Tool Demonstration: Overseer — Low-Level Hardware Monitoring and Management for Java

Achille Peternier, Daniele Bonetta, Walter Binder, Cesare Pautasso
University of Lugano (USI)
Faculty of Informatics
Lugano, Switzerland
{firstname.lastname}@usi.ch

Abstract
The high-level and portable nature of the Java platform allows applications to be written once and executed on all the supported systems. However, such a feature comes at the cost of hardware abstraction, making it more difficult or even impossible to access several low-level functionalities. Overseer is a Java framework that makes it possible on Linux systems by simplifying access to real-time measurement of low-level data such as Hardware Performance Counters (HPCs), IPMI sensors, and Java VM internal events. Overseer supports functionalities such as HPC-management, process/thread affinity settings, hardware topology identification, as well as power-consumption and temperature monitoring. In this paper we describe Overseer and how to use it to extend Java applications with functionalities not provided by the default runtime. A public release of Overseer is available.

Categories and Subject Descriptors D.2.2 [Software Engineering]: Design Tools and Techniques — Software Libraries

General Terms Measurement, Performance

Keywords Hardware performance counters, hardware surveying, thread scheduling, IPMI, JVMTI

1. Introduction
Since it has become difficult to further increase the clock rate of processors, nowadays chip manufacturers are delivering more processing power by increasing the number of cores in their processors. Modern hardware infrastructures for enterprise-level server applications feature several processing units aggregated into one or more CPUs that are often part of a Non-Uniform Memory Access (NUMA) architecture.

To simplify exploitation of such highly parallel computational power, the standard Java class library provides synchronizers and concurrent data structures to build scalable applications [5]. However, the standard Java class library does not offer enough support for developing applications that are aware of the hardware they are running on. This hardware awareness could enable further optimizations based on machine-level data (e.g., improved benchmarking with Hardware Performance Counters (HPCs), configuring thread affinities and NUMA settings to improve processor cache and memory usage, etc.).

In order to give developers a detailed understanding of the runtime hardware-level behavior of their Java applications, and to give them the necessary instruments to influence the way software processes are performed by the hardware Processing Units (PUs, either cores or SMT units according to the machine), we developed a low-level monitoring and management framework called Overseer.

In this paper we describe the main components of our framework, giving details on: their APIs, the native elements they interact with, and we describe some real usage examples that can be used to demonstrate the functionality of the framework.

2. Overseer
The Overseer framework is a set of Java classes interfacing native C/C++ libraries to implement specific low-level hardware functionalities. The framework is composed of three main components, each dedicated to a specific domain of tasks and exposed to the application level through standard Java classes. The three components are OverHpc, OverAgent, and OverIpmi (see Figure 1). Each component is independent from the others and lazily loaded on the first use. Since most of the underlying libraries used by Overseer are written in C/C++, their initialization and de-initialization is transparently managed by our framework to guarantee correct release of the allocated resources.

3. OverHpc
The OverHpc component addresses dynamic information acquisition from hardware sources such as HPCs, processor clocks, and...
3.2 Hardware Topology

When initialized, OverHpc performs a hardware discovery to identify the underlying architecture. This survey reports information on the number and kind of available CPUs, including the way processor caches are organized into shared levels (if any), and how NUMA nodes (if present) are aggregated into a group of one or more CPUs. This feature conceptually extends what Java natively offers through the Runtime.availableProcessors() method. OverHpc completes this information including the way PUs are distributed over several CPUs, the PU ID numbers of cores sharing a common cache, etc. These complementary data permit the creation of so called “affinity groups” based on a list of PU IDs physically running on a same NUMA node or sharing a cache.

3.3 Thread Affinities

While HPCs and hardware topology reports are mainly passive tools that can be used to observe the runtime behavior of Java applications, OverHpc also features active instruments to interact with the OS scheduler by customizing the list of PUs to use for the execution of selected threads. To do so, given a thread process ID, OverHpc allows developers to specify the list of one or more PUs to be used for its execution.

For example, matching the execution of threads accessing and reusing similar data together with the affinity groups described in the previous section can improve performance due to a better exploitation of locality (both at the NUMA memory and processor cache levels). Figure 2 shows a Java code snippet using OverHpc to bind the execution of the current thread to the PU number 1, and to measure the amount of cache misses generated within that thread.

![Figure 2. OverHpc usage example restricting the execution of the current thread to PU #1 and measuring cache misses.](image-url)

3.4 Implementation

OverHpc relies on libpfm4\(^1\) for the management of HPCs. Linux kernels natively embed HPC support since version 2.4.30: the libpfm4 library allows direct access to hardware events without kernel patches or OS modifications required by other frameworks. libpfm4 also includes kernel-level software events, such as number of thread migrations and total invocations of some specific kernel API methods. These events are available through OverHpc as well. Hardware topology information is gathered through the libhuauth library\(^2\). Finally, OS scheduling and thread-related operations are implemented through methods supported by the Linux kernel API (sched.h).

\(^{1}\)http://perfmon2.sourceforge.net/
\(^{2}\)http://www.open-mpi.org/projects/hwloc/

<table>
<thead>
<tr>
<th><strong>Table 1.Overview of the OverHpc API.</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>String[] getSupportedEvents()</td>
</tr>
<tr>
<td>int initEvents(String[] events)</td>
</tr>
<tr>
<td>int bindEventsToPu(int pu)</td>
</tr>
<tr>
<td>int bindEventsToThread(int pid)</td>
</tr>
<tr>
<td>void start()</td>
</tr>
<tr>
<td>void stop()</td>
</tr>
<tr>
<td>double getEvFromPu(int pu)</td>
</tr>
<tr>
<td>double getEvFromThread(int pid)</td>
</tr>
<tr>
<td>int getThreadId()</td>
</tr>
<tr>
<td>int setThreadAffinity(long mask)</td>
</tr>
<tr>
<td>long getThreadAffinity(int pid)</td>
</tr>
<tr>
<td>Object getHardwareInfo()</td>
</tr>
</tbody>
</table>

\[1\] Acquire current thread process ID:
OverHpc ohpc = OverHpc.getInstance();
int thisPID = ohpc.getThreadId();

\[2\] Force thread to run on PU #1 only:
ohpc.setThreadAffinity(thisPID, 1);

\[3\] Measure cache misses for current thread:
ohpc.initEvents("PERF_COUNT_HW_CACHE_MISSES");
ohpc.bindEventsToThread(thisPID);

\[4\] ... do something
long misses = ohpc.getEventFromThread(thisPID);
onThreadNew(int pid, String name) Callback notifying the process ID and Java thread name of a newly created thread.

onThreadEnd(int pid, String name) Callback notifying the process ID and Java thread name of a terminated thread.

float getThreadSysUsage(int pid) Returns a thread CPU usage compared to the whole system.

float getThreadRelUsage(int pid) Returns a thread CPU usage compared to the other threads running in the JVM.

Table 2. Overview of the OverAgent API.

4. OverAgent

Since HPCs and affinity groups are typically used on a per-thread basis, Overseer provides a simple mechanism to intercept and notify the Java application of the creation and termination of threads within the JVM. Such functionality is particularly helpful to immediately apply specific HPC measurements to all threads running within the application. Similarly, it can be used to implement customized scheduling policies. These policies are then applied directly during the creation of each thread, making sure that subsequent memory allocations or processor cache usage happen after the thread has started to run with the proper settings.

OverAgent is based on a native JVMTI agent that interacts with the JVM at a lower level than the Java application, intercepting events such as thread creation and termination. Whenever a thread is created or terminated, OverAgent sends a notification back at the Java level through a callback invoking a method specified by the Java application. In this way, events happening at the JVM level can be exploited at the application level.

OverAgent invokes the callback with two arguments: the OS process ID (relative to the thread involved), and its Java name. Like the process ID functionalities offered by OverHpc, the information obtained through the OverAgent callback can be used later as parameter for the methods described in the previous section.

As an additional feature, OverAgent gives feedback about the CPU usage of each thread allocated within the JVM. This information is given either with an absolute percentage (that is, the CPU time spent by a thread compared to the total CPU time used by all the Java threads of the application). A brief summary of the OverAgent main API methods is reported in Table 2. Figure 3 shows an example of customized callback for OverAgent defined at the Java application level. This callback prints a message when a new thread is created or an existing one ends. The process ID of each thread is appended to the message. Additionally, to show how to combine OverHpc with OverAgent, when a new thread is created its scheduling is restricted to PU number 3.

4.1 Implementation

OverAgent is a native C/C++ library written using the JVMTI interface. The agent registers four Java events: VM start, VM death, thread start, and thread end. OverAgent keeps track of all these events, which are immediately notified to the Java application level using JNI methods to interact with the C/C++ library. On state-of-the-art JVMs, interception of the aforementioned JVMTI events introduces only minor overhead.

Thread CPU times are gathered by analyzing the number of jiffies (i.e., number of ticks of the system timer interrupt) reported in the Unix /proc file system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getNumberOfSensors()</td>
<td>Returns the number of available sensors.</td>
</tr>
<tr>
<td>String getSensorName(int id)</td>
<td>Returns the name of a sensor given its number.</td>
</tr>
<tr>
<td>double getSensorValue(int id)</td>
<td>Returns the current value reported by sensor given its number.</td>
</tr>
</tbody>
</table>

Table 3. Overview of the OverIpmi API.

5. OverIpmi

Besides data gathering on HPCs, architectural topology, and CPU usage, the Overseer framework completes its hardware report by acquiring feedback from sensors compatible with the Intelligent Platform Management Interface (IPMI), a standardized interface used by system administrators to manage computer systems and monitor their operations. Typically, professional server machines feature a series of sensors to monitor metrics such as case temperature or power consumption. OverIpmi brings at the Java level a simplified API for enumerating, initializing, and gathering data from these sensors.

A brief summary of the main OverIpmi API methods is reported in Table 3. Figure 4 shows a code snippet looking for the power consumption sensor and reading data from it.

5.1 Implementation

The IPMI standard is a sophisticated and complex abstraction layer allowing administrators to remotely monitor a set of machines in a data center. The OverIpmi library enables Java IPMI information reading by exposing only a few high-level methods, based on the C library FreeIpmi
\footnote{\url{http://www.gnu.org/software/freeipmi/}}
, for discovery, initialization, and data retrieval from sensors.

6. Related Work

HPCs represent one of the privileged sources of information to improve software performance \cite{3}. HPCs are widely adopted and integrated in the software development cycle \cite{2} and an increasing number of tools for accessing and manipulating performance counters has been proposed.

As an example, the Performance Application Programming Interfaces (PAPI) library \cite{2} is a widely adopted tool for measuring

\footnote{\url{http://www.intel.com/design/servers/ipmi/}}

\footnote{\url{http://www.gnu.org/software/freeipmi/}}

![Figure 3](http://example.com/overagent.png)  
**Figure 3.** OverAgent example of customized callback to manage the notification of thread creation and termination.
HPCs. This library offers a high-level, platform-independent access to CPU counters, providing developers with a standard way to access specific platform related counters as well as generic platform-independent counters. A new version of the library, called Component-PAPI (PAPI-C) [10] has been announced in late 2009. The new library extends the standard PAPI framework with the possibility to obtain information not only from CPU related events, but also from other sources such as GPUs, memory interfaces, network cards, as well as BIOS, ACPI and LM sensors.

Compared to PAPI, our library relies directly on the Linux kernel, thus providing more precise measures [12]. The high precision comes at the price of portability. Since the main target of our library are server-class Java applications running under Unix environments, this is not an issue in our case.

HPCs are not always the best solution in term of accuracy, as for instance they can not be fully trusted for time measurement [4]. However, counter-based solutions have shown their effectiveness in several real-world approaches, like memory optimization [11], thread scheduling [9], real-time power estimation [8], and processor workload and frequency scaling optimization [7]. All these optimization techniques have been implemented using ad hoc specialized solutions. The Overseer framework enables the implementation of similar optimizations in any Java application.

7. Applying Overseer

In this paper we present concrete code examples and use cases, showing the simplicity of use of our framework. The Overseer framework can be easily employed in existing Java applications for hardware-oriented performance tuning, dynamic analysis, and evaluation.

The framework has already been adopted in different contexts to optimize service-based Java applications on multicore machines [1, 6], and we are actively using it in other ongoing research projects. For instance, in Figure 5 we show a recently developed SOA benchmarking tool that relies on the Overseer framework. A presentation of this tool completes the live demonstration of our work at the conference.

A public binary release of the Overseer framework is available\(^3\), and an open-source release of the framework is scheduled for launch at the PPPJ2011 conference. To further increase the usability of our framework, we plan to make our components available also through a JMX interface.

Acknowledgments

This work is funded by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22 127366).

References


\(^3\)http://sosoa.inf.unisi.ch/