment is a modification of its processing of NMI interrupt handler it uses. In the *Section 3.6: OProfile NMI modification* we address this option and show that it brings many obstacles, of which some we could not overcome. And finally, in the *Sextion 3.7: OProfile with fixed interrupt and signal* we discuss option that modifies OProfile more than slightly. This option proved to be successfull.

After we have achieved the notification of userspace, the next problem is the inspection of code position in Java. The *Section 3.8: Java code position* section address this problem. Since the notification from kernel comes in form of signal, it cannot be solved by direct call to the GetStackTrace() function from JVMTI.

The last and probably the least difficult problem is the format of data that will be stored to the disk. The *Section 3.9: Data storage* proposes the solution.

3.3 Code position storing

For the Java language, the runtime environment provides facilities for storing the position of the code by using callback at method entry and exit. This position then could be read at the moment of the performance event (even from the kernel). The overhead of the position storing agent is too high, however.

Method entry and exit events Two events are availabne in the JVMTI library: JVMTI_EVENT_METHOD_ENTRY and JVMTI_EVENT_METHOD_EXIT. They are available after setting can_generate_method_entry_events capability. They are called with every Java method entry and exit. The thread and method references are provided as parameters.

Agent implementation To test the feature, we created an agent that responded to the JVMTI_EVENT_METHOD_ENTRY and JVMTI_EVENT_METHOD_EXIT events and measured the overhead. The agent is simple, it only sets the capabilities and registers callbacks to the events in Agent_OnLoad() function, enables the events in JVMTI_EVENT_VM_INIT event and has empty method entry and exit callback. This agent is available in enclosed CD.

Overhead measurement We measured the overhead of the implemented agent using two benchmarks described in *Chapter 5* – Fhourstone and TestingJavaProgMulti. We used version of these programs that do not run as long as their normal versions, but we think these results are obvious:

	Benchmark	
	Fhourstone	TestingJavaProgMulti
Configuration	Kpos/s	time (s)
Without agent	1303.23	6.04
With agent	45.13	92.13
Overhead	2888%	1523%

This result were created as arithmetic mean from several runs of benchmarks. We think it is obvious that the overhead makes this option not usable.

The next possibility could be the one using dynamic bytecode manipulation, but we would like to avoid it, since the instrumentation is not good option if we want to use statistical sampling.

3.4 PAPI library

One of the options how to notify the userspace about CPU performance counter overflows is PAPI library's (brief description is in *Section 2.2.2*) feature, that allows programmer to define a callback for the overflows. However, this proved to be problematic for profiling native code.

3.4.1 Library facilities

The library provides following function:

```
int PAPI_overflow
 (int EventSet, int EventCode, int threshold, int flags,
    PAPI_overflow_handler_t handler);
```

It sets up the specific event (determined by EventCode parameter) from the EventSet (EventSets are PAPI encapsulation of configured performance counters and events) to generate overflow events, if the counter value reaches threshold. At that moment the handler callback is called. The library can exploit hardware counters overlow feature, or simulate it by the software. If available, PAPI prefers hardware overflows. The software events can be enforced by using flags parameter.

The hardware counter overflow is implemented with perfctr Linux kernel patch (see *Section 2.2.1*) on the Linux x86 systems, which is generating signals that PAPI intercepts and passes the control to the user's callback.

3.4.2 Counter Locality Problem

We tried to use the PAPI to generate the performance counter overflow events for the native code, first. This turned out to be problematic. Even if the perfctr patch provides interfaces for signals from every CPU performance event in system (global counters) the PAPI uses only virtual per-task counters, which need to be turned on explicitly by the every task that they are associated with. This is an insurmountable obstacle for sampling profilers of native code, which cannot easily invoke PAPI functions on behalf of other tasks. The problem is illustrated on a sample code in the *Listing 3.1*.

```
#include <papi.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
```

```
#define THRESHOLD  10000000
#define handle_error( x ) do { printf( "error\n" ); exit( x ); } while(
    0 );

int total = 0;       /* total overflows */

void handler(int EventSet, void *address, long_long overflow_vector,
    void *context){
        fprintf(stderr, "handler(%d) Overflow at %p! vector=0x%llx\n",
                EventSet, address, overflow_vector);
        total++;
}

void* thread_f( void *p ){
        long long x = 0;
        while ( 1 ) x++;
        return (void*) x;
}

int main()
{
        int retval, EventSet = PAPI_NULL;

        /* Initialize the PAPI library */
        retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT) handle_error(1);

        /* Create the EventSet */
        if (PAPI_create_eventset(&EventSet) != PAPI_OK) handle_error(1)
            ;
        if (PAPI_thread_init(pthread_self) != PAPI_OK)  handle_error(1)
            ;

        /* Add Total Instructions Executed to our EventSet */
        if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
            handle_error(1);

        /* Call handler every THRESHOLD instructions */
        retval = PAPI_overflow(EventSet, PAPI_TOT_INS, THRESHOLD, 0,
            handler);
        if (retval != PAPI_OK) handle_error(1);

        /* Start counting */
        if (PAPI_start(EventSet) != PAPI_OK) handle_error(1);

        /* Create new thread */
        pthread_attr_t detached_attr;
        pthread_attr_init( &detached_attr );
        pthread_attr_setdetachstate( &detached_attr,
```

```
        PTHREAD_CREATE_DETACHED );
    pthread_t tid;
    int t_id = pthread_create( &tid, &detached_attr, thread_f, 0 );
    if ( t_id != 0 ) { fprintf(stderr, "pthread_err\n"); }
    }

    sleep( 15 );

    return 0;
}
```

Listing 3.1: PAPI overflow testing

The testing program initializes PAPI, sets some events to count on hardware counters and sets handler for counter overflows. After that it starts new thread that does some computation and the main thread sleeps for some time. Even if the processor is fully occupied, no overflow notification arrive to the application during the sleep. This can be circumvented by calling PAPI_register_thread() function at the thread start.

3.5 perfctr Linux kernel extension

Another option to notify userspace about CPU counters overflow is to use perfctr Linux kernel extension (for overview see *Section 2.2.1*). The extension provides a way to configure the kernel to send the signal to the process at counter overflows. There are 2 options - global and virtual counters. While virtual counters count events per-task, global counters count for all system. Virtual counters have the same problems as PAPI library. The global counters would be usefull. However, absolutely no documentation is available for extension. This makes the usage of it very difficult. The extension is configured by set of ioctl functions at file /dev/perfctr. We did not manage to determine the proper commands and their parameters, unfortunately.

3.6 OProfile NMI modification

In the Linux kernel another component that works with CPU performance counters is already integrated - OProfile (brief description is in *Section 2.1.2*). We decided to try the possibility that modifies OProfile's counter overflow handling. Since the counter overflow event is interrupt, we will describe the hardware mechanism of interrupt generation and their handling, then we will describe how the Linux kernel handles the interrupts (both for x86 architecture). We will discuss the possibilities of notification of the userspace from the kernel. After that we will describe our

attempts to modify the OProfile NMI interrupt handling to our needs. We found out that NMI interrupts have too many obstacles that we did not manage to overcome.

3.6.1 Intel APIC

Here we will provide a short description of the interrupt management hardware, the *Advanced Programmable Interrupt Controller* (APIC). We will cover only topics with relation to this work, see [5, Chapter 5 and 8] for full detail.

If referring to APIC, two of them exist: local APIC and I/O APIC. While the first one is integrated with the processor and controls the interrupts to that processor (or core), the second is external (part of the chip set) and its task is to control interrupts from external devices and relay them to the local APIC. I/O APIC is not important in context of this work. The interrupts handled by local APIC divide to following categories (or interrupt sources):

- Locally connected I/O devices – interrupts asserted on LINT0 and LINT1 processor's pins.
- APIC timer generated interrupts – APIC has a timer that can generate interrupts at local processor when set count is reached.
- Performance monitoring counter interrupts – at performance counter overflows the interrupt can be sent to local processor.
- Thermal Sensor interrupts – can send interrupt when the internal thermal sensor has been tripped.
- APIC internal error interrupts – at error in local APIC programming, the APIC can be configured to send the interrupt to the processor.

These interrupt categories are referred to as *local interrupt sources*. Also following categories are available:

- Externally connected I/O devices – interrupts from external I/O APIC, sent as messages to the local APIC.
- Inter-processor interrupts – interrupts from other processor in multiprocessor system, or programmed self-interrupt

The local interrupt sources can be configured by the Local Vector Table (LVT), which is set of APIC registers. See *Figure 2.2* for its layout and meaning of the register fields. Another important register is the *Interrupt Command Register* (ICR).

By writing to this register the IPIs are sent to the specified processors. IPIs are distributed to other processors through processor system bus (XEON processors) or by special APIC bus (P6 family processors). The self-interrupt can be sent or directly to the local processor, or by message to the bus that returns with delay to the processor. If the interrupt message arrives and the APIC has not free slot for it, it sets the error status to retry and discards the interrupt. If free slot is available, APIC stores it there and waits while processor is ready for next interrupt (interrupts cannot be interrupted, not even by NMI). This description is valid for normal (fixed) interrupts, NMI and other special interrupts are handled specifically. They are delivered directly to the processor core for handling.

3.6.2 Linux kernel interrupt handling

In this section we will discuss the mechanism that uses the Linux kernel to handle the interrupts. This includes the interrupt entry points, processing of the interrupts (passing control to the handlers) and return to the userspace. We will not follow IRQ handling. Then we will name the functions to control interrupt handlers and interprocessor interrupts (IPIs).

The interrupt and CPU exception entry points are in the arch/x86/kernel/entry_32.S (entry_64.S) file for 32-bit (64-bit) version of x86 architecture. It contains entry points for processor exceptions, interrupts, syscall and IRQs. Every CPU exception and architectural interrupt has one entry point defined, Linux system interrupts has the entry points, too. Every handler has the same basic structure:

1. Save registers to stack.
2. Call handler in C language.
3. Restore registers from stack.
4. Return from interrupt/exception (IRET instruction).

Specific exceptions (page fault, debug exception, etc.) have more complex handling, but this basic structure is still same. For NMI this procedure is extended in the beginning – it checks the stack, because the NMI can happen in syscall handling. All Linux system interrupts and syscall have more complex structure, after the C handler can be executed additional handling. The interrupt procedure checks if the task that is going to be resumed has no signals pending. If so, the control is returned not to the original code position, but to the signal handler. It also checks if the reschedule flag is set and if so, it calls the scheduler that can do its work.

3.6.3 Userspace notification options

After interception of the CPU counter overflow by the interrupt handler, we need to notify userspace about the event from that handler. This has two basic reasons:

1. The position in Java code can be determined by userspace library call only.
2. The measurement of data should be done in userspace – interrupt handlers must be as fast (and safe) as possible.

We determined two basic mechanisms: sending signal to the userspace thread or reschedule to prepared thread. As mentioned in *Section 3.6.2*, if signal was created in NMI, it will not be delivered at return from the interrupt. To circumvent this, we thought about following option - send IPI to self, to trigger consequent interrupt, or modification of return from the NMI to simulate the actions done in return from common interrupt.

3.6.3.1 Signal with IPI

One of important requirements is that the userspace is notified about the event (CPU counter overflow) immediately, or at least after delay, that is independent on the running code. The signals are not handled in return from NMI interrupt. This means, that signal created in the NMI handler will be delivered at the first following switch from kernelspace to the process with signal pending. This can be at syscall, or if the application is currently running code without syscalls, at next reschedule of the task. This behaviour would be inappropriate. One idea that could solve this problem would be sending IPI to the same CPU. As described in *Section 3.6.1*, the IPI to the local CPU is delivered or immediately, or after returning the issued message from the bus. In the first case, the next interrupt is created immediately after return from NMI, in the second case the interrupt will occur at some time, that is affected by hardware implementation only and is not dependant on userspace code. This interrupt will be normal, fixed interrupt and in handler of that the signal handling is already active. We used the reschedule interrupt, that is available in Linux. It has empty handler, since everything expected of that is done in default interrupt return (reschedule if needed).

We implemented the kernel patch as described above. We measured the delay of the next interrupt, and learned that on Athlon XP 2600+ processor it is less then 10000 clock ticks. This would be nice result, however the problem appeared always after some time. The system running the mentioned mechanism wasn't stable, after few thousands of events it hanged the system regularly. It seamed like a deadlock in

some locking mechanism in the kernel. Later we determined that the function that sends the signal was causing this. We tried to re-implement all required subroutines to use locking, that cencels the event if some lock was not available. This was not successfull. After some time we had to reject this option. The probable problem is that NMI can happen in very wrong conditions, such a returning from signal handler syscall, mutex locking, etc. Since many actions in kernel are synchronised using some form of locking, it is impossible to change all of them to fail if the lock is not available.

3.6.3.2 Signal with non-NMI return simulation

Another option we were thinking about is an attempt to change the return path from NMI interrupt to incorporate actions done in normal interrupt return, signal handling primarily. This would mean to change the NMI normal return path, `restore_nocheck_notrace` code segment in the arch/x86/entry_32.S file that only restores registers and executes IRET instruction, to something that checks the signals.

Unfortunately, this could have unpredictable consequences, from same reasons as in previous section, and here it would be even more complicated, because the event can happen in spinlock at syscall handling, which means that the kernel is not in consistent state when in NMI handler.

From these reasons and because the deadlock problem mentioned in previous section was caused by functions notifying the kernel that it should create and deliver signal (send_sig_info() function). This means that even if we were successfull in delivering signal from NMI, the deadlock would occur eventually.

3.6.3.3 Reschedule to a servicing thread

The next option we were interested in is the possibility to have prepared thread for handling counter overflows, that is in sleeping state by default. The idea is to wake this thread and make the scheduler to plan this thread to run directly after the return from interrupt handler.

This has some drawbacks, the most important are the complexity of the scheduler and the fact that NMI can happen in time when scheduler is not in consistent state. A scheduler is probably the part of the Linux kernel that overcomes dramatic changes most often. It is also very complex and every change can have unforeseen consequences. This makes very difficult any modifications. The other, even worse problem, is the fact that NMI can happen virtually everywhere, including the fork()

syscall, for example. There is created new process, which means that the internal scheduler structures must change, and any external action can be lethal. From these reasons we had to give up this option.

3.7 OProfile with fixed interrupt and signal

After failure of all previous options, which were or without modifications of underlying technology (PAPI, perfctr extension, position storing) or without modification of sampling mechanism (OProfile's NMI) we had following possibilities:

- Modify PAPI to use global counters.
- Study perfctr extension to the detail and modify the signal data.
- Modify the OProfile to use normal (fixed) interrupt instead of NMI and send a signal to userspace process.

We decided to focus on OProfile modification, because it is very well documented (as opposed to perfctr extension), based on simple ideas and it has the advantage that we can still collect the samples as OProfile usually does along with our tool sampling.

This approach proved to be successful. Here we will describe what had to be done for userspace notification and show that the overhead of sending a signal is acceptable.

3.7.1 Linux and OProfile modifications

The required modifications can be separated into two parts:

- Create interrupt handler for the overflow interrupt.
- Modify OProfile to use the created interrupt handler and send signals from it to the userspace.

For the interrupt handler entry point we used the code from perfctr kernel extension. It defines new section (symbol) in the entry_32.S or entry_64.S files. This symbol is later in C declared as asmlinkage. A call to the symbol that can be defined in C and declared asmlinkage is also there. The body of a handler is in C language. The routine gets structure with stored registers as parameter. During initialization, the handler is registered by set_intr_gate () function call, that associates the handler with

specified interrupt vector. We do not need to care of interrupt tables and other processor structures ourselves, the kernel handles it automatically.

Then we moved OProfile's counter overflow handling to our newly created handler (arch/x89/oprofile/nmi_int.c). After this, we modified architecture-specific overflow handlers to send signals using send_sig_info () function if the process requested this. Since this is normal (fixed) interrupt, the signals are delivered at return to the interrupted thread. In signal data we provide program counter where the thread was interrupted, the number of CPU that generated the interrupt and what event trigerred the interrupt.

3.7.2 Overhead measurement

After we implemented the options mentioned here, we measured the overhead caused by signal sending and receiving. We created simple C program, that only counted number of signals in signal handler and we measured how the sending of signals will slow down the program. After running the program several times, we got the average slow down of less then 0.5%. The OProfile was configured to send signal after completion of every 500000 instructions. In case without signals, the OProfile was disabled, so the overhead includes the overhead of OProfile, too.

3.8 Java code position

After successfully entering signal handler in userspace, the requirements for native (C) code are almost accomplished - the program counter could be used to identify the source line of the application. We do not do this, but we will suggest a way that could lead to effective way of getting this information. This is not true for Java, we need to determine the position in the Java code. In following sections we will discuss JVM and JVMTI facilities for this, and options how it can be done from signal handler.

3.8.1 Code position inspection

The facilities to inspect code position provides JVMTI (see *Section 2.2.3* for description). It defines two functions: GetStackTrace() and GetAllStackTraces(). The first one gets the call stack of one specified thread (or current thread, if no specified), the second gets stacks from all Java threads running. In following listing the declaration of the GetStackTrace() function is displayed.

```
jvmtiError
GetStackTrace(jvmtiEnv* env,
              jthread thread,
              jint start_depth,
              jint max_frame_count,
              jvmtiFrameInfo* frame_buffer,
              jint* count_ptr)
```

Listing 3.2: GetStackTrace function

It gets stack trace of thread, to maximum depth of max_frame_count frames. The result is stored to the array specified by frame_buffer pointer and the number of stored frames is in memory specified by count_ptr.

The GetAllStackTraces() function gets the stack frames of all Java threads, to the maximum depth of max_frame_count frames, and stores it to special structure, that the function allocates itself. The user provides only stack_info_ptr pointer, and thread_count_ptr where the number of threads is stored.

```
jvmtiError
GetAllStackTraces(jvmtiEnv* env,
              jint max_frame_count,
              jvmtiStackInfo** stack_info_ptr,
              jint* thread_count_ptr)
```

Listing 3.3: GetAllStackTraces function

In the *Listing 3.4* an example of GetStackTrace() function usage is shown. See *Section [9]* for full detail.

```
jvmtiFrameInfo frames[5];
jint count;
jvmtiError err;

err = (*jvmti)->GetStackTrace(jvmti, aThread, 0, 5, &frames, &count);
if (err == JVMTI_ERROR_NONE && count >= 1) {
    char *methodName;
    err = (*jvmti)->GetMethodName(jvmti, frames[0].method, &methodName,
        NULL);
    if (err == JVMTI_ERROR_NONE) {
        printf("Executing method: %s", methodName);
    }
}
```

Listing 3.4: GetStackTrace example

The problem is that no function that would determine the current executing thread is available. This is problem if trying to make profile from external thread, that gets

regular samples, for example. It must use GetAllStackTraces() function and the profile will not be accurate, or profile only one thread. For this work is important to solve this problem.

3.8.1.1 Executing thread identification

For correct sampling, samples must be generated from the thread that generated the event. In our case, we must determine where the Java code was executing in the moment of the counter overflow in the currently running thread. JVMTI does not provide this information and we did not find any other mean that would. In some cases, using the GetStackTrace() function can help, because if no thread specified, it gets the position of code in the Java thread from which context the function was called. It means that, for example, in JVMTI callbacks, the function returns the values from thread that generated the event. The proposed tool cannot get the position in code from callbacks – the events are independent on the JVM.

We have found one simple solution, but it is subtle and relies on assumption that is not guaranteed by virtual machine vendors. This assumption is that the Java threads are mapped to POSIX threads 1:1. This is true for JVM from Sun on Linux systems. If this assumption is valid, we can create a data structure, that holds associations between Java threads and POSIX threads. We use associative array for this. Single associations are inserted to the data structure at the time of thread creation, JVMTI provides JVMTI_EVENT_THREAD_START event. In handler the association is created (using pthread_self() routine to get POSIX thread identification). When the thread ends, the event JVMTI_EVENT_THREAD_END is generated and the association deleted from the data structure.

We think, that JVMTI or JVM should provide information that identifies the currently running Java threads on native threads. If the thread mapping is 1:1 this involves no overhead, for other mappings only storing one value in scheduler is needed.

3.8.1.2 Overhead

The pair of GetStackTrace() and GetAllStackTraces() functions is the only mean how to get the information where the running code is executing, except rejected instrumentation and method entry and exit callbacks. Unfortunately, it has significant overhead. To measure approximate overhead, we created a simple Java profiler, that has one sampling thread, that regularly called the GetAllStackTraces() function. We measured the average time spent in that function using RDTSC instruction. On machine with AMD Athlon XP 2600+ processor we get the average of approximately

27 millions of ticks for Java application with one thread (plus 2 servicing threads). This is really high, but we can do nothing with that if we do not want to modify JVM.

3.8.2 Signal handler

Now, we have solved the problem how to notify userspace about the CPU counter overflow event. In the userspace we have to handle the signal sent from kernel and find a way to determine the code position in Java, that means to call GetStackTrace(). The straightforward idea is to call this function directly from the signal handler. This does not work, the function returns empty stack. We also tried to call the handler on alternate signal stack. In this case the function returned error. This made us to look into JDK sources available as opensource [13]. Here we found out that this function searches the stack for prepared variable stored at the stack, that holds information needed for stack trace. This is the reason why direct call to the GetStackTrace() function fails. The signal handling procedure stores data on the stack and it becomes inconsistent in the meaning that backtrace is interrupted.

3.8.3 Stack modification

One workaround for the stack problem described in the *Section 3.8.2* could be the following procedure:

1. Send a value of stack pointer (SP) register of interrupted code from kernel in signal data.
2. In signal handler:
   (a) Copy part of the stack, from received SP to the top of the stack to some place in memory along with SP.
   (b) Reset SP to the value before CPU counter overflow event.
   (c) Call the GetStackTrace() function, this should simulate the behaviour outside of a signal.
   (d) Store the results to some place in memory.
   (e) Copy the original top of stack to original location.
   (f) Restore original SP value.

We had to reject this mainly because the signal data are not big enough to hold all required information. The size of the structure is 128B, however the kernel or userspace libraries copies only the size of the biggest data defined for specific sources, and they are only one pointer and one integer long in union part of the structure. We need 2 pointers – stack pointer and program counter. They are too large on 64bit machines, where pointers are 8 bytes and integers are 4 bytes long. Another probable problem is that the stack may be in undefined state – if the code was just creating new stack frame, the stack would be inconsistent as well as in normal signal.

3.8.4 Thread priority and semaphores

The attempts to handle the problem in signal handler showed that this approach is too difficult. We had to try something else. An effective way could be to let the task to determine the position in Java code to another thread. The problem is how to give the control to that thread when needed, and how ensure that the code position will be determined as close as possible to the position where the event has happened.

We found out that nature of semaphores and the scheduler along with thread priorities makes it possible. We can have prepared thread with high priority that is determined to detect the code position. It is sleeping on some semaphore. When the event in the form of signal arrives, we can raise the semaphore. In this moment the control is passed to the position detecting thread. This control passing works even if the threads have same priority.

We implemented a simple implementation using described approach. The tool had a very high overhead (same as measurements described in *Chapter 5*). We clearly identified the GetStackTrace() function as the reason of that overhead. If we commented out its call, the overhead dropped to insignificant values.

3.9 Data storage

After the events are received, the code position determined, we can measure a set of data associated with that event. The natural requirement is a way to store that data. The requirements are that the storage cannot use much more space than necessary and makes seeking in the data possible. The data can be very large, and can consist of very frequent events with little data or not too many events but much data collected at the events.

We proposed a data structure which consists of 3 components:

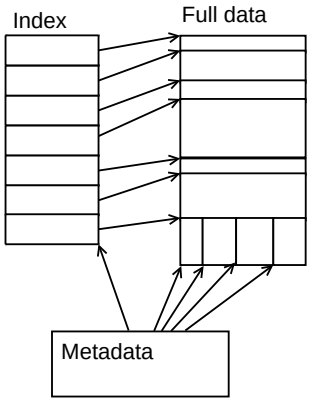


Figure 3.1: Storage structure

- Metadata – provides information about data collected for separate event types.
- Index – common event data, along with positions in full data are stored as set of equally-sized structures to make seek possible.
- Full data – the measured data, stored one by one in flat structures of variable length.

It is implemented using three files, the metadata file is in the XML format, index file is binary file of structures and full data file is a stream of records of variable length. In the index file positions in data file of the beginning of the data for one event are stored, along with length of that data. The the values from data file can be read by using data from metadata, that defines the order and type of data fields. The scheme of the storage is in the *Figure 3.1.*

Chapter 4

Solution documentation

We created a profiling tool in accordance with discussion in the *Chapter 3*. Here we provide a programmer's documentation of the solution. The description of the basic structures and principles in reasonable detail is provided. We provide a set of UML diagrams. Their intention is to show the basic relations between classes and the most important methods, not to be a complete reference of the classes. The detailed documentation of the classes can be found in doxygen-generated documentation provided as an attachment of this work.

4.1 General overview

The tool needs to have two parts – one for generating performance counter events and the other that receives events and measures performance data. The first part is implemented in Linux kernel. It takes form of a kernel patch. We modify OProfile's CPU performance counter overflow handling to suit our needs. It can generate signals at overflows and provides interface for reading counter values.

In the userspace, the signal handling is required, along with measurement of the performance data that can be associated with the events and storing the data to the file. Also an interface for use in native (C++) code is needed and JVMTI agent for Java profiling. The userspace part of implementation tries to use the interfaces and principles described in [2].

The simple overview of operation is following procedure:

1. Event generation – the condition that generates event occured somewhere in the system. It can be the performance counter interrupt or the application code runs into a trigger prepared by programmer.

2. Reception of event – the events are intercepted by entities defined in the *Event Sources* component. They provide unified interface for common configuration tasks. The goal of all entities in this component is to provide information about what happened and notify the measurement infrastructure about the event using its associated Event Delegate from the *Event Processing* component.

3. Performance data measurement – the Event Processing component measures performance data that it was configured to collect on the particular event using *Data Access* component.

4. Storing of data – the measured data are stored to the memory using the *Memory Storage* component to provide quick storage with relatively small overhead. Here ends the event processing.

5. Exporting data to external storage – when the memory storage is filled up to certain limit, the data is exported using the *External Storage* component.

Since the performance counter overflow events are received using signals, some limitations must be applied what can be executed in the handler. For this reason we divided the performance data into two groups – signal-safe and synchronous. The first can be sampled within signal handler, the other must be delayed while the tool is outside the signal handler. We use same mechanism as for determining Java code position here. The signal handler has one other consequence – we must be extremely cautious when using synchronisation mechanism (mainly spinlocks). If we lock the spinlock in some thread, and that thread is then interrupted by signal, we must ensure we do not wait for that lock, because it will be never released. From this reason, we use the locks in non-waiting mode from signal handlers, and if lock cannot be acquired, we must cancel the event. This does not happen too often, however (in average, 1 event of 20000 is cancelled in our tests).

Other problem that can happen – in the case of incorrect configuration mainly – is that next event arrives while the previous one is still being processed. In that case, we do not measure performance data and Java code position again, but we store the event with copy of data that will be measured by first event. For this reason every performance counter overflow event tries to acquire a lock at start of processing, that is released when the event is recorded or canceled of some reason.

In the following text, we provide detailed documentation of mentioned components. In addition, we have created a management component, that helps using the tool and Java agent interface that defines entry point and communication with JVM using JVMTI.

4.2 Kernel patch

The kernel patch provides configuration and delivery of the signals at the performance counter overflows and interface for reading the performance counter values. For configuration we use files in the /dev/oprofile filesystem. The signals are sent from OProfile's modified interrupt handler and the counter values are read using the ioctl syscalls. For sending the signals we use solution described in the *Section 3.7*. The patch was created for version 2.6.24 of the Linux kernel.

4.2.1 Sending signals

To be able to send the signals from the overflow handlers, two tasks are required to fullfill:

1. Modification of interrupt configuration – OProfile uses NMI, we must change this to use a fixed interrupt.
2. Modification of counter-checking routine – in OProfile, NMI handler calls a function that checks counters for overflow. We need to modify this function to send signals if it is configured.

In order to use a fixed interrupt, we needed to define a new entry point. This code is present in the arch/x86/kernel/entry_32.S (entry_64.S for 64-bit version) file. We used the entry point from perfctr patch (described in the *Section 2.2.1*). These entry points are displayed in the listings 4.1 and 4.2. The 32-bit version is more complex, but it does the same – stores the registers, calls the C handler (the smp_oprofctr_interrupt function) and executes actions like signal handling and reschedule if needed before IRET. As it is clear from example, we use the interrupt vector with number *0xf9*.

```
#define LOCAL_OPROFCTR_VECTOR 0xf9
ENTRY(oprofctr_interrupt)
    RING0_INT_FRAME
    pushl $~(LOCAL_OPROFCTR_VECTOR)
    CFI_ADJUST_CFA_OFFSET 4
    SAVE_ALL
    TRACE_IRQS_OFF
    pushl %esp
    CFI_ADJUST_CFA_OFFSET 4
    call smp_oprofctr_interrupt
    addl $4, %esp
    CFI_ADJUST_CFA_OFFSET -4
    jmp ret_from_intr
```

```
    CFI_ENDPROC
ENDPROC(oprofctr_interrupt)
```

Listing 4.1: 32-bit interrupt entry point

```
#define LOCAL_OPROFCTR_VECTOR 0xf9
ENTRY(oprofctr_interrupt)
    apicinterrupt LOCAL_OPROFCTR_VECTOR, smp_oprofctr_interrupt
END(oprofctr_interrupt)
```

Listing 4.2: 32-bit interrupt entry point

To use this interrupt, we must declare the entry point in C language and define a handler function. Entry point is declared asmlinkage **void** oprofctr_interrupt(**void**); and the handler's declaration is asmlinkage **void** smp_oprofctr_interrupt(**struct** pt_regs * regs). The only parameter respresents the structure with register values stored in the entry point. We created these handlers in arch/x86/oprofile/nmi_int.c. This is main OProfile's file for handling x86 performance counter overflow events, so we will do most of our changes in this file.

Then we needed to change OProfile to use our newly declared interrupt instead of NMI. This can be done in the nmi_setup() function by replacing register_die_notifier () function call by call to set_intr_gate () with paremeter set to the requested interrupt vector number (0xf9) and declared interrupt entry point (oprofctr_interrupt). This associates the the interrupt gate with our handler. Next is needed to configure APIC to use our interrupt instead of NMI. This is done in the nmi_cpu_setup() function by replacing the original call to the apic_write() function by apic_write(APIC_LVTPC, APIC_DM_FIXED | PERFCOUNTER_VECTOR);. Now the interrupt is ready. The handler only calls the machine-specific function that checks the performance counters.

For the signal delivery, we created a new model-specific function declared in the **struct** op_x86_model_spec defined in the arch/x86/oprofile/op_x86_model.h file. We added the check_ctrs_signal function that is called from handler, if signal sending is turned on. The implementations in the arch/x86/oprofile/op_model_ppro.c and arch/x86/oprofile/op_model_athlon.c files are identical as implementation of the check_ctrs model-specific operation, but if overflow is positively identified we check, if the interrupted process (from its PID stored in the current->tgid field of task structure) wants to be notified by signal (we will describe this checking later). If the check is positive, we prepare structure with signal parameters (we fill in program counter, processor number and CPU event) and send the signal by call to the send_sig_info () function.

4.2.2 Signal configuration

It is not possible to send signal to every process that is interrupted. This would cause the processes to terminate in most occasions. So we need to ensure only the processes that are prepared for these signals will receive them. We cannot use any complex data structures, because we must be able to use them in the interrupt handlers without locking. One of the options would be to add a special field to task structure that would tell if the task wants the signals. But it is unclear what to do if the task is duplicated (fork, clone syscalls). So we decided to create only a static array of fixed length where will be the process IDs (as seen from userspace – not thread IDs) of the processes that wants to receive the signals stored. PIDs are stored from first array field one by one, the list is terminated by 0. For changing the array, we use the locks, but for reading in the interrupt it is not needed since access for reading these values is atomic. Then in the interrupt handler, we traverse the arrray until we find the PID of the interrupted task or 0.

For setting up the signal delivery we created the /dev/oprofile/signal file using the tools provided by OProfile. It has defined read and write operations. The read operation returns string that contains all PIDs configured to receive signals. The write operation can modify the array - the userspace has to write one ASCII number that is interpreted as follows:

- "0" – disable the delivery of signals to the process that wrote the value.
- "1" (or any other positive number) – enable the delivery of signals to the process that wrote the value.
- "-1" (or any other negative number) – if executed by process with root priviledges, cancels delivery of signals to all processes, error otherwise.

This behaviour is defined in the read_signal_tgid () and write_signal_tgid () functions.

4.2.3 Counter value reading

For effective reading of performance counter values we provide an ioctl operation on the /dev/oprofile/read_value file. It uses OProfile mechanism for creating files, again. The ioctl handler is in the counter_value_ioctl function. The supported command is defined with the **#define** READ_COUNTER_CMD 129 macro. The parameter is the following structure:

```
struct perfctr_counter_value_args{
    int cpu;          //in param - cpu number or -1 for current
```

```
    int counter;          //in param - counter number
    unsigned long * value;  //out param
};
```

The function stores the value from defined counter (currently on executing CPU only) to memory specified by the value pointer. The reading is done by another extension of **struct** op_x86_model_spec. We added the read_ctr operation. In the model-specific implementations it uses CTR_READ macros defined by OProfile. It return the value as the number of CPU events that occurred since last overflow (that means *threshold_value+current_value*, since counters are counting from negative numbers towards 0). we have implemented another option for reading counter values, that use text files in /dev/oprofile. This serves for debugging purposes only, because it has significant overhead.

4.3 Event sources

The Event Sources part of the implementation is responsible for generation and reception of the events. The generation is used in manually triggered event types (Atomic events), reception in automatic events (performance counters). All event sources are descendant classes of EventSource abstract class (interface). To create event source objects for performance counter classes, creators are available. Global overview of the event sources component is in the *Figure 4.1*. Timer and delegates are members of different components and are described later.

Every event source may consist of many events. They are defined by their name and type and have fixed indices within one event source. The EventSource class with its implementations in our work is displayed in the *Figure 4.2*. The interface defines the getEventNames() and getEventTypes() methods that provides information about supported events. They return vectors with types or names, they have fixed order and the index of every event in these vectors can be later used to set event delegate or enable/disable the single event.

Next method is defined to set the timer counter (setTimerCounter()) that is used at every event to point out the time of the event. The setEventDelegate() method sets the event delegate, that is used to notify the infrastructure about the event and process the event (see *Section 4.5* for details). Methods enable() and disable() can enable or disable event reception for all events from the event source, while methods enableEvent() and disableEvent() do the respective action only for a single specified event.

The task of the event sources is to receive an event, create event data object (de-

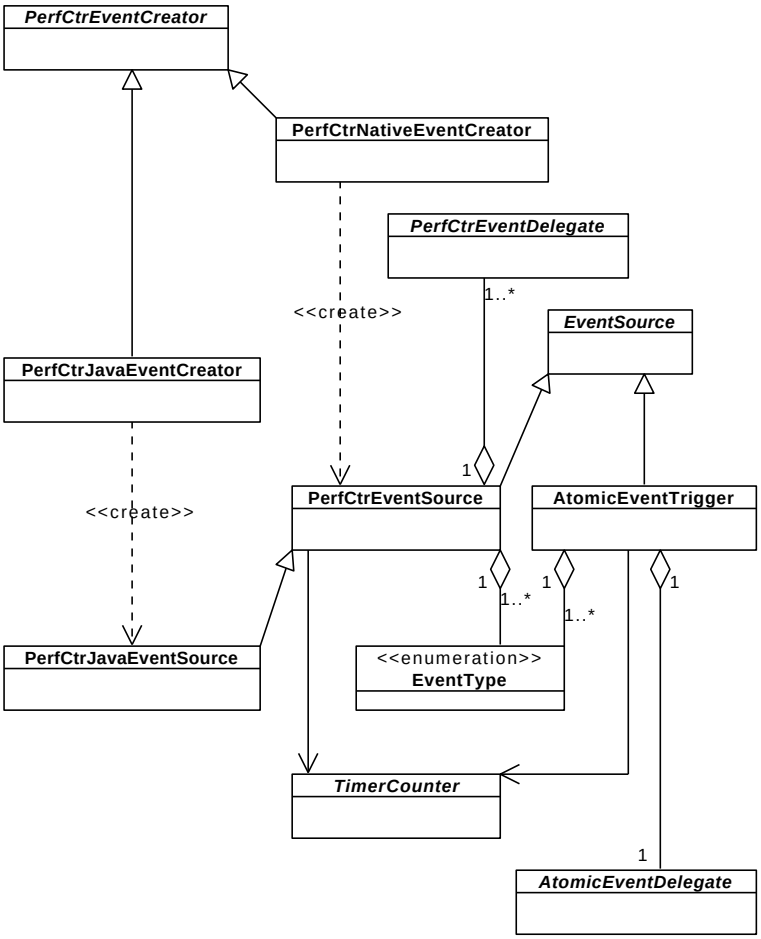


Figure 4.1: Event Sources

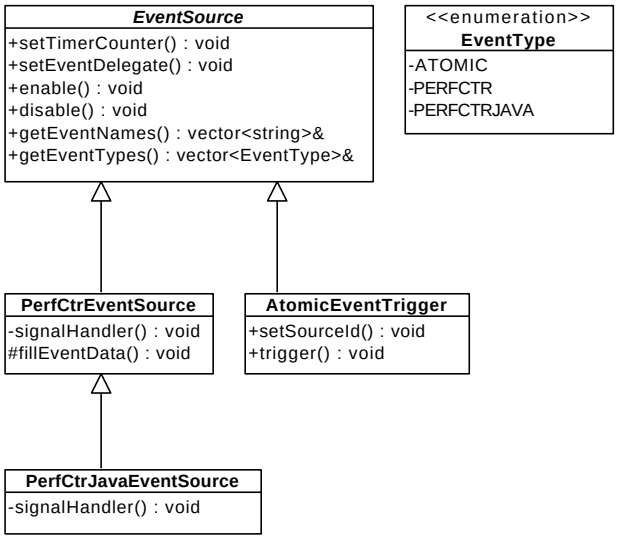


Figure 4.2: Event Sources class diagram

scribed in *Section 4.5*), fill it in and notify the infrastructure about event by using associated event delegate. This task does the trigger () method for atomic event and signalHandler() method for performance counters events. The atomic events are triggered manually from code. As it is clear from the name, for performance counters the events are processed from signal handlers. All events use single signal handler and the actual event is determined from signal data.

The event data that event source fills consists of time stamp, that is read from associated timer counter, event ID, which is a unique number, assigned in sequence from 1 and other fields from signal handler or source ID for atomic events.

Other classes in this component are responsible for creation of event sources for CPU performance counter overflows. They are implementations of abstract class PerfCtrEventCreator. This class defines the getEventSource() method that returns a reference to the event source created and held in the descendants classes.

We use two implementations – PerfCtrNativeEventCreator for native code profiling and PerfCtrJavaEventCreator for Java profiling. They are different only in the class type they create, so we decided to use a template mechanism to create these classes. This template is called PerfCtrEventCreatorTemplate and implements the event source generation. This is done in constructor that has list of requested CPU performance events with their overflow values specified. List of the PerfCtrEventSpec structures is used for that. Since we use OProfile for CPU performance events management, the communication with it is here. The procedure of event creation can be described in

following points:

1. Initialisation of OProfile (`#opcontrol --init` command in a suid wrapper).
2. Determine the CPU type by reading from the `/dev/oprofile/cpu_type` file.
3. Read events from the `/usr/share/oprofile/{CPU_TYPE}/events` file and parse it.
4. Check if requested events are available in OProfile.
5. Create sensors for requested events to be able to read values of counters.
6. Create event source with specified events.

4.4 Data access

The Data Access subsystem is responsible for measurement of the data associated with the events. It consists of sensors, which are the entities that measure data, timers, descriptors that provides information about sensors and timers, measurement context that is a set of sensors and management part. In the *Figure 4.3* the diagram of the whole component is displayed. We realize it is too complex – we placed it for demonstration only here. We will describe the component part by part.

4.4.1 Sensors and timers

Sensors are means that provide access to performance data. In this work we use very simple implementation without many features described in [2]. The sensors are required to provide generic interface to measure required data simply. To be able to do that, generic structures for values are needed. The ValueHandle interface provides this. In the *Figure 4.4* diagram with the interface and its descendants is displayed.

The **enum** ValueType defines data types that the sensors can return. The ValueHandle interface defines an inspection method for every possible type, method that gets the actual type stored and method that returns the memory size needed to store the data type represented by the object. The descendant class EmptyValueHandle is an error-checking class. It implements all methods from the ValueHandle interface, but every method generates error. Then the leaf classes for actual data types re-implements the methods that return their data type, memory size and value type. This along with the EmptyValueHandle parent class ensures that only correct methods can be called for the object implementing concrete data type.

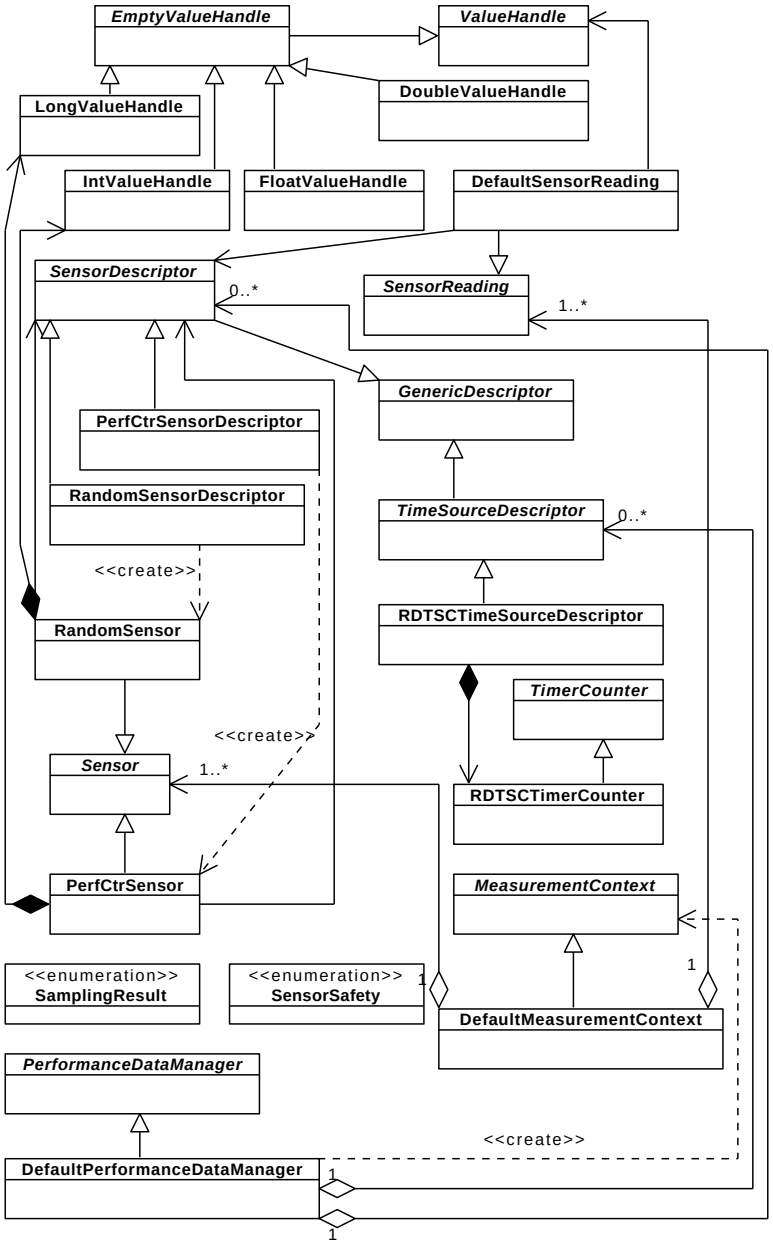


Figure 4.3: Data Access component diagram

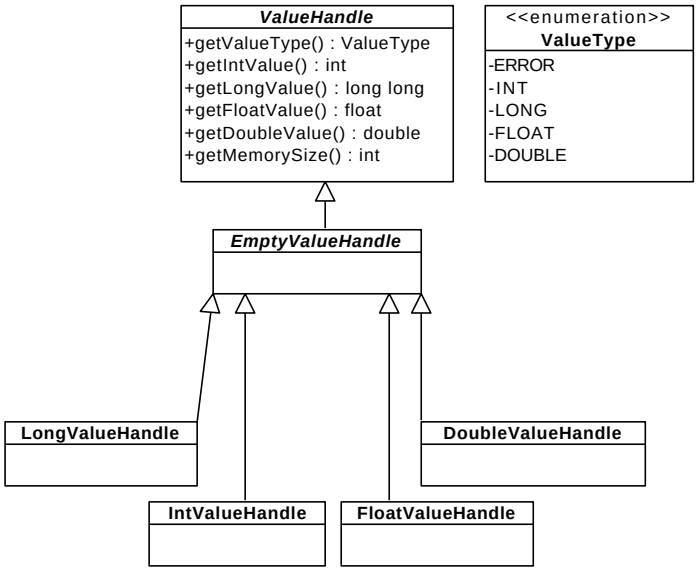


Figure 4.4: ValueHandle class hierarchy

Next part includes descriptors (diagram is in the *Figure 4.5*). They are classes that provide information about sensors and timers to user and creates sensors or provides references to timers. The GenericDescriptor interface provides information about sensor in strings - ID, name and longer description. Its descendants – SensorDescriptor and TimeSourceDescriptor provides information specific for sensors and timers. The first has getSensorSafety() method that tells if the sensor can be sampled within signal handler and getValueType() method that returns what type of values will the sensor provide. Next it declares the createSensor() method, that is responsible for creating new objects of the sensor class. All descendants of SensorDescriptor should return new instance of the sensor class that the descriptor describes. The caller is responsible for destruction of returned objects. TimeSourceDescriptor declares getTimercounter() method that returns a pointer to existing timer object. The implementations holds the counter object as private member and returns a pointer to it. Since all descriptors (except one for CPU performance counter reading) are almost identical – they differ only in the returned strings and type of created sensor, we use a preprocessor macro CREATE_SENSOR_DESCRIPTOR to create the classes. The PerfCtrSensorDescriptor class holds the CPU event number in addition to other sensors, that is used to create sensor.

Timers are implementations of the TimerCounter interface. This interface defines the getTime() method that return the time (methods getPrecision () and getResolution () are also defined as in [2], but we do not use these methods). We implement

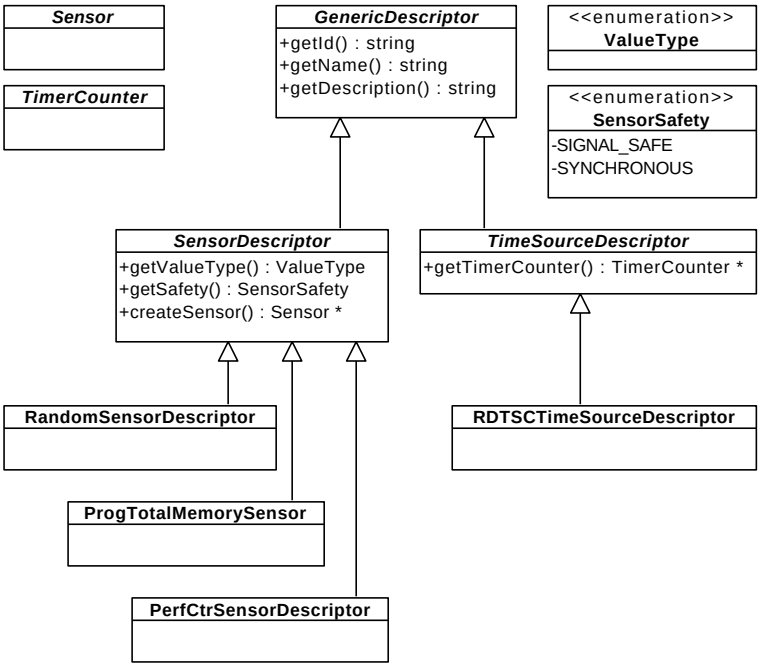


Figure 4.5: Descriptors class hierarchy

one time source in the RDTSCTimerCounter class, that reads the processor's timestamp counter using RDTSC instruction. It has corresponding descriptor – the RDTSCTimeSourceDescriptor class.

Sensors (diagram in the *Figure 4.6*) are responsible for data sampling. The Sensor interface defines methods that handles the data sampling and data storing, along with informational methods. The getSafety() method informs if the sensor can be sampled in signal handler. Next three methods handles sampling:

- prepare() – action that can be done prior to sampling (allocations, opening files, etc.)
- sample() – reads the data. Returns status: SUCCESS, FAILURE and ANOMALY if the data were sampled, but some unforeseen condition happened (like unexpected allocation) that makes the sample less valuable.
- decode() – parses the sampled data to an internal structure.

For external entity to be able to read the sampled data, it must call sensor's allocateSensorReading() method that returns new instance of object in that it can store values (the SensorReading interface, described later). Then the storeData method

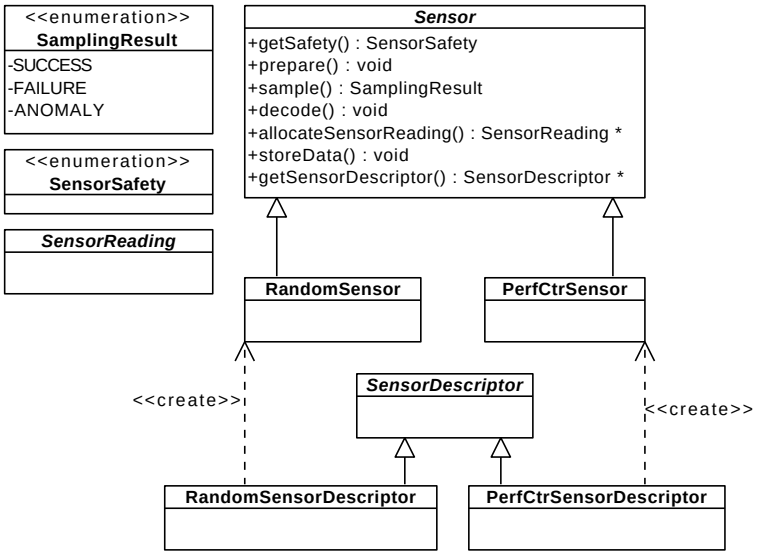


Figure 4.6: Sensors

stores the data to the object provided in parameter, that must be obtained from allocateSensorReading() call. Sensors can also return references to descriptor that created them.

As mentioned, data passing from sensors to other entities is implemented by the SensorReading interface. It has the getValueType() method that returns the type of the stored data, getSensorValue() that returns the value encapsulated in the ValueHandle object and the getSensorDescriptor() method that provides the descriptor of the sensor that created the object. All our sensors use inherited class DefaultSensorReading, that has one additional method – setValue() used by sensors to set value.

4.4.2 Measurement context

Measurement context is a container of sensors that handles sampling of data that sensors provide. As shown in diagram in the *Figure 4.7*, the context is defined as the MeasurementContext interface. Every context has assigned a unique identification number, the getId() method gets the number. The getSensors() method returns all sensors that are included int the context and it has the getSensorReadings() method. This method should acquire readings from all included sensors and pass them back to caller. Methods prepare(), sample() and decode() calls the identically named methods of all included sensors. The update() method is a convenience method that calls all three previous methods in sequence. Other methods have identical name, but have

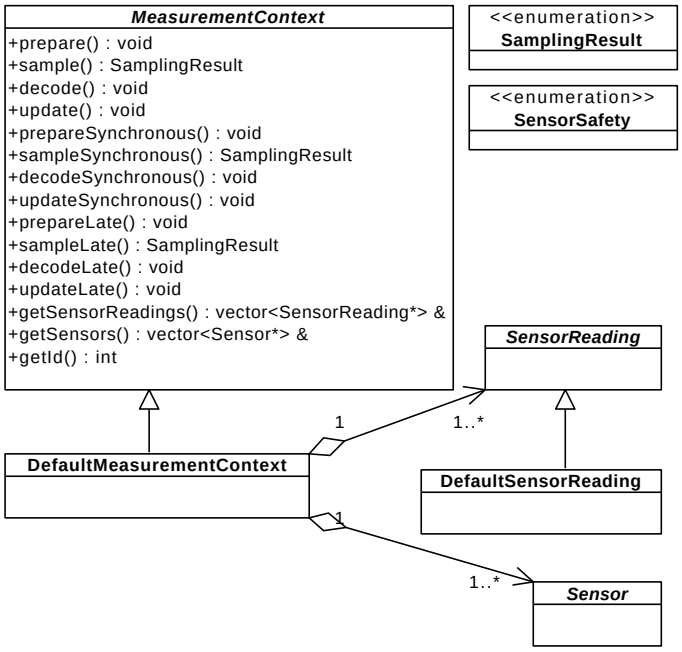


Figure 4.7: Measurement context

Synchronous or Late suffix. The first are for sensors, that can work in signal handler (they are sampled synchronously), the second are for sensors that must be sampled in safe place (they are sampled later after event). We implemented this interface in the DefaultMeasurementcontext class, which has internally two lists of sensors - one for synchronous sampling and another for late sampling. The sensors that will be included in the context are specified in the constructor.

4.4.3 Performance data manager

Task of the performance data manager is to create the measurement contexts, timer and sensor registration, providing descriptors of registered sensors and timers and provide timers for usage. The PerformanceDataManager interface (see diagram in the *Figure 4.8*) provides registering methods (registerSensor() and registerTimer()), unregistering methods (same, with un- prefix), methods that provide descriptors of registered sensors and timers – one pair for all of them returned in vector (getSensorDescriptors() and getTimeSourceDescriptors()) and one pair to get one concrete descriptor by its name (getSensorDescriptor() and getTimeSourceDescriptor()). It also provides references to timer counters, and provides two methods. One for default timer (getDefaultTimerCounter()) and one for specific counter specified by name

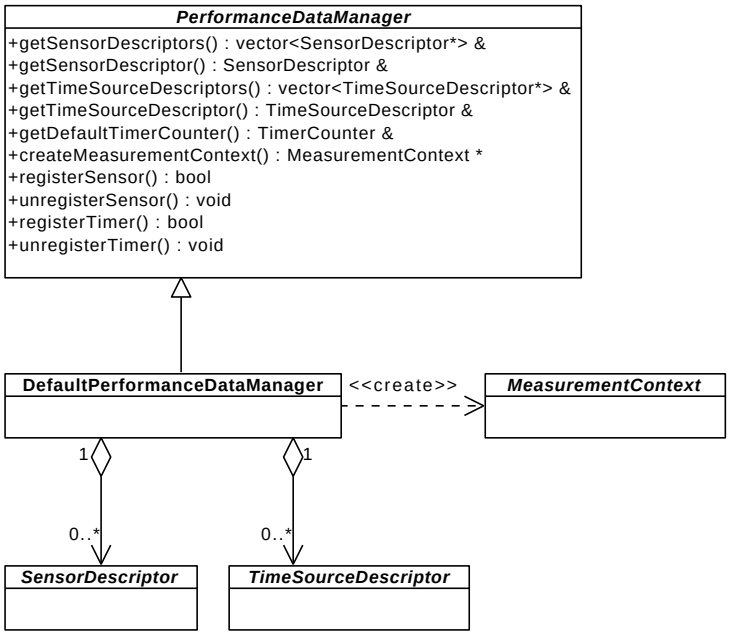


Figure 4.8: Performance data manager

(getTimerCounter()). The main task of performance data manager is creation of measurement context. Two methods with same name – createMeasurementcontex() – are available for this task. They differ in parameters, one takes a vector of sensor descriptor and the other a vector of sensor names.

Our implementation is in the DefaultPerformanceDataManager class. Its implementation is straightforward, with the sensor and timer descriptors stored internally as vectors. The context IDs are assigned from private member without any synchronisation mechanism – this means that contexts should be created at initialisation time and later do not change.

4.5 Event processing

The Event Processing component is the part responsible for receiving events from event sources, measurement of performance data associated with that events and recording the measured data and event data to the memory using the Memory Storage component (*Section 4.6*). Class diagram of the component is available in the *Figure 4.9*.

The basic idea is that an event notifies the infrastructure, it measures the perfor-

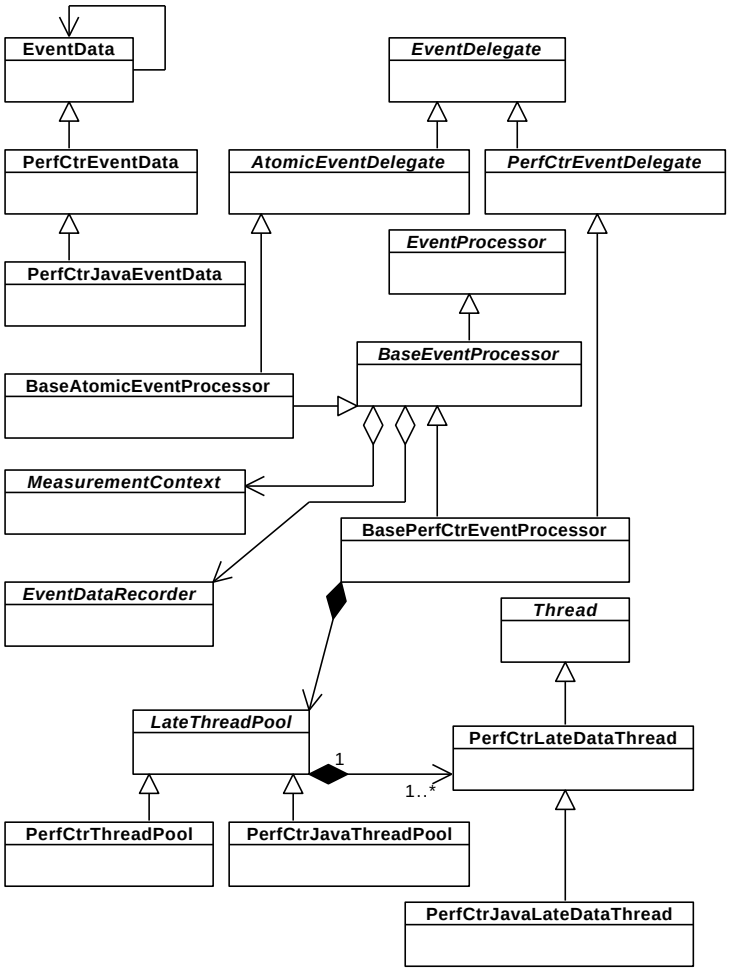


Figure 4.9: Event processing classes

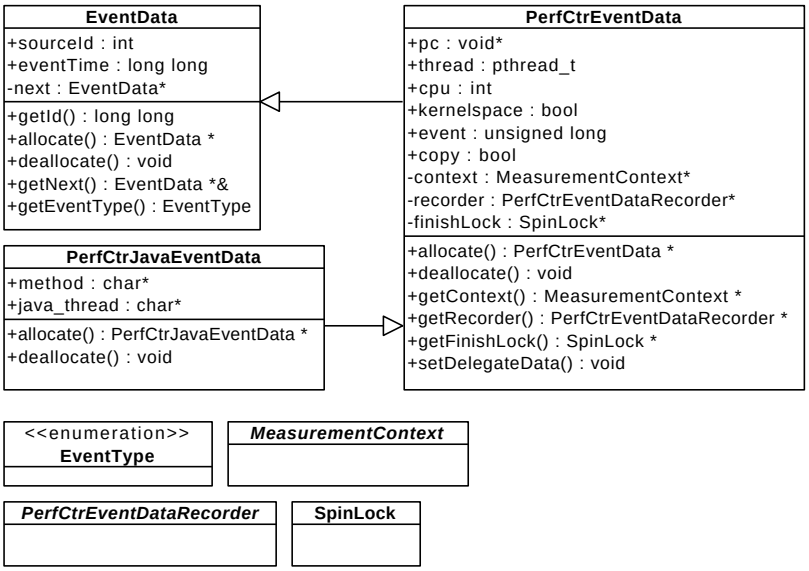


Figure 4.10: Event data classes

mance data using the prepared measurement context and stores the event data and the measured data to memory using some recorder. For atomic events, the procedure is this straightforward, but for events generated in signals – CPU performance counter overflows – not all data can be measured in the moment of event reception from the signal handler. We use same idea as for inspection of Java code position (*Section 3.8.4*). The description of component follows: we will describe event data, sampling threads and mechanism that records data.

4.5.1 Event data

Event data is a set of simple structures that are filled by event source with information about it. These information are time stamp and event source identification for all events. This is extended for the performance counters event by the program counter (at which was the performance counter overflow triggered), interrupted thread identification, CPU number on that the event occurred, information if the interrupted code was currently executing in kernelspace and identification of actual performance event. For events in Java language, two additional fields are available – Java method name and Java thread name. Few additional fields are defined, but they are not related to events directly and will be explained later. Class diagram is available in the *Figure 4.10*.

The problem with the event data is that the objects must be created and filled directly in the event. This implies need for allocation. Standard allocation from heap is not very good option, however, because it has some overhead and can change measured data. For performance counters overflow events it is also dangerous, since allocation from signal handler can damage the allocation mechanism. Because of these reasons we implemented an allocator of the event data from a pre-allocated memory pool. Every class has its own pool, that is sized as multiple of structure memory size. The multiple is defined by the EVENT_DATA_PREALLOC_SIZE constant. In this pool the linked list is created, linking memory areas as large as the classes. Then all classes have redefined **operator new** and **operator delete**. They are private operations, to disallow direct call of them because these operators cannot be virtual and we need to call allways the appropriate one. To be able to allocate objects, we defined static method allocate() that calls the **operator new**. This ensures that call to correct class' operator is made. For deallocation, we define virtual method deallocate() that calls correct **operator delete**.

We often needed to put event data into linked list. To make this simple, we added support for this directly to the EventData class. It has private member next, that can be accessed from the getNext() method.

4.5.2 Performance data measurement and recording

For performance data measurement the event delegates and processors are responsible. Event delegates are classes that are available for events and events call their notification methods at event occurrence. Event processors are classes that do the data measurement and recording. As displayed in the *Figure 4.11*, we use one class for both delegates and processors using multiple inheritance.

The EventDelegate interface is empty, and its only purpose is to provide common ancestor for delegates. We have created two derived interfaces, AtomicEventDelegate and PerfCtrEventDelegate that provide methods that must be called by event source at event occurrence – notify() for atomic events and notifyPerfCtrOverflow() for performance counter events. These methods implementation should initialise performance data measurement and recording using event processors.

Event processors are implementation of the EventProcessor interface. It defines methods that identify the type of event that the class can process (getEventType()). Next it defines two pairs of methods, with set- and discard- prefixes. The first pair (setMeasurementContext() and discardMeasurementContext()) sets and cancels the measurement context, that represents the data that will be sampled at events. This is the way of associating events and performance data. The second pair involves data recorders (described in the *Section 4.6*), methods ends with –EventaDataRecorder().

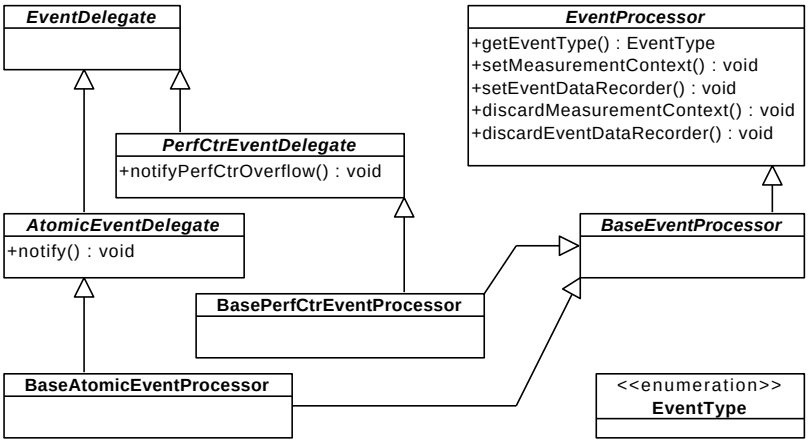


Figure 4.11: Event delegates and processors

It sets or cancels the data recorder – the object that is responsible for storing measured data and event data to the memory. These four methods are implemented in abstract class – BaseEventProcessor. It holds pointers to set context or delegate and destroys them in its destructor.

The BaseAtomicEventProcessor is defined as descendant of the AtomicEventDelegate and BaseEventProcessor classes. It implements the getEventType() method and notify() method. Since we use one class for delegate and processor, this method measures and records data directly. For atomic events that cannot be triggered in signals this is simple – it just calls the prepare(), sample() and decode() methods from context and the record() method from data recorder.

For the performance counter overflow events we defined the BasePerfCtrEventProcessor class. It is descendant of the PerfCtrEventDelegate and BaseEventProcessor classes. It implements the getEventType() method and the notifyPerfCtrOverflow() method. Since these events originate in signal handler, the handling is more complex than for atomic events. As mentioned before, we use the principle described in the *Section3.8.4*.

The implementation consists of several threads, one for every processor in the system (class diagram for threads is in the *Figure 4.12*). To provide easy access to these threads, we defined abstract class – LateThreadPool – it has the getThread() method that returns thread for specified processor number. We use two implementations, one for native application profiling – the PerfCtrThreadPool class and one for Java profiling – the PerfCtrJavaThreadPool class. These classes differ only in type of the threads they encapsulate, so we used a template class (PerfCtrThreadPoolTemplate) to create them. The native code uses the PerfCtrLateDataThread class as thread implementation and for Java code we have the PerfCtrJavaLateDataThread class. The thread class for Java

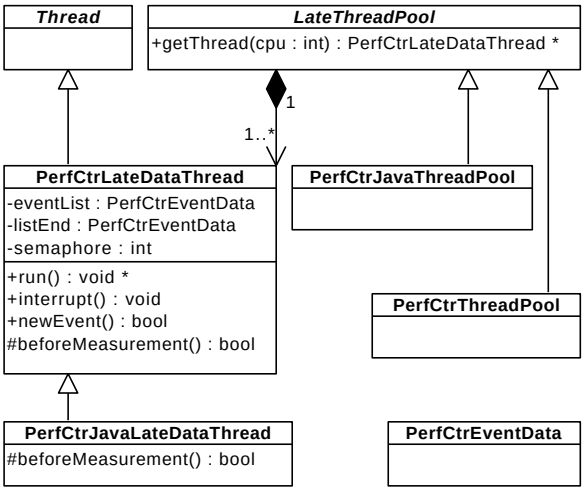


Figure 4.12: Measurement threads

is descendant of native class.

The thread class defines method newEvent() that notifies the thread that the next event is available for processing. It adds the event represented by its event data (the PerfCtrEventData class) to linked list (defined by the eventList field) and executes the *up* operation on the semaphore.

The run() method in thread classes is the thread routine. For our measurement threads it does the following procedure in the infinite loop:

1. Execute the *down* operation on semaphore – this makes the thread to sleep until the newEvent() method is called. As explained in the*Section 3.8.4*, this ensures immediate task switch to the measurement thread.

2. Retrieve event data from linked list. The event data contains pointers to measurement context and event recorder.

3. Execute pre-measurement action defined in the beforeMeasurement() method.

4. Measure the data from synchronous sensors, using prepareLate(), sampleLate() and decodeLate() methods of measurement context.

5. Record the measured data with the recordLateData() method of the recorder.

6. Make the event available again by unlocking finish lock.

The mentioned beforeMeasurement() method is empty for native threads, but for Java threads it gets the Java code position using the GetStackTrace() function. The threads' interrupt () method ends the infinite loop by destroying the semaphore what the run() method evaluates as request for exit.

Now we have explained the work of measurement threads so we can describe the operation of event processor for performance counter events. As for atomic events, delegate and processor are together in one class (BasePerfCtrEventProcessor). It is used both for native and Java profiling, since the difference is only one – the Java event must inspect the position of code. This difference is implemented by using different thread classes. They are passed as parameters to the constructor of the BasePerfCtrEventProcessor class, encapsulated in appropriate thread pool object.

The event source calls the notifyPerfCtrOverflow() method from signal handler. Its implementation is following procedure:

1. Prepare data recorder.
2. Try to lock the inProgress spinlock, what ensures that only one event of single type can be measured at once. If the spinlock is already locked, use the recorder and store the event as a copy only using the recordCopy() method of the recorder and return from method.
3. Store the context, recorder and inProgress spinlock pointers to event data (used in measurement threads).
4. Measure data from signal-safe sensors, using the –Synchronous methods from measurement context.
5. Record these data with recorder using its recordSynchronous() method.
6. Notify the measurement thread for the CPU on that the event was generated (stored in event data) about new event by its newEvent() method.

4.6 Memory storage

When the event data are sampled they must be stored in some common storage in memory to be ready for delivery to disk. Data could be stored to the disk directly, but it would cost overhead that can change measured data. Therefore it is better to store data to the memory, with almost no overhead and later save them to a file. Data storage component serves this purpose. Its class diagram is displayed in the *Figure 4.13*. The component consists of the –Recorder classes, that are means for

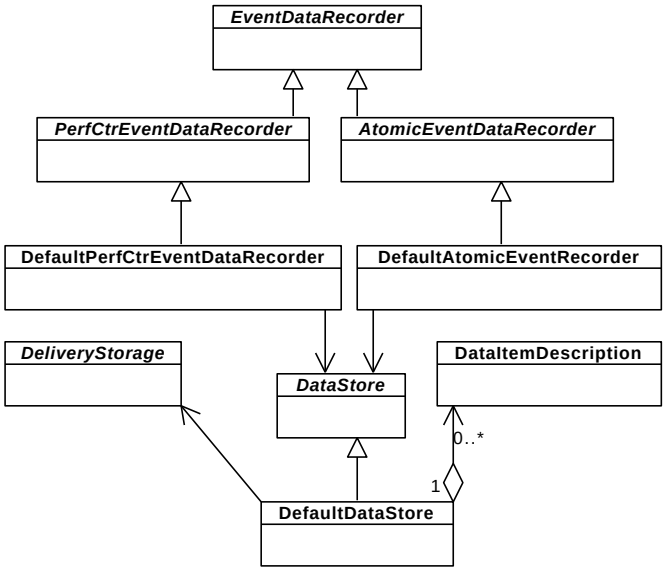


Figure 4.13: Data storage component

event processors to store data for its events and –DataStore classes that are used for unified storage of event data and measured data to the memory. While recorders are separate for every event source, data store should be single for all profiler.

4.6.1 Data recorders

As mentioned in previous section, event processors use the data recorders to store measured data and event data to the memory. The EventDataRecorder interface declares the root class of all recorders. It defines the method to register measurement contexts that will be stored using that recorder (registerMeasurementContext() and unregisterMeasurementContext()) and the setDataStore() method that associates the recorder with memory data storage. It is possible for one recorder to have more contexts registered.

Every event type has defined its recorder interface inherited from EventDataRecorder (except PERFCTR_JAVA type, but it is special case of PERFCTR event type only). For the atomic events, the AtomicEventDataRecorder interface is defined. It adds one method – record() that can be used to record one event and its data. In the implementation class (DefaultAtomicEventRecorder) the methods only passes its arguments to appropriate methods of the DataStore object. The registerMeasurementContext () method calls the method with same name, record() calls the storeData() method

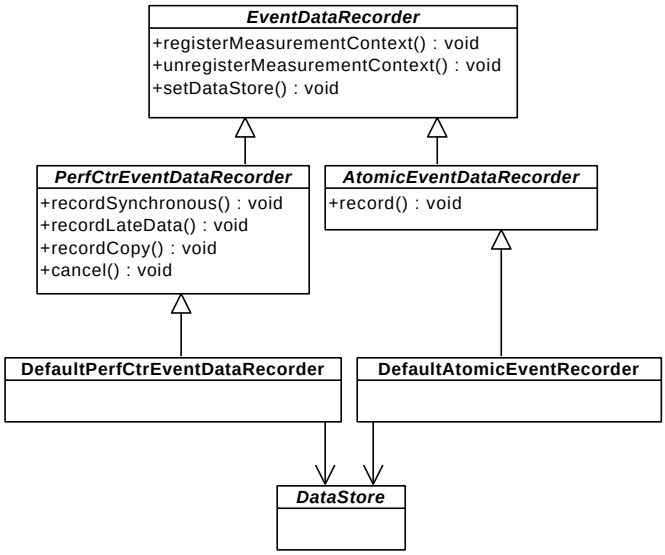


Figure 4.14: Data Recorders

and unregisterMeasurementContext() is empty, because data store does not support this operation.

The recorders for the performance counter overflow events are not so simple. The reason is the data are collected in two phases and we decided to use recorders to put them together again (literally, they are never apart, because they are stored within contexts, but we must be aware of both measurement phases and join the sampling status). The PerfCtrEventDataRecorder interface defines methods to support storing data in two phases (recordSynchronous() and recordLateData()), method to cancel events that has already recorded synchronous phase (cancel()) and method to record event with a copy of data of next event (recordCopy() – this is used when next event of the same type arrives while still processing previous instance). The implementation is in the DefaultPerfCtrEventDataRecorder class. The implementation of context registration method and setDataStore() method is same as for atomic events. If event is recorded by the recordCopy() method it is only put into linked list of other events recorded as copy. Method for recording synchronous data (recordSynchronous()) creates structure in which it stores event data pointer, result of sampling and pointer to measurement context with data. Then it puts this structure to a linked list and returns. The recordLateData() method then searches for record in that list that has stored event with same event ID as specified in the eventId parameter of the method. The record is removed from the list and a new result is created as a worse one from sampling of synchronous and late data. After this, list of events stored by recordCopy() method is traversed and events that were generated by the same processor event are linked

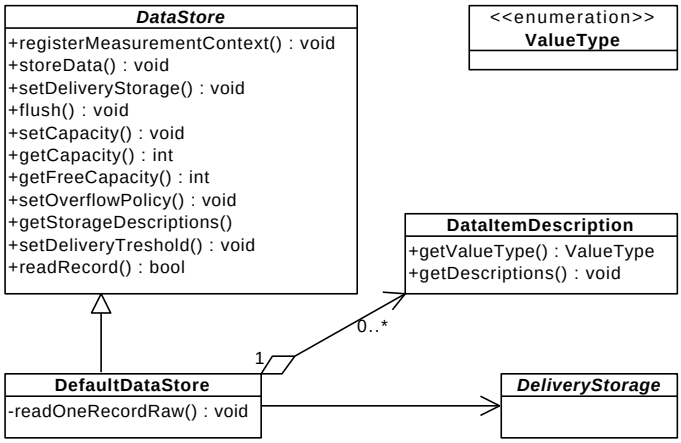


Figure 4.15: Data store

to one linked list using the next field of event data. This list is then passed to the storeData() method of data store. At request for event cancellation (cancel() method) the list with recorded synchronous data is traversed, and the record with appropriate event is destroyed.

4.6.2 Data store

The data store is a part that unifies all events in one place. It takes data from all events and stores them in memory and provides interface that no longer distinguish the event types. The interface supports two possible approaches how to deliver data to external storage. It can flush all data at once when it runs out of capacity or the external storage can asynchronously read the records one by one. The flushing approach brings overhead that counts to the application's threads while asynchronous reading needs its own thread and in total it is a little less effective, but does not affect application's threads. Simplified class diagram is displayed in the *Figure 4.15*.

The DataStore interface defines several methods – some are for configuration, other provide information about state of storage and the last methods manipulates the data. The interface supports following configuration methods:

- registerMeasurementContext() – the storage must store data in some common format and the storage must be effective. The storage needs a way to prepare its data structures for data it will be storing.
- setDeliveryStorage() – configures the external storage where the stored data will be delivered.

- setCapacity() – request to set size of internal buffers to support specified number of records for every registered context.
- setOverflowPolicy() – specifies what to do if the internal storage is full. FLUSH causes all stored events to be delivered to the external storage and IGNORE will ignore all events until storage is available.
- setDeliveryTreshold() – sets when the external storage will be notified that it should start reading data. The value is in per cents of capacity.

To inspect the state, we defined three methods – getCapacity() returns the size of internal buffers in number of data records, getFreeCapacity() tells free capacity for specified measurement context and getStorageDescriptions() returns descriptions of data that the data storage is storing in way with that the external storage can work. It returns a vector of description lists. The order is important, since the data that are passed from the memory storage to the external storage use an index to that vector.

In the data manipulation group of methods we defined the storeData() method, that is called by recorders to store event data and measured data. The flush() method collects all data stored in storage and passes them to the external storage. readRecord() method reads the oldest record from data storage stored with specified context.

Our implementation is in the DefaultDataStore class. To store the data it uses the separate memory buffers for all registered contexts. At context registration, the class creates description of the data that the context provides using its sensor descriptors. It also determines, how much memory is needed to store one record. It is simple sum of memory sizes required by the sensors. Then it allocates memory that is as large as $capacity * datalength$ and stores the pointer to the associative array. The key is the context ID. It has also two pointers to the data buffer (actually they are indices from 0 to capacity) – one points to the location that will be next written to and other to the location that will be next read from.

Then the data storing (the storeData() method) selects the appropriate buffer and stores the data to the location specified by writing pointer. The data are stored one by one item to the memory in their binary representation. This resembles using structures, but we cannot use compiler support for that so we must compute data offsets manually. If the number of written recordes reached the threshold defined by the setDeliveryTreshold() method (or its default value – set to 30 per cents), it calls the startReading() method of external storage class (the DeliveryStorage interface). To read the data we defined the readOneRecordRaw() method that reads one record from the specified buffer and location in that buffer. The reading is same as writing – we use the created data descriptions and read binary data from the memory buffer as it is a structure. Then the flush() method reads all records and passes them to external

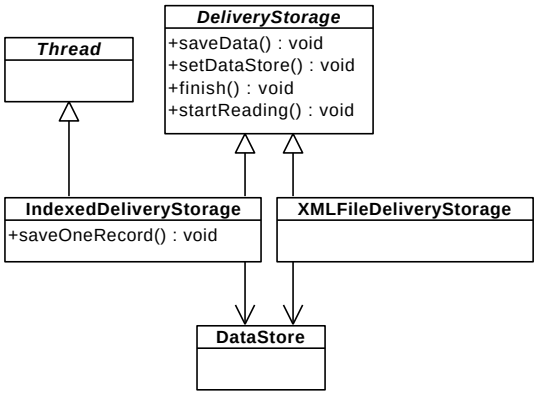


Figure 4.16: External storage component

storage, while readRecord() reads one record from the specified context's buffer.

4.7 External storage

The external storage component, respresented by the data delivery classes, is responsible for storing the data from the memory storage to the external devices. We have defined two implementations, indexed file storage and XML file storage. The storage to the XML file was created only for debugging purposes and we will not describe it. It is almost same as indexed file storage, except it stores only text data. The component diagram is displayed in the *Figure 4.16*.

The DeliveryStorage interface communicates with the memory storage (DataStore) and writes the data to the external devices. It defines following methods:

- saveData() – used in the flush operation – stores a set of data.
- setDataStore() – sets the data storage that will be processed.
- finish () – called by infrastructure at exiting from application – used to close files, stop threads, etc.
- startReading() – called by data storage when the capacity threshold is exceeded. The implementation should start reading data after being notified by this method.

Our implementation (the IndexedDeliveryStorage class) uses mechanism described in the *Section 3.9* to store the data. It defines private method (saveOneRecord()) that

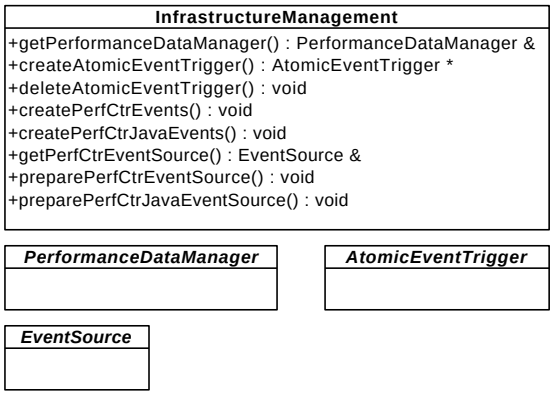


Figure 4.17: Infrastructure management

stores one data record. We believe it is not necessary to describe this procedure exactly, it only creates record in index file as structure and then stores data fields one-by-one to the data file and stores record length and position to the index record.

The implementation of the saveData() method is just loop that calls saveOneRecord() . To implement asynchronous data delivery, the class is defined as descendant of the Thread class. It makes possible to define run() method that can be executed in separate thread. Our implementation uses semaphore, that is raised in calls to the startReading() method. In the thread routine we have a loop that lowers the semaphore and then reads the available data from memory storage. The read data are stored by the saveOneRecord() method. The finish () method stops the thread and closes the files.

4.8 Infrastructure management

The infrastructure management part of the implementation provides a way for easy usage of other components. In principle, it is possible to use the infrastructure without management object, but it would be difficult. The InfrastructureManagement class (diagram is in the *Figure 4.17*) provides methods that create and destroy atomic event source (createAtomicEventTrigger() and deleteAtomicEventTrigger). The user only needs to specify the measurement context and then he or she can call the trigger () method to generate event. The contexts can be created using performance data manager, that is available by calling getPerformanceDataManager() method (manager is described in the *Section 4.4.3*). The createPerfCtrEvents and createPerfCtrJavaEvents methods prepare the event source for CPU performance counters and sensors for

reading their values. The preparePerfCtrEventSource() and preparePerfCtrJavaEventSource () methods are used to initialize performance counters event source to be able to process events and measure data specified in the context parameter.

When no longer needed, the user is obliged to destroy all created atomic event sources by calling the deleteAtomicEventTrigger method. All other objects created or passed to manager (such as the measurement contexts) are destroyed automatically by the manager.

4.9 Java interface and other tools

In order to use the infrastructure with Java, it must have a form of JVMTI agent. We define the Agent_OnLoad() entry point, that initializes JVMTI environment, sets JVM capabilities and registers JVMTI events. We are using the JVM initialisation and death events, and thread start and end events. In entry point we only enable VM init and death events. In VM initialisation event callback we create the environment for infrastructure, and start measurement of data. This cannot be done in Agent_OnLoad (), because there cannot be created new threads the infrastructure needs. Here we also enable thread start and end events. Their callbacks maintain associations between the Java and native threads, as described in the *Section 3.8.1.1.* In VM death event the infrastructure is stopped and destroyed.

We have created other supporting tools, that are worth explanation. The most important is the Thread class. It can run code defined in its descendants (in the run() method that is abstract in the Thread class) in separate thread. This works both for native and Java profiling. In native profiling, it just creates new POSIX thread, while if profiling Java the thread must be created using procedure described in the *Section 2.2.3.3.* The class determines if it is run in native environment or within Java and chooses the correct way. We are also using simple classes that provides encapsulated access to POSIX mutexes, spinlocks and SYSV semaphores.

Chapter 5

Evaluation

All profilers are exposing the tested applications to some overhead. It is important to keep the overhead as low as possible. In this chapter, we will measure the overhead of our tool. First, we will introduce benchmarks we have used and then provide the measurement results.

5.1 Benchmarks

Our tool is able to profile both Java and C++ code. We will focus on measurement of Java profiling, but we will also measure in C++. For Java, we have used three benchmarks:

- *Fhourstones* benchmark [20] – integer operation benchmark computing positions in modification of tic-tac-toe game. The results are in thousands of positions computed per second (Kpos/s). The higher values are better.
- *Sampler* benchmark from CCPSuite [18] – benchmarking suite that consists of set of measurements. We have chosen 3 benchmarks: two measurements of memory allocations and one measurement of thread lifecycle. The results are in processor's clock ticks – lower values are better.
- Our simple Java program – *TestingJavaProgMulti* – computes some integer calculations, using very frequent method calls. Uses two identical threads. The result is measurement of time it spent on processor using `time` command.

In C++ we measured using the C version of Fhourstones.

5.2 Measurements

We measured on two computer configurations:

- Single CPU Athlon XP (Barton) @ 1921 MHz, 512 KB cache, 1 GB system memory
- Dual CPU Pentium III (Coppermine) @ 800 MHz, 256 KB cache each, 512 MB system memory

For each benchmark, we made a set of measurements. The first measurement was always without profiling and the results serve as the base values. Then we measured the same benchmark with three or four different configurations of the tool. We decided to do the measurements with processor events that count the completed instructions. For Java, we used values 50000000, 100000000, 150000000 and 200000000 for overflow threshold. For C++, the values 500000, 1000000 and 1500000 were used. We executed every combination of benchmark and configuration several times and we will provide results as an arithmetic mean.

5.2.1 Fhourstones – Java

The benchmark can be run using following command:

```
$ java -Xmx70m SearchGame < inputs
```

The inputs file contains starting positions for calculation. For our measurements, we used the third value from inputs distributed with benchmark – 13333111. For every configuration we executed the benchmark 5 times. The results are following:

1x Athlon XP		
Configuration	**KPos/s**	**Events**
Base	1400	0
50000000	758	9620
100000000	808	4530
150000000	999	2470

2x Pentium III		
Configuration	**KPos/s**	**Events**
Base	635	0
50000000	624	4990
100000000	626	2490
150000000	629	1660

The difference in overhead is obvious. Generally, in profiling Java, almost all overhead is caused by GetStackTrace() function (details in the *Section 3.8.1.2*). In single-processor machine, the function must be run on the same processor as the benchmark and the overhead becomes visible. On multiprocessor, as shows the second measurement, the overhead is not so apparent. This is because the measurement is using separate thread and it is using the other processor. Also important is the difference in number of events. Since we are using number of completed instructions as events, the number of events can be interpreted as number instructions needed to finish the task. The difference is probably caused by the fact that in multithreaded case, all events that comes from processor running the measurement thread are stored only as copies (because that thread is processing events or is sleeping), and therefore not causing additional overhead. In single-processor case the code of measurement is probably not so often interrupted, because it may run less then the time required to complete measurement. But the used instructions are counted for the benchmarking thread and that is interrupted sooner then in multi-processor.

5.2.2 Fhourstones – C++

The benchmark is run in same manner as from Java. It must be compiled, we used the `-O3` optimisation level, and executed with data on standard input. We measured the benchmark with same data as the one for Java. Here are the results:

1x Athlon XP		
Configuration	**KPos/s**	**Events**
Base	2570	0
500000	2410	288000
1000000	2490	141000
1500000	2520	93000

2x Pentium III		
Configuration	**KPos/s**	**Events**
Base	1110	0
500000	1000	293000
1000000	1040	143000
1500000	1060	95000

In the C++ case, the overhead is much lower, even if the event frequency is 100 times bigger. Also, the event count is approximately equal in both computer configurations. No significant source of overhead (as opposed to Java's GetStackTrace()) is present in native version and it makes the overhead much smaller.

5.2.3 TestingJavaProgMulti

This benchmark does a simple integer computations. It has two identical threads. We measure total time spent on processor as provided by `time` command. We use the sum of *user* and *sys* times. We ran the benchmark 5 times on every configuration and presented results are the arithmetic mean of measured times in seconds. The results follows:

1x Athlon XP		
Configuration	**Time (s)**	**Events**
Base	58	0
50000000	79	5600
100000000	79	2800
150000000	78	1900

2x Pentium III		
Configuration	**Time (s)**	**Events**
Base	161	0
50000000	211	5000
100000000	192	2400
150000000	189	1600

From the measurement it is clean the advantage of using 2 processors is lost if the application has more threads. Interesting result are the almost same time in single-processor measurement. The third measurement has the lowest overhead, as expected, but the measurement at highest event frequency has better time then less demanding case. This is probably caused by fact that the events were triggered while processing previous event very often. These events are stored as copies of another event and it is free of the overhead caused by GetStackTrace(). The event count is also similar in this benchmark.

5.2.4 Sampler

This benchmark is a set of different measurements. Each measurement provides several readings, called samples. They are computed from pre-defined number of measurements and consist of minimal measured value, average value and maximum value. The data is stored in the Client.out file, in XML format. An example follows:

```
<Benchmark Type="Thread Locking">
 <Measurement>
  <Sample LoadBefore="2" LoadAfter="2">22297 23386 116092</Sample>
  <Sample LoadBefore="2" LoadAfter="3">22297 22936 52262</Sample>
```

```
  <Sample LoadBefore="2" LoadAfter="3">22297 23710 130906</Sample>
  <Sample LoadBefore="3" LoadAfter="9">22297 22580 34861</Sample>
  <Sample LoadBefore="8" LoadAfter="4">22297 22661 36595</Sample>
 </Measurement>
</Benchmark>
```

We have chosen three measurements from two benchmarks. From *Memory Allocation* we picked the measurement of 10000 allocations of 1023 and 8121 bytes long blocks. The other measurement is from the *Thread Lifecycle* benchmark, and we used measurement with 128 threads. all used measurements provide 5 samples and we executed the benchmark 3 times for every configuration, getting 15 samples in total. We provide average values in following results. We ran the benchmark on both machines, but since the benchmark takes very long to execute, we did not manage to keep the single-processor machine at low load and the results are not reliable. Therefore we provide results for dual processor machine only here:

2x Pentium III			
	Benchmark		
Configuration	Allocation 1023 10^6 ticks	Allocation 8191 10^6 ticks	Threads 128 10^5 ticks
Base	193	814	186
50000000	197	896	264
100000000	190	841	276
150000000	189	815	274
200000000	188	803	266

In the memory allocation benchmarks the overhead is low (and sometimes surprisingly negative). The reason is probably the fact that the Sampler is written in Java, but it uses some native code parts. Moreover, we noticed that the benchmarks with the same configuration tended to have better results if run multiple times. The third benchmark is using only Java language and the overhead is very high again. Again, the configuration with the highest event rate has better results then other profiling configurations. We think the reason is the usage of event data copies instead of exact measurement if two events appear in same time.

5.2.5 Summary

As the measurements implies, the overhead of Java profiling is very high, ranging from 20 to 50 per cents. This is caused by Java code position inspection, that takes aproximately 20 millions processor clock ticks for each event. The native profiling has acceptable overhead of 2-10 per cents depending on configuration. For native

code, minimal event threshold was set to 500000 events and we do not recommend to set it lower.

5.3 Comparison

Here we will compare the overhead to one of the commonly used JAVA profilers – HPROF. It takes, like our tool, the form of agent. We used HPROF with *Fhourstones* and *TestingJavaProgMulti* benchmarks. The HPROF was configured with option `cpu=samples` that orders the profiler to create statistical profile. We ran both of them several times and here is the table with average results, measured on the single-processor machine:

1x Athlon XP	
Benchmark	**Result**
Fhourstones	697 Kpos/s
TestingJavaProgMulti	79.5 seconds

As shows the results, the overhead of our tool is slightly lower that the overhead of HPROF, and we think it cannot be different with any tool that uses JVMTI or similar library.

Chapter 6

Conclusion

6.1 Summary

The main goal of this work – to devise a tool that is able to profile Java and native code using data from hardware, operating system and virtual machine – was accomplished in priciple. Specifically:

- A method for communication between the profiling interrupt, intercepted in kernel space, and the monitoring agent, running in user space, was devised (*Section 3.7*).
- A method for querying the position of the virtual machine from within the asynchronously invoked monitoring agent was devised (*Section 3.8*).
- The framework for performance data collection from [2] was adopted for use in the combination of native and interpreted environments.
- A format for storing the collected performance data, with separation of index and data for efficient random access, was designed (*Section 3.9*).
- The entire approach was prototyped in a tool that runs on current version of Linux and Java platforms with very decent stability given the number of low level assumptions that the prototype relies on. The tool is designed for running on x86 Linux systems with processors not based on NetBurst microarchitecture and JVM from Sun.

From the practical perspective, the usability of the prototype is limited due to high overhead of Java profiling. Even if this overhead is not higher than for other sampling profilers (like HPROF), it lowers the advantages provided by the performance

counters. For native code, the overhead is acceptable. Still, the prototype is unique in its ability to combine profiling of both interpreted and native code, which was one of the main goals of the thesis.

Besides the major goal of designing and implementing the prototype, other potentially useful output was delivered as a part of the thesis:

- We have created a comprehensive overview of hardware performance counters (*Section 2.1.1*), because in the documentation available from CPU vendors the description is split in many parts.
- We have also provided the description of handling of performance counter in OProfile for x86 machines, which the official documentation lacks (*Section 2.1.2.2*).
- We have also investigated a number of ways how the kernel interrupt handler can communicate with userspace without delays (*Section 3.6* and *Section 3.7*) and options of calling JVMTI functions from the signal handler (*Section 3.8*).
- We have proposed adjustments to the performance monitoring framework [2](*Section 2.3.1.2*).

6.2 Future work

The biggest disadvantage of Java profiling is the overhead of the code position detection. The further investigation for possibilities hot to do this task would be very helpfull. Moreover, the opensource version of JDK is now available, what make the option to change the mechanism used by JVMTI possible, but the task would be very difficult. Maybe also the option to use bytecode instrumentation instead of GetStackTrace() would be worth the effort.

The created tool could be improved of options to identify the code position in the native code. The tool currently provides only absolute value of program counter, but the tool could cooperate with OProfile and use its infrastructure to provide annotated source. To accomplish this, the OProfile data that are stored to the userspace should be extended by the absolute value of program counter and process identificator. From this couple the source code position could be determined using OProfile's tools.

Also the mechanism of performance data measurement is not ideal. It could be modified to use multiplexing of contexts or use one data snapshot for measurement of data that can be read from common data source, as proposed in [2]. One of

the future task could be to use some more common (and reliable, maybe) management of CPU performance counters. The most commonly used options are PAPI library and perfctr Linux kernel extension. As mentioned, both of them would need modifications.

The disadvantage of the tool for native code is the need to modify source codes and include the infrastructure. The methods how to attach the library to running application, or how to start the application to use the profiling library could be elaborated.

Bibliography

[1] BIOS and Kernel Developer's Guide for the AMD Athlon 64 and AMD Opteron Processors, 2006 – http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF

[2] Bulej L.: Connector-based Performance Data Collection for Component Applications, Ph.D. Thesis, Charles University, 2007 [p. 52-97]

[3] GNU gprof Manual – http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

[4] Hauswirth M, Sweeney P., Diwan A., Hind M.: Vertical profiling: understanding the behavior of object-priented applications, Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 24-28, 2004, Vancouver, BC, Canada

[5] Intel 64 and Ia-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1 – http://developer.intel.com/ design/processor/manuals/253668.pdf

[6] Intel 64 and Ia-32 Architectures Software Developer's Manual – Volume 3B: System Programming Guide, Part 2 – http://www.intel.com/ design/processor/manuals/253669.pdf

[7] Intel VTune Performance Analyzer – http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

[8] Jikes RVM web site – http://jikesrvm.org/

[9] JVM Tool Interface Reference – http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[10] Levon J.: OProfile Internals – http://oprofile.sourceforge.net/doc/ internals/index.html

[11] McDougall R., Mauro J., Gregg B.: Solaris Performance and Tools, Prentice Hall, 2006, ISBN 0-13-156819-1

[12] O'Hair K.: HPROF: A Heap/CPU Profiling Tool in J2SE 5.0, 2004 – http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[13] Open JDK home page – http://openjdk.java.net/

[14] OProfile homepage – http://oprofile.sourceforge.net/news/

[15] PAPI homepage – http://icl.cs.utk.edu/papi/

[16] Pettersson M.: Linux performance monitoring counters kernel extension – http://user.it.uu.se/~mikpe/linux/perfctr/

[17] Prasad C.K., Ramchandani R., Rao G., Levesque K.: Creating a Debugging and Profiling Agent with JVMTI – 2004 – http://java.sun.com/developer/ technicalArticles/Programming/jvmti/

[18] Sampler benchmark – http://dsrg.mff.cuni.cz/~ceres/prj/CCPsuite/

[19] Sweeney P., Hauswirth M., Cahoon B., Cheng P., Diwan A., Grove D., Hind M.: Using hardware performance monitors to understand the behavior of java applications, Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium, May 06-07, 2004, San Jose, California

[20] Tromp, J: The Fhourstones Benchmark – http://homepages.cwi.nl/~tromp/ c4/fhour.html

Appendix A

Installation and usage

A.1 Installation

The system requirements are following:

- x86 system with non-NetBurst processor (all x86 processors except Pentium 4-based ones).
- Linux operating system compatible with 2.6.24 version of kernel.
- JDK from Sun, version 1.5 at least.
- Development version of expat library.
- OProfile (most recent version recommended - currently 0.9.3).
- GCC compiler with C and C++ support (version 4 and newer recommended).
- Linux kernel sources - 2.6.24 version (2.6.24.X works, too).

The installation consists of two phases:

1. Kernel patching and configuration.
2. Userspace compilation and setup.

To patch the kernel, change working directory to the directory with kernel sources (`kernel-source` in example). Copy kernel patch from enclosed CD (file named msi-kernel-2.6.24.patch in the src/patch directory) to some accessible location (we will refer to the file as `patch-dir/patch`). Then use the `patch` command to to apply patch. The following example illustrates this procedure:

```
$ cd kernel-source
kernel-source$ patch -p1 < patch-dir/patch
```

After this, prepare a kernel configuration suitable for your system. The OProfile must be compiled as in-build, not as a module. It is the option `CONFIG_OPROFILE` in the .config file, available in the "Instrumentation Support" submenu in menuconfig. Also make sure that Local APIC is enabled. Then the kernel can be built and installed. After reboot, the presence of the patched code can be checked by following commands:

```
# opcontrol --init
# ls -l /dev/oprofile/signal
-rw-rw-rw- 1 root root 0 2008-04-10 12:51 /dev/oprofile/signal
```

If the signal file is available, the patch was properly installed.

For the userspace part, copy the sources from CD (available in the src/infrastructure directory), and make sure the `JDK_HOME` environment variable points to the installation of JDK. Then execute `make depend`, `make` and `make suidwrappers` commands and change access right to the suid wrappers:

```
# chown root:root initoprofile startoprofile stopoprofile
# chmod +s initoprofile startoprofile stopoprofile
```

The functionality of infrastructure can be tested by running `test2` test:

```
$ ./test2
```

In case of error message about unknown event, the test is prepared for different type of CPU and the event that is set for the test is not available for current CPU. The solution is explained in *Usage*.

A.2 Usage

To use the infrastructure with Java, copy `libinfrastructure.so` and suid wrappers to the working directory of the application. The EmptyThread.java file must be available through CLASSPATH environment variable. Make sure, the suid wrappers have correct access rights. Use following command to run JavaApplication with profiling:

```
$ export LD_LIBRARY_PATH=.
$ java -agentlib:infrastructure JavaApplication
```

or

```
$ export LD_LIBRARY_PATH=.
$ java -agentlib:infrastructure=outputname JavaApplication
```

to store the results in files with name outputname.

We have not developed any configuration utility or data format for selecting the events and sensors for the measurement.Because of this, the measurement must be configured in infrastructure's source code directly. For Java, the configuration is in the JavaInterface/agententry.cpp file, for C++ profiling the infrastructure must be included and configured directly from its code.

The following code is in the JavaInterface/agententry.cpp file:

```
//initialize the infrastructure
IM = new InfrastructureManagement( output_name_option ?
    output_name_option : "javaoutput" );

//Create performance couter event(s)
std::list< PerfCtrEventSpec > events;
PerfCtrEventSpec ev_spec1( "RETIRED_INSNS", 50000000 );
PerfCtrEventSpec ev_spec2( "DATA_CACHE_REFILLS_FROM_L2", 500000, 0x1f )
    ;

events.push_back( ev_spec1 );
events.push_back( ev_spec2 );

IM->createPerfCtrJavaEvents( events );
```

This code initializes the infrastructure and selects the performance events to count. The user must specify the event name and the overflow threshold. The third parameter is optional and can specify the event mask. Maximum number of events is defined by CPU (4 for AMD and 2 for Intel CPUs). The IM->createPerfCtrJavaEvents (events); command registers the events and prepares sensors that can read values of the counters.

Next task is to create the measurement contexts. The following code does this task:

```
//create context(s)
PerformanceDataManager & pdm = IM->getPerformanceDataManager();
std::vector<std::string> sensors1;
sensors1.push_back( "Random" );
sensors1.push_back( "ProgTotalMemory" );

std::vector<std::string> sensors2;
sensors2.push_back( "SysFreeMemory" );
```

```
MeasurementContext * context1 = pdm.createMeasurementContext( sensors1
    );
MeasurementContext * context2 = pdm.createMeasurementContext( sensors2
    );
```

The first step is to acquire the performance data manager that can create contexts. Then we specify the sensor names in the lists and create the contexts using createMeasurementContext() method. We support following sensors:

- Random – returns random number.
- ProgTotalMemory – gets the total memory consumed by application.
- ProgSwappedMemory – gets the amount of the application's memory that is swapped.
- SysFreeMemory – gets the total amount of free memory in the system.
- SysUsedMemory – gets the total amount of used memory in the system.
- SysSwappedMemory – gets the total amount of swapped memory in the system.
- NetSentSensor – gets total amount of data sent on all network interfaces.
- NetReceivedSensor – gets total amount of data received on all network interfaces.

The number of sensors in one context is not limited.

After creation of contexts, they must be associated with events. This is done by the following code:

```
std::list< MeasurementContext* > contexts;
contexts.push_back( context1 );
contexts.push_back( context2 );

IM->preparePerfCtrJavaEventSource( contexts );

IM->getPerfCtrEventSource().enable();
```

The code puts the contexts to the list. Here, the order of items is important. The first context is associated with first event defined in events list, the second with second and so on. The IM->preparePerfCtrJavaEventSource(contexts); associates events with contexts and creates event source. The last line starts the measurement.

So, for configuration the change of these parts of code is required. The list of available events for running CPU can be obtained from the command:

```
$ opcontrol --list-events
```

For the C++ code, the infrastructure must be included directly to the application and configured from its code. This is done usually in the file that contains the main() function. An example is in the *Listing A.1.*

```
#include <infrastructuremanagement.h>

int main()
{
    MeasurementInfrastructure::InfrastructureManagement IM("testoutput"
        );

    //Create performance couter event(s)
    std::list< MeasurementInfrastructure::PerfCtrEventSpec > events;
    MeasurementInfrastructure::PerfCtrEventSpec ev_spec1( "
        RETIRED_INSNS", 500000 );
    events.push_back( ev_spec1 );

    IM.createPerfCtrEvents( events );

    //create context(s)
    MeasurementInfrastructure::PerformanceDataManager & pdm = IM.
        getPerformanceDataManager();
    std::vector<std::string> sensors1;
    sensors1.push_back( "Random" );

    MeasurementInfrastructure::MeasurementContext * context1 = pdm.
        createMeasurementContext( sensors1 );

    std::list< MeasurementInfrastructure::MeasurementContext* >
        contexts;
    contexts.push_back( context1 );

    IM.preparePerfCtrEventSource( contexts );
    IM.getPerfCtrEventSource().enable();

    /* ... code of application ... */
}
```

Listing A.1: Example of using infrastructure in C++ program

It is from test2 program mentioned in *Installation*, the code is available in the Tests/test2.cpp file. So, if you get error that the event does not exists, experiment with "RETIRED_INSNS" string. The program must be linked with libinfrasructure.so library and compiled with infrastructure source directory in include path.

The running profiler creates profile in the files that begin with the name specified in the InfrastructureManagement constructor, or name specified as parameter in Java

agent (the default name is "javaouput" for Java). In the file basename.desc.xml is description of measured data. You can pass it as the only parameter of the `reader` program that is available in the directory where the infrastructure was build. This program interprets the data in human-readable format.

Appendix B

Content of enclosed CD-ROM

`/README` – description of context of the CD.

`/src/` – source codes of applications and kernel patch.

- `infrastructure/` – code of userspace part of profiling tool.
- `patch/` – patch for required kernel modifications.
- `java_tests/` – simple testing Java programs.
- `test_agents/` – simple testing Java agents.

`/doxygen/` – generated documentation in HTML format.

`/thesis/` – electroic version of this document.

`/results/` – some results measured by the created tool.