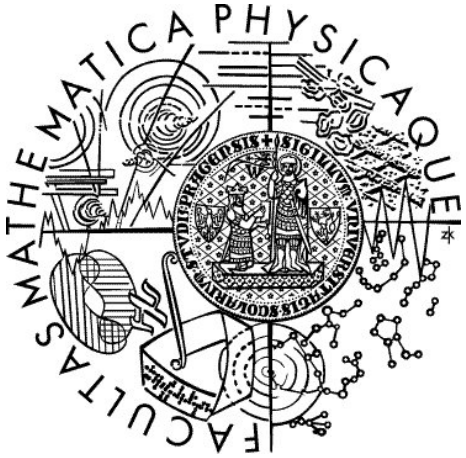
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Dráb

Extending Java Performance Monitoring Framework with Support for Windows Performance Counters

Department of Software Engineering

Supervisor: Ing. Lubomír Bulej Ph.D.

Study programme: General Computer Science

Specialization: Computer Science

Prague 2012

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Extending Java Performance Monitoring Framework with Support for Windows Performance Counters

Autor: Martin Dráb

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ing. Lubomír Bulej Ph.D.

Abstrakt:

Java Performance Measurement Framework (JPMF) je javovská knihovna, která si klade za cíl poskytnout aplikacím rozhraní dovolující sbírat výkonnostní statistiky různého charakteru nezávisle na operačním systému. Od specificky zaměřených měřících aplikací se tento framework liší větší mírou obecnosti – dovoluje tvůrci měřícího experimentu libovolně určit druhy statistik, které se mají z testovaného prostředí sbírat, a v jakých momentech se mají sbírat.

Cílem této bakalářské práce je vytvořit knihovnu, která umožní sbírat výkonostní statistiky na operačních systémech Microsoft Windows, a integrovat ji do frameworku. Tímto krokem dojde k značnému zvýšení nezávislost frameworku na operačním systému.

Klíčová slova: JPMF, data source, Windows, měření statistik, registry

Title: Extending Java Performance Monitoring Framework with Support for Windows Performance Counters

Author: Martin Dráb

Department: Department of Software Engineering

Supervisor: Ing. Lubomír Bulej Ph.D.

Abstract:

Java Performance Measurement Framework (JPMF) is a library that allows to collect performance data from underlying operating system. The main goal of the framework is to provide a way of performance data measurement regardless of the application under test. This goal sets the framework apart from many ad-hoc performance measurement solutions targeted at specific applications or middleware platforms. Such solutions collect certain performance data at fixed points of the execution of the application under test.

The main goal of this thesis is to implement a library that allows to collect performance statistics of various kinds on machines running Microsoft Windows operating system. The library should be integrated into the framework, which extends its portability.

Keywords: JPMF, data source, Windows, performance measurement, registry, caching

Contents

1. Introduction

Benchmarking is a way how to examine internal behaviour of the target system or application. It reveals how many resources the target uses during its individual operations. The big advantage of the method is that it can be performed without particular impact on the target. The method can be used to determine which application or framework is more suitable to achieve certain task. There are many libraries, frameworks and applications that do nearly the same things and their different consumption of resources, like physical memory or processor time, might be crucial in some situations.

Nonintrusive benchmarking is not a good solution in case we are interested in some more details about the resources consumed by the target system. The benchmarking usually shows how much memory and processor time the target uses, or whether it communicates with file system or over the network and how big this activity approximately is. However, it usually does not reveal files and registry objects the target accesses, or its partners in the network communication. Such a task might be accomplished by methods that are either more intrusive (e.g. binary modifications of the target), or require special privileges to the operating system where the experiment runs (e.g. filter drivers).

Java Performance Measurement Framework (JPMF) is a library that allows to collect performance data from underlying operating system. The main goal of the framework is to provide a way of performance data measurement regardless of the application under test. This goal sets the framework apart from many ad-hoc performance measurement solutions targeted at specific applications or middleware platforms. Such solutions collect certain performance data at fixed points of the execution of the application under test. JPMF allows its user to define these points of performance data collection. It also allows to decide which performance data should be measured for each such point.

Therefore, the framework might be understood as a generalization over benchmark systems targeting specific applications. Bigger overhead is the usual price for generalization, and this case is not an exception. JPMF attempts to be independent on operating system and hardware platform and pays for it by more complicated internal architecture.

Main goal of this thesis is to implement a data source and integrate it into JPMF infrastructure. The data source should allow the framework to access performance data counters found in the Windows NT family of operating systems. The framework is already able to collect some performance data from Linux and Solaris systems.

JPMF design is described in great detail in [1]. For this work, only a small part of the framework is important; hence only a short description of the framework is required. And this is the main goal of this chapter.

1.1 JPMF Overview

Basic architecture of the framework is shown in Figure 1.1 [1]. The framework consists of five main subsystems: Management Infrastructure, Event Sources, Performance Data Access, Event Processing and Data Delivery. Except Perfor-

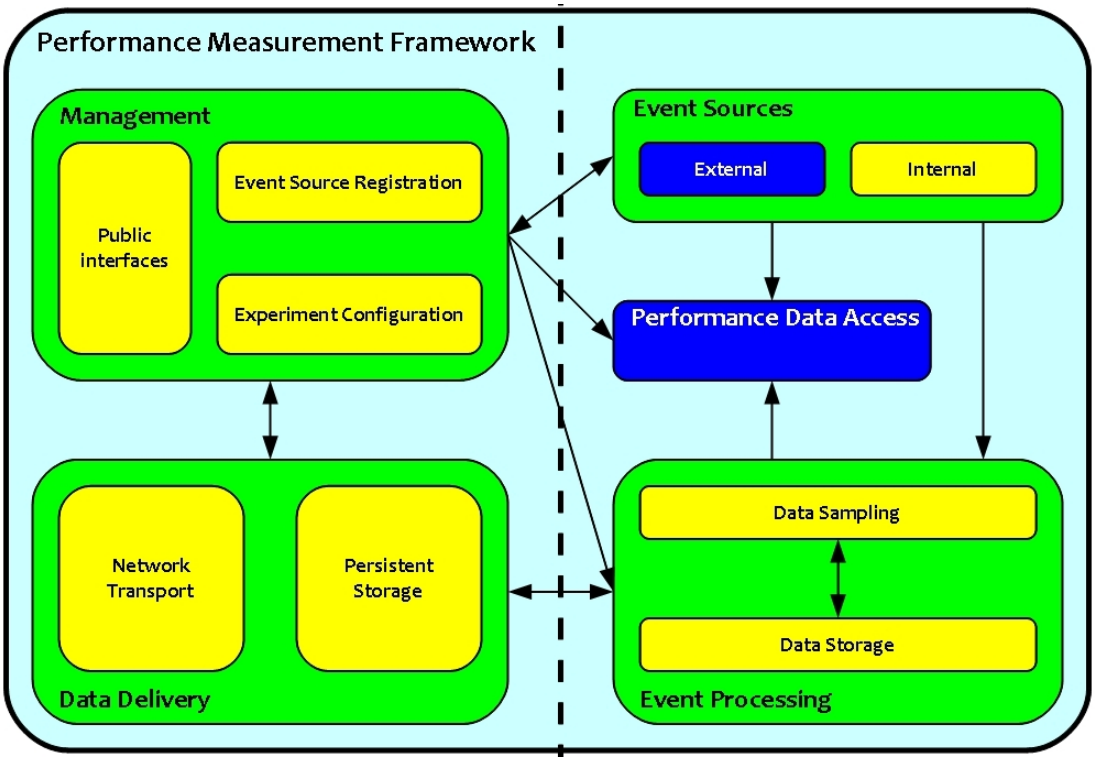


Figure 1.1: JPMF architecture

mance Data (Access), detailed knowledge about the subsystems is not required to understand this thesis. Hence, a short description of the subsystems will be given, followed by more details about Performance Data subsystem.

The runtime entities of the Event Sources subsystem are responsible for emitting performance vents that drive the collection of performance data [1, p48]. A performance event can be understood as a point in the life of system under test where a performance data needs to be collected. Association between performance event and its performance data is maintained by Infrastructure Management subsystem.

The Performance Data subsystem is responsible for providing an unified interface and generic access to different sources of performance data [1, p49]. The performance data typically provides readings of various quantities that can be observed in a running system. Infrastructure Management uses this subsystem together with Event Sources component to create measurement contexts, a selection of performance data associated with particular performance event.

The Performance Data subsystem is independent of other components of the framework. Thus, it may be also used as a standalone component usable in other applications. This may also apply for its individual components called *data sources*, described later in this chapter.

Entities of the Event Processing subsystem are responsible for collecting performance data associated with a particular event and creating a data record for them. This process is called *event processing*. The records is then stored in memory. Performance Data subsystem is used to actually collect the performance data. Creation of the event record lies within responsibilities of the Data Storage

part of the subsystem.

The Data Delivery subsystem is responsible for delivering the collected performance data to the consumers that initiated the benchmark experiment of the system under test. The consumers of the experiment usually do not run in the context of the benchmarked entity (i.e. in the address space of the tested application). hence the Data Delivery is the first subsystem that does not entirely run within the context of the tested system, which holds for the components described above.

Data Delivery subsystem is connected to the Data Storage component of the Event Processing subsystem. Hence, the subsystem can be notified when the memory occupied by event records created during event processing grows beyond preconfigured limits and the data need to be transferred to a persistent storage. There is also a connection with Infrastructure Management subsystem which allows to configure the methods of data transport and other options.

The Infrastructure Management subsystem represents the key part of the measurement infrastructure and of the framework. The subsystem is responsible for configuration and coordination of all other subsystems of the framework. It also maintains an externally accessible management model of the application created from the names of event sources and supported performance events wich allows configuring a measurement experiment from outside [1, p51].

1.2 Performance Data Subsystem

The architecture of the Performance Data subsystem is depicted in Figure 1.2 [1, p55]. Top parts of the component are written in some platform-independent language which is, in case of JPMF, Java. These parts provide level of abstraction over performance data and time information collection methods used in different operating systems and various libraries. In fact, Performance Data subsystem unifies performance data collection (measurement) process, so other parts of JPMF can be written in platform independent manner.

The low level parts of the subsystem are supposed to be developed in platform-dependent languages like C++, they often need to use functions available only on certain operating system, or provided by certain libraries. The task of this code is to collect performance data and time information, and pass them, with usage of appropriate bindings, to the higher level of the Performance Data subsystem.

The most important concepts of the subsystem are*time sources* and *data sources*. Time sources collect time information of various origin. Their internal structure is usually very simple. The purpose of data sources (or data source modules) lies in performance data retrieval. Internal structure of such modules is rather more complex, because they are intended to work with data of various types.

Design of Performance Data subsystem assumes that every single data source module would use certain method of performance data retrieval. As the number of implemented modules will rise, the whole framework becomes more and more platform-independent. Similar statements hold also for time sources.

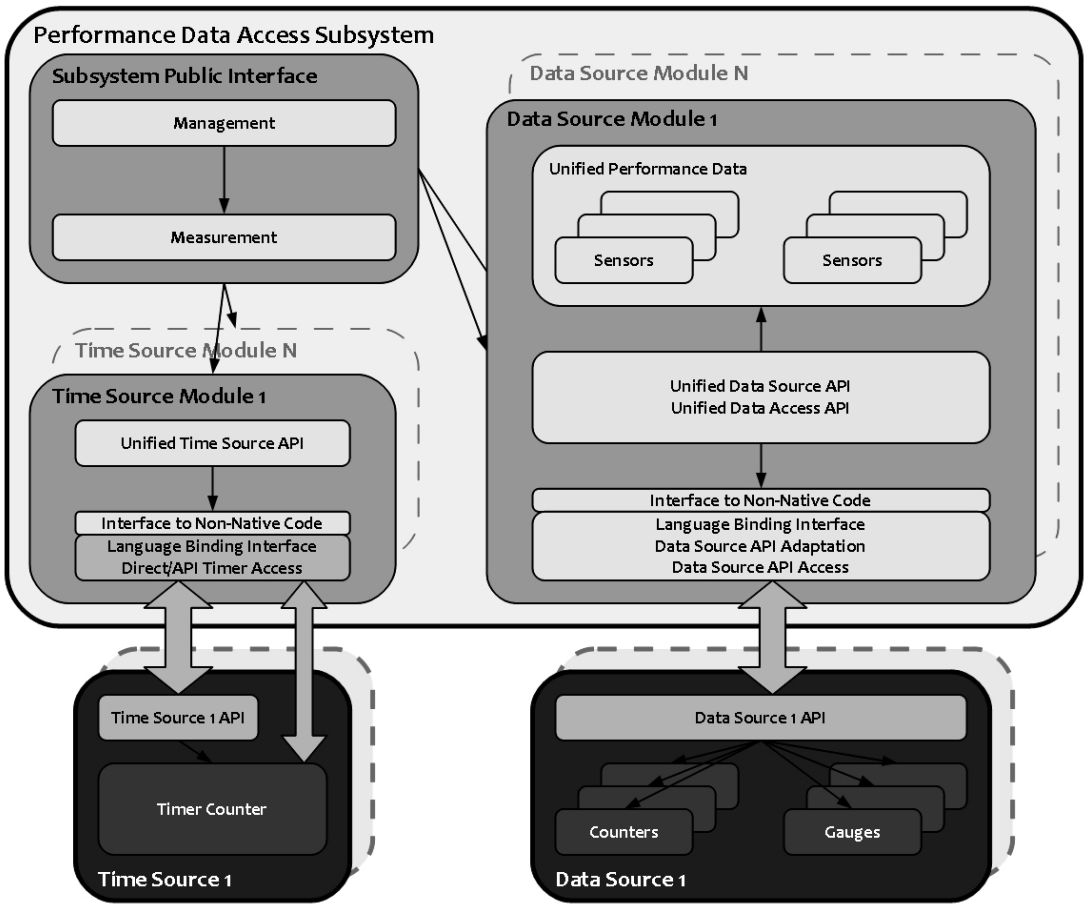


Figure 1.2: Performance Data subsystem structure

1.2.1 Data Source Module

A data source module should allow certain parts of JPMF to examine the structure of the performance data and to to collect them. These modules usually serve as bridges between Java, which, as a managed and platform-independent language, is not able to use operating system specific features directly, and a library, full of code dependent on architecture and operating system, that provides access to the specific sources of performance data.

Because structure of the data provided by a particular source is usually non-trivial, the data source module can be internally divided into pieces that reflect the structure of the performance data. This subsection describes the possible internal structure of a data source from the point of view of JPMF and what services the data source should provide and how.

Internal Structure of the Data Source

The operating system typically consists of many entities. Some of them represent physical hardware, many of them are, however, purely virtual. Structure of performance data typically reflects this diversity. And data source module must reflect it too.

The data source is divided into performance counters and every such entity contains information about one statistics of certain aspect of the system under test. For example, there are counters that report installed amount of RAM, and size of free virtual memory. JPMF represents these single statistics by an abstraction called *a sensor*.

A sensor contains various information about the performance counter, it represents. Such information include sensor name, sensor kind and sensor type. Sensor kind determines the type of the value of the performance statistics represented by the sensor. Sensor type describes the behaviour of the statistics more precisely. This characteristics exists in JPMF and might not appear in the performance data. Its value divides sensors into two classes: *counters* and *gauges*.

Consecutive readings of a single *counter* are required to be monotonous and nondecreasing, with the exception of the counter overflow, which must be taken into account. A typical use of counters is to track number of events or quantities, such as number of bytes transmitted over a network interface [1, p32].

Gauges, on the other hand, provide absolute readings for a given moment and there are no restrictions on the nature of changes between multiple readings other than those implied by the semantics of a particular gauge. Gauges are typically used to convey information about consumable resources, such as the amount of memory available, or system utilization [1, p32].

However, this work does not differentiate between counters and gauges, because it is not important in the context of performance data measurement in Windows. We use the word *counter* to describe an equivalent of sensor in the context of measurement interfaces provided by Windows. So, the definitions given above are not too relevant for this work and we decided to include them only in order to make the introduction into JPMF data source infrastructure more complete.

Sensors might not form the lowest level of granularity. They can be divided into yet simpler entities – *sensor instances* (or simply, *instances*). A sensor

instance represents part of performance counter related to certain entity. This is the case of sensors that describe characteristics of the whole kind of entities - processes, threads, processors etc.

For example, assume that there is a set of sensors that stores statistics related to running processes. This set contains sensor named ThreadCount, the purpose of which is to report the number of threads owned by each running process. Because the set represents a kind of entities, not a single entity, each of its sensors is divided into multiple instances. There is one instance present for each running process e.g. there is an instance representing processes *explorer.exe*, *winlogon.exe* etc. ThreadCount sensor contains a separate value for every instance of the set.

JPMF represents sensors as instances of two classes. Sensors with no instances are represented by instances of `SingleSensor` class. Instances of `StaticSensor` class are used to represent sensors with multiple instances. Sensor instances are part of the `StaticSensor` class. Sensor values (and values for sensor instances) are stored inside instances of classes that implement `ValueHandle` interface. JPMF contains two such classes: `IntValueHandle` to hold 32-bit integers and `LongValueHandle` to store 64-bit integers.

There are different sensor naming schemes across various interfaces provided for performance data measurement. Responsibility of the data source is to convert these schemes into the general format used within the JPMF. JPMF uniquely identifies every single sensor (or its instance) by an URL-like string. The format of the string looks as follows:

```
datasource/group/sensor#instance
datasource/group/sensor
```

The first format is used for sensors that does not split into instances. Such sensors usually reports information about one entity, such as physical memory. Sensors with instances uses the second naming format. Such sensors report statistics for a kind of entities, like running processes and threads.

Measurement

The infrastructure described in the previous section serves mainly two purposes: it allows higher level of JPMF code to examine names and types of sensors available through the data source, and it provides storage for collected performance data. However, the process of measurement is performed by another entity called *a measurement context*.

To initialize the interface for measurement, an application must create an instance of the measurement context. On input, the application specifies sensors which performance data need to be collected. The instance of the measurement context finds data source corresponding to every given sensor.

The measurement context can be understood as an unifying wrapper above data sources. It ensures that an application can collect performance data of certain sensors without any knowledge about data sources to which the sensors belong. The main task of measurement context is to delegate requests of the application to responsible entities from appropriate data sources.

The measurement itself is defined as a three-stage operation. At first, the instance of measurement context instructs all necessary data sources to prepare for raw performance data collection. This stage is quite important because it gives data sources a chance to allocate and prepare all data structures necessary for quick raw data collection, which is done in the second stage. The idea behind this division is to reduce the time of performance data retrieval and minimize unwanted anomalies in the system under test during the collection operation. For example, it is desirable to perform the second stage with as little memory allocations as possible. Every effect that could influence the environment during the data collection stage, should be prevented by the first stage.

The third stage consists of decoding the raw performance data retrieved in the second stage, and putting them into the sensor data structures of the data sources. The raw data itself often do not make any sense. Sometimes, the value of a sensor must be calculated with use of raw data from two snapshots, and with help of time information in various forms. There are no extra restriction placed on this stage, as they were in case of the second one.

Entities responsible for measurement process within the data source are called *probes*. Number of probes defined for a single data source usually depends on the number of methods the data source collects its performance data. Each probe usually implements one of these methods. When looking at the definition of a probe, one might think that this entity should be used to represent a group of sensors of the same origin. This is not true, JPMF does not request data sources to group their sensors in any way, with exception of their unified names. The probes groups the sensors by performance data collection method, however, this usually does not correspond to the aspect of the system which behavior the sensors are reporting. The only purpose of probes is to split up the data source in the context of measurement.

When a measurement context is being created, it determines probes responsible for measurement of the given sensors. The probes respond with creation of *probe contexts*, entities that form a bridge between measurement context and a code that actually performs the measurement. Then, when the measurement context is instructed to perform the three-stage operation described above, it delegates execution of all the stages to its probe contexts which calls appropriate code.

1.3 Goals of This Work

Main goal of this thesis is to implement data source module and integrate it into JPMF infrastructure. The data source module allows the framework to access performance dat counters found in the Windows NT family of operating systems.

Since the Windows Performance Objects interface (in documentations also called as Performance Counters or Performance Data) only supports access with coarse granularity, thus requiring transferring significant amount of data even for a single performance counter, the module should implement measures to avoid issuing unnecessary requests to the interface to minimize the influence of the data gathering process on the monitored applications.

Since the JPMF target platform is Java, the implementation of the performance data source module must also provide appropriate bindings for using the

module from the Java environment. This means that our implementation will be divided into two pieces: Java implementation of necessary JPMF interfaces and classes, and a library (written in C++) that communicates directly with the measurement interface.

2. Analysis

This chapter discusses various methods of performance data collection available on Windows operating systems, and outlines problems these methods come with. The end of the chapter summarizes our findings and specifies goals of the thesis more accurately.

2.1 Performance Metrics in Windows

There are several ways how an application can retrieve various performance statistics about the operating system. This section discusses three of them: native functions, Windows Performance Objects interface nad Performance Data Helper library. Each of these techniques is briefly described together with its advantages and drawbacks.

2.1.1 Native Functions

Native functions (also called *native API functions* or *native system services*) are, mostly undocumented, underlying services (routines) in the operating system that are callable form user mode. These functions represents the lowest level of the operating system code in user mode. Windows API and other interfaces are positioned above this layer.

Native functions are stored in `ntdll.dll`, `user32.dll` and `gdi32.dll` dynamic link libraries. The second and the third library are not of much interest for this work, because they deal with tasks related to graphical user interface and graphics in general. On the other hand, `ntdll.dll` exports many native functions which are intended to work with kernel objects, such as processes, threads and files, and provide basic functionality of the kernel to applications running in user mode.

Although higher level functions, exported for example by `kernel32.dll` or `advapi32.dll`, use routines from `ntdll.dll` quite often, they do not take advantage of all the power of the native functions. This means that applications that use native functions can achieve tasks not solvable by routines at higher level, such as Windows API,.

Some of the native system services can be used to collect various performance statistics about the operating system. Table 2.1 shows several examples of such functions.

Although very powerful, native functions suffer many drawbacks which make them unusable for most of common tasks. Their are, in most cases, officially undocumented and developers are advised not to use even the documented ones whenever possible. Native functions are not guaranteed to be compatible across various Windows versions. There may be subtle changes in their behavior, or they may disappear completely.

Programming with native functions is also quite uncomfortable because they are barely more than wrappers around system call mechanism. This, and the other reasons discussed above, makes native functions unusable for general performance data collections.

Table 2.1: Examples of native system services useful in collecting performance data

Native function	Description
`NtQuerySystemInformation`	Allows to collect various statistics about, for example, the operating system, memory usage, processor, processes and threads.
`NtQueryInformationProcess`	Can be used to get more detailed information about processes.
`NtQueryInformationThread`	Intended to retrieve various information about threads.

2.1.2 Windows Performance Objects

Windows Performance Objects interface allows applications to collect performance data from various sources in one single step. The interface does not operate at granularity of single sensors, that are, in the scope of the interface, called *performance counters*. Hence it does not fit applications that need to know values of only several counters. Such applications usually take advantage of Performance Data Helper library (PDH) described in subsection 2.1.3.

The interface retrieves performance data in snapshots that are sometimes called *data blocks*. The snapshot consists of *performance objects* which purpose is to group counters originating from the same area of the system under test. For example, there are performance objects that contain performance counters related to processor performance, thorough system performance or memory usage statistics.

Some performance objects are divided into *instances*. Every counter that belongs to such a performance object contains separate values for every instance. For instance, there is a performance object that provides various information about currently running processes. The object is divided into number of instances – there is exactly one instance for every process, running in the system. The performance object contains a counter named **ID Process**. This counter stores value of PID (process identifier) of every running process in its instances.

As mentioned above, the Windows Performance Objects interface works only with the whole performance objects or bigger entities, not the single counters. When an application wants to collect performance data, it cannot instruct the interface to collect data only from necessary counters. It must specify performance objects responsible for that counters and extract the information from the snapshot of whole performance objects.

Windows Performance Objects is not a typical interface; it is not a set of routines and variables, exported to provide certain functionality. An application uses the interface when accessing `HKEY_PERFORMANCE_DATA` registry hive. It is not a real hive (it is neither stored in a file on disk, nor present in registry structure in memory) and although the application must use registry functions to read its contents, they are used in a quite different manner. than usual.

`HKEY_PERFORMANCE_DATA` can be treated as a registry hive with root key only. Unlike the normal hives, its root key contains values. However, they are not real values and cannot be enumerated, just queried.

When an application needs to retrieve a performance data, it attempts to read

Table 2.2: Value name formats for HKEY_PERFORMANCE_DATA pseudo hive

Value	Description
Global	Retrieves performance data for all performance objects registered on the computer, except for those included in the Costly category.
nnn xxx yyy	Collects data for the specified performance objects. Specify the index value associated with the objects that you want to retrieve. For example, if we want to retrieve data of System and Memory performance objects, we specify "2 4". The query can return more objects than requested if the specified performance object depends on another one. For example, threads depend on processes, so when data for the Thread object is requested,the query also returns data for the Process object.
Costly	Retrieves performance data for performance objects whose data is expensive to collect in terms of processor time or memory usage. Collection of performance data of this type should be performed in a worker thread to keep application responsive during the measurement process.

a value from this pseudo hive through RegQueryValueEx function. The name of the value indicates which performance objects is the application interested in. Possible formats of the value name are shown and described in Table 2.2 [9].

As the table shows, performance objects are referenced by numeric values rather than by their names. This way of identification is used also in other data structures related to performance data (for example, structures describing characteristics of single counters). Thorough this text, such identifiers are referred to as, *performance object indices*, *counter indices*, *name indices*, or just *indices*.

This identification of performance objects and counters allows the whole data measurement interface to support multiple languages, with exception of counter instances which names are not identified by indices. The index-to-name mapping is stored in registry under the key

```
HKLM\SOFTWARE\Microsoft\Windows NT \CurrentVersion\PerfLib\LID
```

where LID is a number representing a particular language. The key contains two pseudo values named Counter and Help. The former contains registry mapping for performance object and counter names, the latter stores mapping for counter and performance object descriptions. The data of these values have the form of string array (sequence of Unicode strings separated by null character) which elements are pairs. The first member of every pair represents index number and the second one the string which the index translates to.

The application might also read index-to-string mapping by reading value data of several pseudo registry keys listed in Table 2.3 [10] together with description of their effect.

Table 2.3: Pseudo registry keys available for index-to-string mapping retrieval

Key	Description
HKEY_PERFORMANCE_DATA	Queries strings based on the language identifier value specified in the value name. Value name needs to be set to either `Counter langid` or `Help langid` to retrieve the names or description (also called *help text*), respectively. The language identifier isl optional. If omitted, English strings are retrieved.
HKEY_PERFORMANCE_NLSTEXT	Queries strings based on the default UI language of the current user. The value name needs to be set to either `Counter`, or `Help` to retrieve the names or help text, respectively.
HKEY_PERFORMANCE_TEXT	Queries English strings. The value name needs to be set to either `Counter`, or `Help` to retrieve the names or help text, respectively.

Performance Data Structure

As stated above, an application retrieves performance data by reading pseudo value of HKEY_PERFORMANCE_DATA pseudo registry hive. The basic format of the data snapshot is depicted in Figure 2.1 [8]. The snapshot begins with PERF_DATA_BLOCK structure that contains general information about the measurement, such as number of performance objects and time information in various formats.

The rest of the snapshot is filled with sequence of variable length PERF_OBJECT-_TYPE structures, with each of them containing information about one performance object, its counters and their performance data. The index of the name of the performance object is stored in ObjectNameTitleIndex member and the field ObjectHelpTitleIndex contains index for the description string of the object.

There are two possible layouts for the performance object. They are shown in Figures 2.2 [8] and 2.3 [8]. In case of a performance object with no instances, the header describing the object itself is followed by a sequence of PERF_COUNTER_DEFINITION structures. Each of them describes one counter and contains index of the name and description, type and location and size of related performance data. The performance data itself are not present within the counter definition but in *counter block*, represented by a record of the PERF_COUNTER_BLOCK type which follows the last counter definition structure.

When the performance object consists of one or more instances, the sequence of counter definitions is followed by structures describing individual instances and counter performance data for them. Every instance is described by PERF-_INSTANCE_DEFINITION structure, followed by an instance name in form of Unicode null-terminated string. After the name, there is a counter block, containing performance data of all counters, that belongs to the performance object, for a particular instance. This implies that there is one counter block per performance object instance.

Performance data stored in the data block are not ready to be displayed.

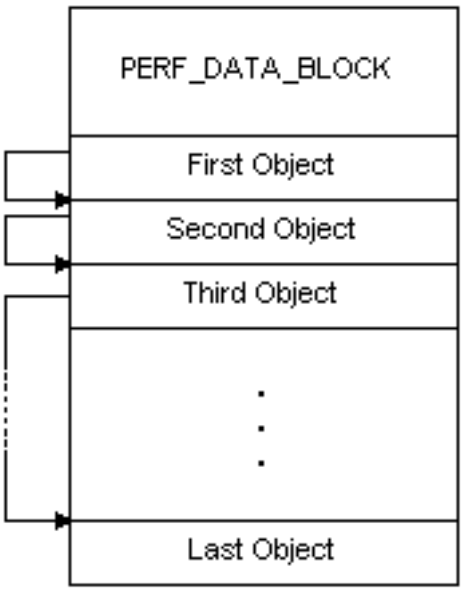


Figure 2.1: Performance data format

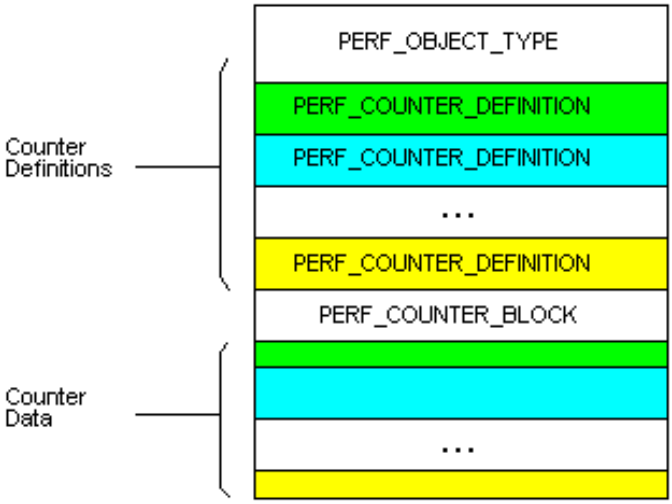


Figure 2.2: Data format of instanceless performance objects

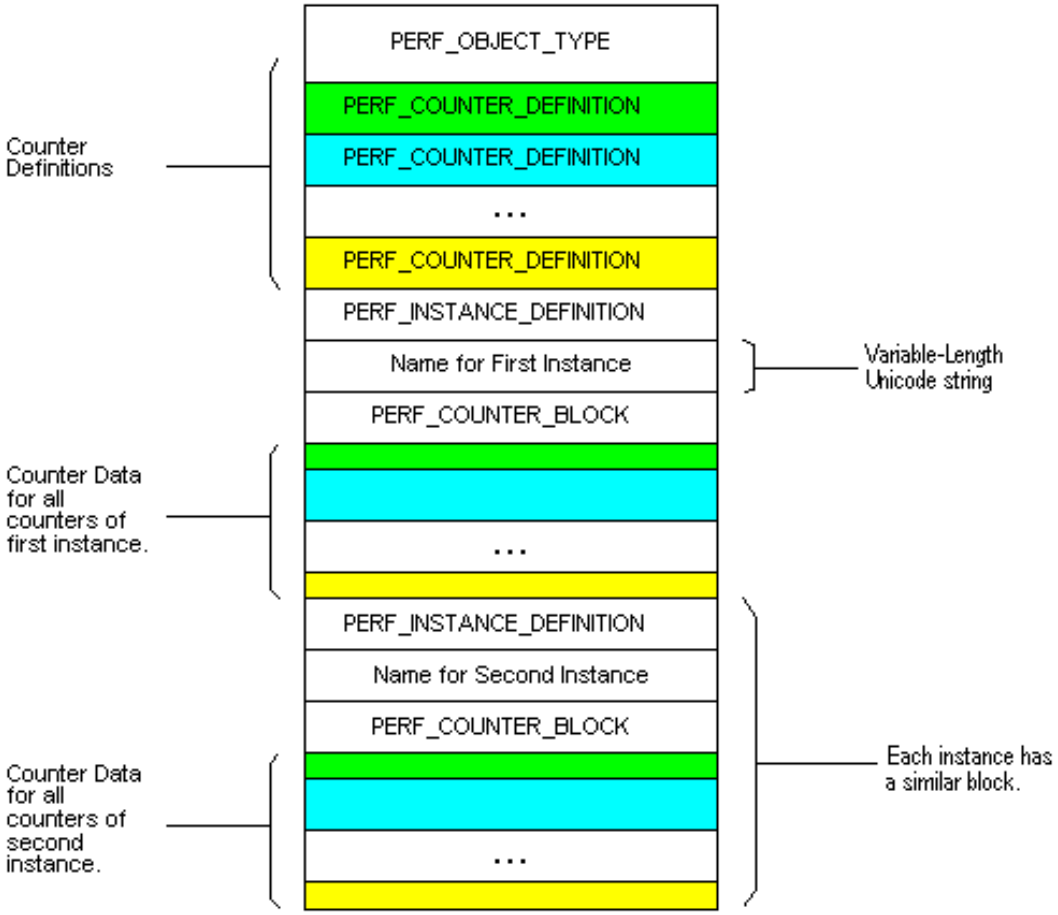


Figure 2.3: Data format of performance objects with instances

The real values of the counters must be calculated. The particular calculation formula depends on the type of the counter and may take more than just the raw performance data as its inputs. Some types of counters require to use time information from the performance object or from the data block to calculate the values. And some types of counters require to perform more snapshots of the data.

2.1.3 Performance Data Helper (PDH)

Performance Data Helper (PDH) interface is provided by `pdh.dll` library and is actually built on the top of the registry interface described in Section 2.1.2. An interesting feature of the interface is the ability to save collected performance data to log files, which can be read later. The interface provides an unified way of collecting performance data – an application can use the same code to either collect performance data in realtime, or read them from the log files. Basically, the PDH library supports two kinds of operations: enumeration of counters and instances, and performance data retrieval.

The library provides two functions to enumerate available performance objects and counters. `PdhEnumObjects` allows an application to find out which performance objects are available. `PdhEnumObjectItems` routine retrieves information about counters and instances of given performance object. Both functions require to specify the source of the performance data to examine. It is possible to query realtime performance data, or a single log file, or even a set of the log files. It is also possible to query these sources on a remote computer, specified by its name or IP address. When an application needs to use PDH for performance data collection, it should follow these steps:

- **Create a PDH query**. This is a task for `PdhOpenQuery` routine. The application must specify the source, from which it wants to collect performance data. The routine returns handle of the new query., wich is required in subsequent PDH calls.
- **Add counters to the query**. `PdhAddCounter` serves this request. An application can specify a computer, where performance data of every counter must be retrieved.
- **Collect raw data for the counters**. The application uses `PdhCollectQueryData` routine to achieve this task. Because many counters need at least two samples of their raw data, the application should call this routine twice and put a short time interval between the invocations. For counters that need it, PDH library keeps their raw data obtained in the last two samples.
- The application uses `PdhGetFormattedValue` to compute the value of certain counter from its raw data, collected in the third step. The routine must be called once for every counter, the value of which the application wants to calculate. The application must specify a handle of the counter as a parameter of the function call. The handle is returned by the call to the `PdhAddCounter` function.

- When the application no longer needs to collect performance data, it should call `PdhCloseQuery` to release resources allocated for the query.

In case the application wants to write the performance data to a log file, rather than calculate counter values, it should not perform the third and the fourth step. Instead, it should invoke `PdhOpenLog` to associate a log file with the query. Then, it should use `PdhUpdateLog` routine, which collects performance data and writes them to the log file. Every single call to the function means one data sample written to the log file. The log file should be closed by a call to the `PdhCloseLog` routine. The application should also perform the fifth step of the algorithm described above.

The disadvantage of the PDH interface is that its design is counter-oriented. This means that it perfectly fits applications that need to compute values for several counters only. As mentioned above, an application must select individual counters the values of which are then being calculated. This approach is quite slow when taking snapshots of many counters, or especially when there are more entities retrieving performance data in parallel.

2.2 Summary

We conclude that using Windows Performance Objects interface for implementation of a data source is better for the goals of JPMF than using other discussed ways of collecting performance data. We think that the interface should provide acceptable performance when collecting large amounts of performance data and, with certain optimizations, it could work well enough in parallel environment.

Performance Data Helper Library present a measurement interface with interesting options, such as single counter access and retrieving performance data from log files. However, these options are considered as not very useful for current implementation of JPMF. Ability to retrieve larger quantities of performance data in reasonable time is more important in this case. And PDH lacks this ability.

Native functions, discussed in Subsection 2.1.1, provide fast interface to various kinds of performance data, however, they are mostly undocumented and their abilities depend on the installed version of the operating system. Additionally, compatibility with new versions of Windows is not guaranteed. Similarly to PDH, this method of performance data collection might represent a good option for applications and utilities that only need to measure values of several counters. However, it cannot be a method of choice for universal performance monitoring system like JPMF.

However, it is not wise to consider Windows Performance Objects in its raw form as the ideal method of performance data collection for JPMF. The interface employs a lot of memory copying which is not always desirable. This means that a mechanism that reduces number of queries to the interface should be presented. We think that caching of recent performance data snapshots will solve the problem. Another problem lies in the way the interface names counter instances. In several cases, instance names are not unique within the scope of their counter. This is the case of Process performance object that reports information about running processes. The data source modules should provide solution to these rare cases.

Additionally, Windows Performance Objects interface cannot be accessed directly from Java programming language, in which JPMF is written. The solution of this problem lies in the library written in a native language that is capable of registry access. The library should build some abstractions over the interface and should provide services of the interface to JPMF. The library might be written in C++ language. Java Native Interface (JNI) should be powerful enough to establish a connection between the framework and the library.

Also, our abstraction over Windows Performance Object interface should be implemented thread-safe. Although JPMF probably won't benefit from such property too often, a thread safe implementation might show very useful in future, particularly in case, the library would be used as a backend to interface than JPMF. This request implies usage of a primitive to synchronize access over the Windows Performance Object abstraction. Such a primitive should allow multiple threads to read performance data, however, it must ensure that only one entity can measure them (obtain fresh data through the registry interface) at a time. Reader-writer lock suits this needs probably the best. Because such a primitive is not provided by Windows API until Windows Vista, which introduces Slim Reader Writer Locks, it is required to implement its replacement for earlier versions of Windows, that are still officially supported.

Although the structure of raw performance data retrieved by Windows Performance Objects might suggest it would be wise to represent every performance object by a probe, we decided not to follow this way. We think that the big advantage of the interface is a possibility to retrieve raw data for more performance objects in one step. Multiple-probe approach would cancel this advantage, although this effect might be reduced by certain optimizations. Additionally, probes are not designed to represent groups of counters (or performance objects). Their purpose is to build a bridge between measurement contexts and code for performance data retrieval. The multiple-probe approach should be used only in cases a method of performance data collection depends on the origin of the counters. In such a case, the data source should use multiple probes, each implementing one method of data collection.

2.2.1 Goals Revisited

Text in the previous sections of this chapter gives more specific goals to achieve in this thesis. They are the following:

- Implementation of an abstraction over Windows Performance Objects interface that allows JPMF to collect performance data from various areas of Windows operating system. The abstraction should use the registry approach discussed in Subsection 2.1.2, extensions such as PDH have been considered as inadequate with respect to the architecture of JPMF..
- The abstraction needs to be written in a native language, e.g. C++, and connected to JPMF through carefully designed binding. Java Native Interface (JNI) should be strong enough to build the bridge.
- The abstraction should cope with two main problems of the interface chosen to measure performance data. The first problem lies in sometimes uncomfortable naming of counter instances. The solution is to use different names

for such instances. The second problem is the quite low granularity of Windows Performance Objects interface. This problem should be solved by implementing caching of performance objects, recently obtained through the registry interface.

- Access to the individual entities of the abstraction needs to be synchronized. Prior Windows Vista, there is no suitable synchronization primitive exported to usermode applications. It is needed to implement a reader-writer lock in case the library would be used on Windows XP or Windows Server 2003.

3. Design

Implementation of our data source and its integration into JPMF brings several obstacles. An abstraction must be built on top of the Windows Performance Objects interface, in order to get it into shape more suitable for the concepts used within JPMF.

The second problem lies in sensor identification. The data source must provide names of all its sensors, names that match the format used by JPMF. The third problem is caused by inability of Windows Performance Objects interface to effectively collect small amounts of performance data. The problem might be reduced by implementation of performance object caching mechanism.

Problem of connection between the library and JPMF should also be solved. Because the framework is written in Java and the library uses C++, JNI can be used to put them together. It is needed to think up a reasonable interface to allow both parts to interact with each other.

All these problems were introduced in Section 2.2. The main goal of this chapter is to design architecture of our data source and to suggest a solutions to these problems.

3.1 Overview

The schema of the architecture of our data source module is depicted in Figure 3.1. It is influenced mostly by the characteristics of the Windows Performance Objects interface, described in Subsection 2.1.2.

Because the Windows Performance Objects interface cannot be invoked directly from JPMF code, and because its design is quite different from that of the JPMF, we decided to develop a native library that enables Java code to actually access the data source, and adapts the registry interface to JPMF concepts. The functionality of the library is exported to JPMF world through native methods of Java classes.

The library implements two kinds of entities for usage in JPMF (or other frameworks): WPOSnapshot objects and WPONames objects. In this section, we take these entities only as concepts. We are interested in what services the they should provide, not in implementation details like what classes are used to represent the entities. The latter has no counterpart among JPMF concepts. Its purpose is to translate integers used to identify performance objects and counters within data structures of the Windows Performance Objects interface into their human readable string form. The former entity, WPOSnapshot object, corresponds more or less to the concept of a probe context found in the JPMF. Its main task is to measure performance data, specified during construction of the of the object. During initialization of the whole data source module, an instance of the WPOSnapshot object is created and instructed to collect all available kinds of performance data. This allows the module to create all necessary `StaticSensor` and `SingleSensor` objects in order to represent individual counters and their instances.

Apart from performance data collection, WPOSnapshot object allows inspection of the structure of the snapshot, i.e. it can be used to enumerate performance

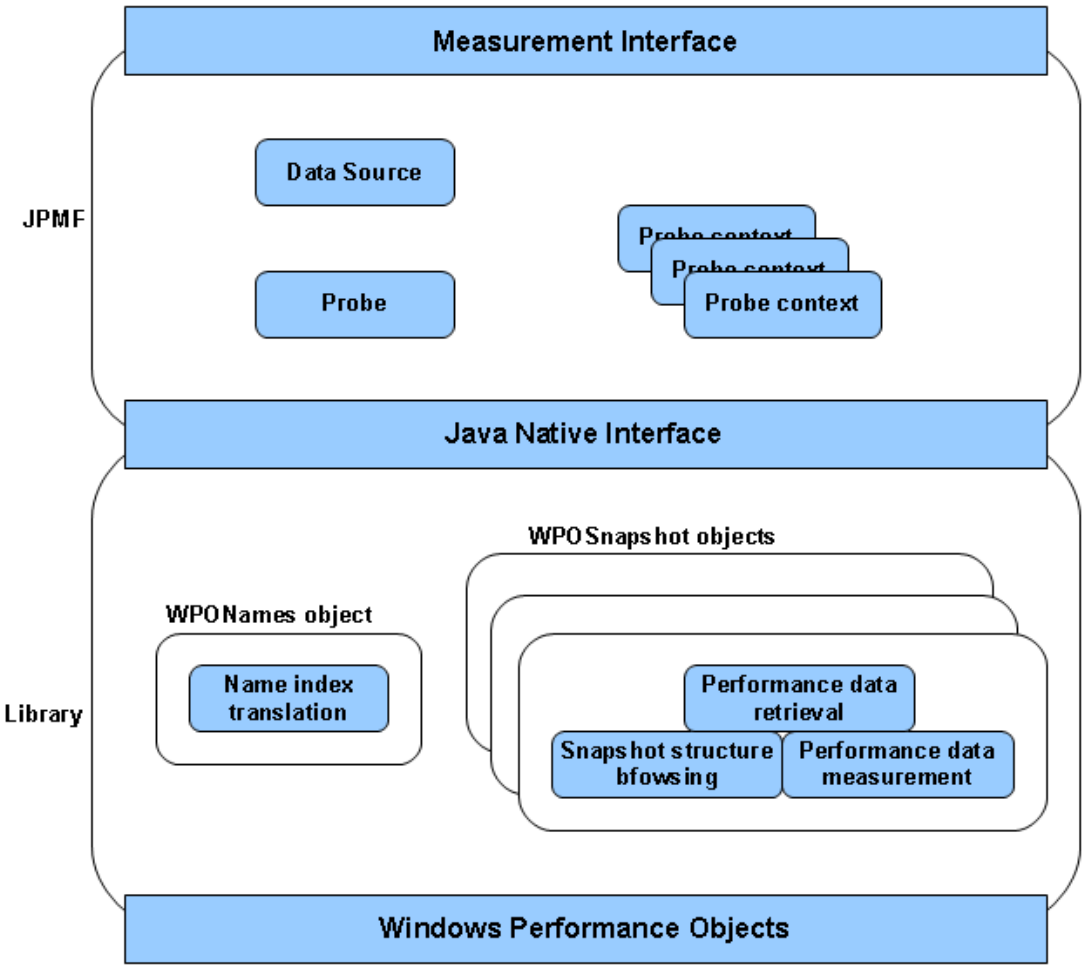


Figure 3.1: Architecture overview

objects, counters and instances present in the current snapshot.

We assume to use our data source module for measurement of static counters only – this means that the module may not function properly in case counters are added, modified or deleted. There is a way how one can add or delete counters in the Windows Performance Objects interface, but because this may happen only during installation or uninstallation of certain application, we decided not to consider this possibility. Installation of that kind of applications is a quite rare process and performance measurement should be stopped until it is finished.

Changes in counter instances are more considerable problem, because they may happen very often. For instance, when a new process is created, a new instance is added to Process performance object. When a process exits, its instance is removed. The same applies to threads and dynamic link libraries mapped to address spaces of processes.

So, everything depends on which counters and instances our data source module is requested to monitor. Changes to counter instances should be detected during the measurement process and reported as an anomaly.

There exists only one instance of WPONames entity because index-to-string mapping of names does change only in cases of counter addition, deletion or modification, and as stated above, these events are so infrequent to adapt the data source to them. Names of counter instances are not covered in this mapping.

However, there should exist more instances of the WPOSnapshot entity, each representing one instance of probe context within the library. We do not count instances created during initialization of our the data source, in order to build the structure of sensors and their instances, because they are destroyed shortly after they fulfill their purpose.

3.2 Sensor and Instance Naming

This section describes how WPOSnapshot objects and other parts of our registry interface abstraction handles names of the performance counters and instances. It also describes how global names for JPMF sensors are built from names of performance objects, counters and instances used in the registry interface.

3.2.1 Building unique names for sensors

JPMF defines special format to uniquely identify every sensor and instance within the whole framework. Our data source adheres this format in quite straight forward way. For its sensors and instances, it builds the names in one of the following formats:

```
windows.winperf/group/sensor
windows.winperf/group/sensor#instance
```

The first format is used for sensors that have no instances, the second is used for the latter case. The data source sets the first part of the names to `win32.winperf` to inform the framework to identify the data source, operating with the sensors. The `group` field contains name of performance object the counter, represented

by the sensor, belongs to. The last two parts contain name of the counter and optionally name of its instance.

As mentioned in Subsection 2.1.2, describing Windows Performance Objects interface, the interface stores names of performance objects and counters in form of integer numbers, that might be translated to strings when necessary. The situation is different with names of counter instances. Their string representation is stored inside of their performance object structure.

The problem is that name of an instance might not be unique within the scope of its performance object. For example, performance object named Process is divided into instances whose names match image names of running processes, they represent. Because there may be multiple processes with the same image name running at the same time (i.e. *svchost.exe*), names of some instances might be identical. Although this case is very rare, it must be taken into account.

In its definition structure, every instance contains an interesting field called *parent instance index*. This field is used in some cases and contains order number of the "parent" instance in the "parent" performance object. For example, performance object named Process is parent object for the Thread performance object, and hence every instance of the Thread object contains an index of the parent instance, representing its parent process, in the Process object. So, the values of the parent instance index field do not provide way to uniquely identify instances within their performance object, although they usually makes the structure of the object tidier.

Every instance may also stores its *unique identifier*in its definition structure. The unique identifier of the instance is valid only in case, the provider of the performance object decides not to use instance names for the identification. Unfortunately, Process and Thread performance objects do not use the unique identifier, although we think that setting these identifiers to PIDs and TIDs of the entities would provide a nice and easy way of instance identification. However, the identifiers are not used.

The instances might be uniquely identified by their order numbers in the performance object structure. The ordering is preserved across snapshots of the performance data. However, the order number may change within change to structure of the performance object, which happens when an existing instance is deleted. Additionally, it is not natural to identify instances of some performance counters by such kind of numbering. For example, counters related to processes and threads should be identified by PID or TID of the corresponding thread or process. Hence it is necessary to map these names to instance order numbers and attempt to preserve this mapping across changes of the instance order.

Names of performance objects and counters

Windows Performance Objects interface stores names of performance objects and counters in form of indices. Their mapping to names is stored in registry key, as described in Subsection 2.1.2.

This form of name storage is very convenient because the indices are unsigned 32-bit integers only, so there is less data copying between the library and JPMF. The indices need to be translated into string form only when a new sensor is being created, or when the sensor name in JPMF format needs to be converted into corresponding sensor.

Because the translations of name indices are required mainly in JPMF code, the library provides an abstraction called *WPONames object* that stores the mapping in its internal structure. Because the goal of this work is not to deal with situations when the mapping changes during one run of JPMF (which happens when a performance object is added or deleted), there is only one WPONames instance, created during the initialization stage of the library. During its creation, the instance reads index-to-string mapping from the registry pseudo values and stores it inside itself.

The WPONames abstraction provides the following operations:

- **Index translation**. On input, an entity specifies the index and the abstraction retrieves corresponding string value.
- **Mapping retrieval**. The WPONames abstraction provides all information about the mapping to the entity (from the scope of JPMF) that requested the operation. Because of the assumption that the mapping does not change, the entity is able to fully replace the WPONames abstraction in terms of its functionality. At first, it request all mapping information and then it serves as a cache to the rest of the data source, so no additional calls through JNI are needed.
- **Counter name translation**. On input, a caller specifies a performance counter by its string name and name index of its performance object. The WPONames abstraction returns name index corresponding to the string name of the counter.

At first glance, the second operation might not seem very useful, however, it can save transitions between interpreted and native code and thus avoids data copying between the library and JPMF (or another component that uses the library).

In some situations, it could be useful to provide also a way of translating a string value to the corresponding name index. However, there are multiple indices that translates to the same name, although stored in the different place of the mapping in the registry pseudo values. The inverse mapping (translation of string values to name indices) is a function only in case of performance object names, and in case of counter names originating from the same performance object. In other cases, it is a binary relation. This means the caller must specify name index of performance object on input in order to be able to perform the "inverse" translation.

For example, there are multiple counters named % Idle Time. The counter exists in performance objects named Processor and PhysicalDisk. The former reports statistics related to performance of processors, the latter informs about various statistics of installed physical drives. The name index of the counter in Processor performance object is 1746. The counter in PhysicalDisk performance object has name index of 1482. This proves that there are counters with different name indices that translate to the same string name.

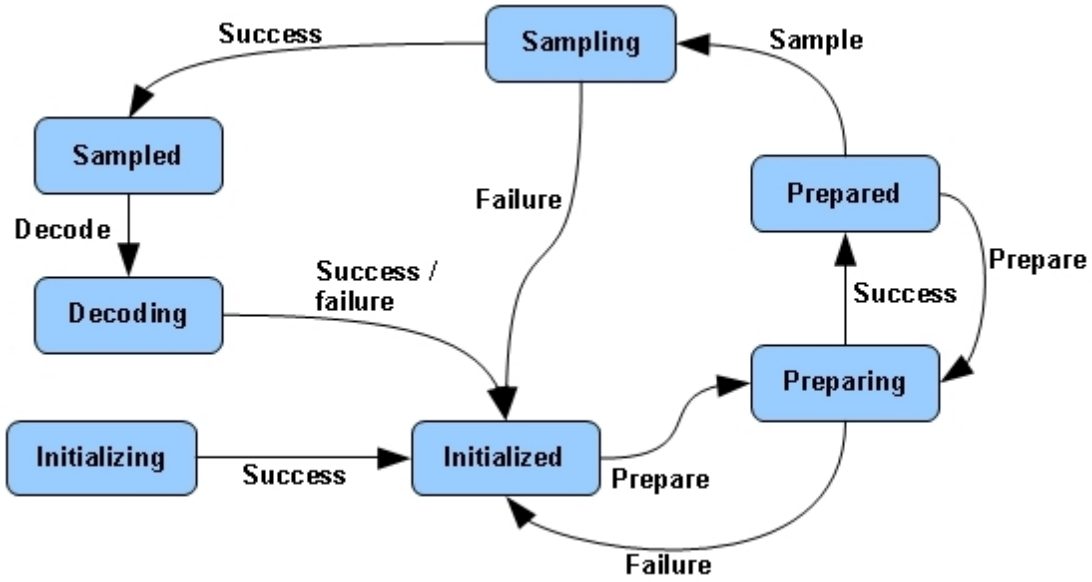


Figure 3.2: State diagram for WPOSnapshot objects

3.3 WPOSnapshot Objects

WPOSnapshot object is an abstraction over Windows Performance Objects interface. It makes working with this interface more convenient by hiding certain details that application developers working with the registry interface must bear in mind. WPOSnapshot objects hide the fact that the registry interface typically retrieves more data than the caller desires. They also calculate displayable values from collected performance data. Additionally, WPOSnapshot objects cooperate with the performance object caching mechanism (described later in Section 3.4) to minimize interactions with the registry interface in order to reduce overhead. WPOSnapshot objects are counterparts of probe contexts in JPMF.

Because of the relation with probe context entities, WPOSnapshot objects must support three operations which, put together, form one measurement cycle: *prepare*, *sample* and *decode*. The object should allow to execute these operations only in correct order.

To detect execution of these operations in incorrect order, we provided the objects with internal state. The WPOSnapshot object must always be in in one of the states depicted in the state-chart diagram in Figure 3.3. The diagram also indicates which sequences of state transitions are acceptable.

According to the diagram, a WPOSnapshot object may be in one of the following states:

- **Initializing**. The object enters this state immediately upon its creation and stays there until its whole internal structure is prepared to operate. Then it enters the **Initialized** state.

- **Initialized**. The object is ready to provide information about structure of performance data snapshot it represents. If at least one measurement cycle has already been performed, the object also allows to retrieve calculated

values for individual performance counters and instances. The object can be instructed to start new measurement cycle by entering the **Preparing** state.

- **Preparing**. The object is just performing the *prepare* operation. It does provide neither information about the structure of the snapshot, nor calculated values for individual counters and instances. Success of *prepare* operation is signaled by entering **Prepared** state. Otherwise, the object returns to **Initialized** state.

- **Prepared**. The object successfully performed the first operation in the measurement cycle. It can be instructed either to start performance data sampling, or to perform *prepare* operation again.

- **Sampling**. The object is just in the middle of the *sample* operation. It does provide neither information about the structure of the snapshot, nor calculated values for individual counters and instances. Success of performance data sampling is indicated by entering **Sampled** state. Otherwise, the object returns to **Initialized** state.

- **Sampled**. The object has successfully performed the *sample* operation. It can be instructed to attempt to calculate values for collected performance data. The object does provide neither information about the structure of the snapshot, nor calculated values for individual counters and instances.

- **Decoding**. The object is just calculating values for performance counter and instances, which raw data were sampled earlier. After the operation is finished, the object always enters **Initialized** state. Success of the operation is indicated by an error code.

3.4 Performance Counter Caching

Windows Performance Objects interface is not intended to be used for collecting data of single performance counters. The smallest unit of granularity is the performance object. When gathering performance data using this interface, an entity cannot obtain values for a single counter but only for the whole performance object and all its instances. Additionally, for some performance objects, it is not possible to obtain performance data for themselves only. For example, when an entity attempts to get performance data for the Process object, which aggregates counters related to running processes, it also receives performance data for the Thread object, the counters of which contain information related to threads. This means that an entity might receive more performance data that it actually asked for.

It may happen that two entities (WPOSnapshot objects) attempt to get performance data almost at the same time and one of an them receives data for performance objects it did not ask for. However, these extra data might be useful. If they were retained for a while, another entity could get them, thus avoiding the potentially expensive interaction with the Windows Performance Objects interface. This is one reason to implement caching of recently obtained performance data.

Such a cache can also be useful when two or more entities attempt to get data for the same set of performance objects. If these requests occur within a sufficiently short time interval, the performance data can be sampled only once. Other entities would get them from the cache.

However, if an entity asks for performance data from the same performance object twice (or more times) in a short time interval, the cache should not satisfy these requests. it may satisfy the first request, however, subsequent requests should be forwarded to the Windows Performance Objects interface to obtain a fresh copy. As mentioned earlier, some counters need at least two measurements to calculate their values. If both these measurements were provided by the cache, the measured data would be identical in both cases, which could lead to inaccurate or even completely wrong results.

Besides the performance data itself, the cache must also store time information in order to properly calculate values of certain kinds of counters. Without keeping this information for every cached sample of performance data, it is impossible to compute the displayable values accurately enough. To distinguish between old and fresh data, a timestamp is associated with every entry in the cache.

3.4.1 Operations

External entities can perform several kinds of operations on the cache. This subsection gives general overview of these operations only. They are covered in great detail in chapter 4. The operations are the following:

- **Insert**. Adds new content to the cache. This operation is usually performed when a fresh performance data are obtained through the registry interface.
- **Find**. Attempts to find data for given performance object in the cache.
- **Release**. When an entity no longer needs the cached data, it should tell this to the cache by performing the Release operation. This operation, together with the Find operation, allows the cache to keep track of entities that uses certain cached performance data.
- **Lock**. Locks given performance data in the cache. The lock guarantees that the cache will not manipulate with the data until the lock is released.
- **Unlock**. This operation is the opposite to the Lock operation. It releases the lock that prevents the cache from manipulating with certain performance data.

3.4.2 Components

The whole performance object cache mechanism consists of three parts: Cache Table, Outdated List and Banned Entity List. Together, these parts guarantee characteristics described above.

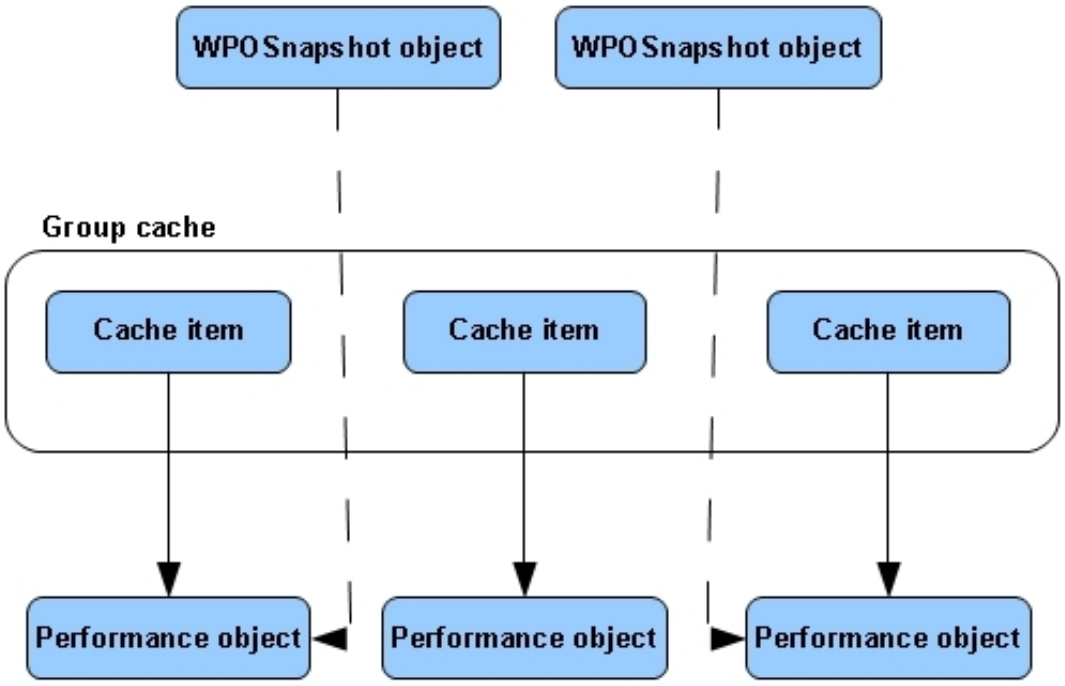


Figure 3.3: Invisibility of cache items outside the scope of the group cache

Cache Table and Cache Items

The Cache Table serves the main purpose of the whole performance object cache. It allows entities (WPOSnapshot objects) to quickly retrieve performance data obtained earlier through Windows Performance Objects interface. The caching is performed at the level of performance objects, not single counters. This suits the nature of the registry interface much better than caching all single counters separately which requires additional processing of fresh data before they can be put into the cache.

Each cached snapshot of a performance object is represented by an abstraction called *a cache item*. Cache items are invisible outside the cache. The cache provides external entities with direct pointers to the cached performance data, and counts number of these references. The external entities give these references back by performing the Release operation.

This behavior is illustrated in Figure 3.3

The Outdated List

Outdated List is a data structure that gathers cache items which are connected to performance data too old to reside in the Cache Table. Unlike the table, the Outdated List might include more cache items representing different snapshot of performance data for the same performance object. Particular cache item stays in the list as long as there are references to it held by entities outside the cache (the reference count of the item is greater than 1).

The Banned Entity List

The purpose of existence of this component is to prevent cache usage when it is undesirable. Such scenarios usually happen when an entity (Wposnapshot object) attempts to request cached data of the same performance object twice in a short period of time. The reasons why they are undesirable were discussed in the beginning of Section 3.4.

The component consists of pairs $(pi; eid)$ where pi is name index of a performance object and eid is an unique identifier of an entity (WPOSnapshot object). The cache uses addresses of `CWPOSnapshot` classes, that represent WPOSnapshot objects in the scope of the library, as an unique identifier of an entity. Every such a pair instructs the cache to prevent entity with identifier eid from accessing cached performance data of performance object with name index pi.

4. Implementation

This chapter describes technical aspects of the design presented in Chapter **??**. Solutions to other problems presented in Section 2.2, not addressed in the design chapter, are also discussed here.

4.1 Overview

Figure 4.1 shows the main classes that make up the data source, and the relations between the classes. Compared to Figure 3.1, it shows the classes used to implement WPOSnapshot objects and WPONames entity, the concepts introduced in Section 3.1.

WPONames entity is represented by `CWPONames` class in the scope of the library. `JWPONames` class can be understood as an entry into the library space that allows to control `CWPONames` class from JPMF world. Both classes are connected together through Java Native Interface (JNI). As we have stated in the beginning of Section 3.1, there is only one instance of WPONames entity which implies exactly one instance of both classes.

Similarly, WPOSnapshot entities are represented by instances of `CWPOSnapshot` and `JWPOSnapshot` classes; the former is visible only in the scope of the library and implements the whole functionality of the entity, the latter allows JPMF to access methods of the former class, again through JNI. For every instance of probe context entity, one instance of `CWPOSnapshot` and `JWPOSnapshot` is created.

4.2 WPOSnapshot Objects

4.2.1 Object Structure

Each WPOSnapshot object is represented by `CWPOSnapshot` class the structure of which is shown in Figures 4.2 and 4.3. This section describes individual members of the structure and how they are used during object operations.

The fields `DataBlock`, `DataBlockSize`, `RequestededPerfObjectsStr` and `RequestedPerfObjectsStrLength` are directly related to the underlying registry interface. `DataBlock` stores address of `PERF_DATA_BLOCK` structure of the latest snapshot of performance data. Size of the buffer for the data block is stored in `DataBlockSize` field and may be larger than actual size of the data block. The buffer to hold the data block is preallocated during initialization of the WPOSnapshot object and is not supposed to be resized during the life of the object.

However, size of the performance data might grow in case of performance objects with instances, so it may happen that the preallocated buffer is not large enough to hold the whole data block. This condition is typically detected just before retrieving performance data from the registry interface. The buffer is resized and its new length is stored in `DataBlockSize` field of the `CWPOSnapshot` class. The reallocation might cause anomaly to appear in the measured performance data, which must be reported. WPOSnapshot object detect the reallocation by comparing buffer size to its value before interaction with the registry. If the new

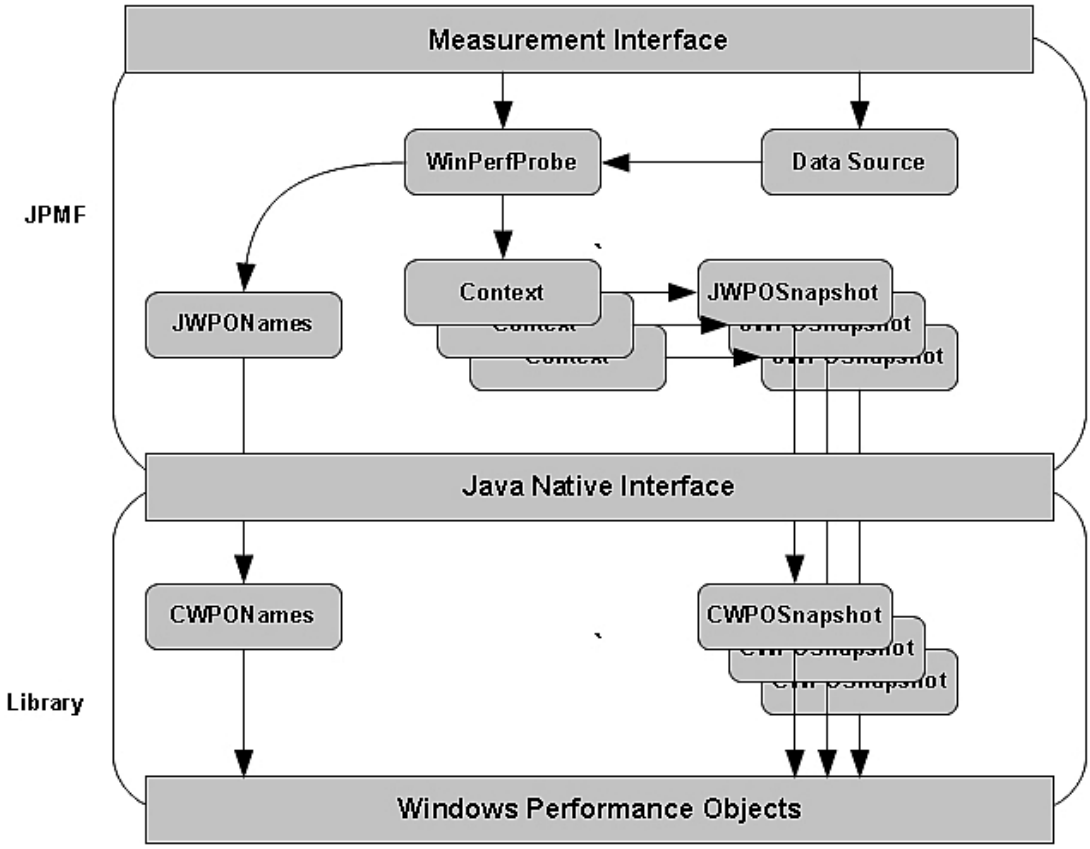


Figure 4.1: Implementation architecture

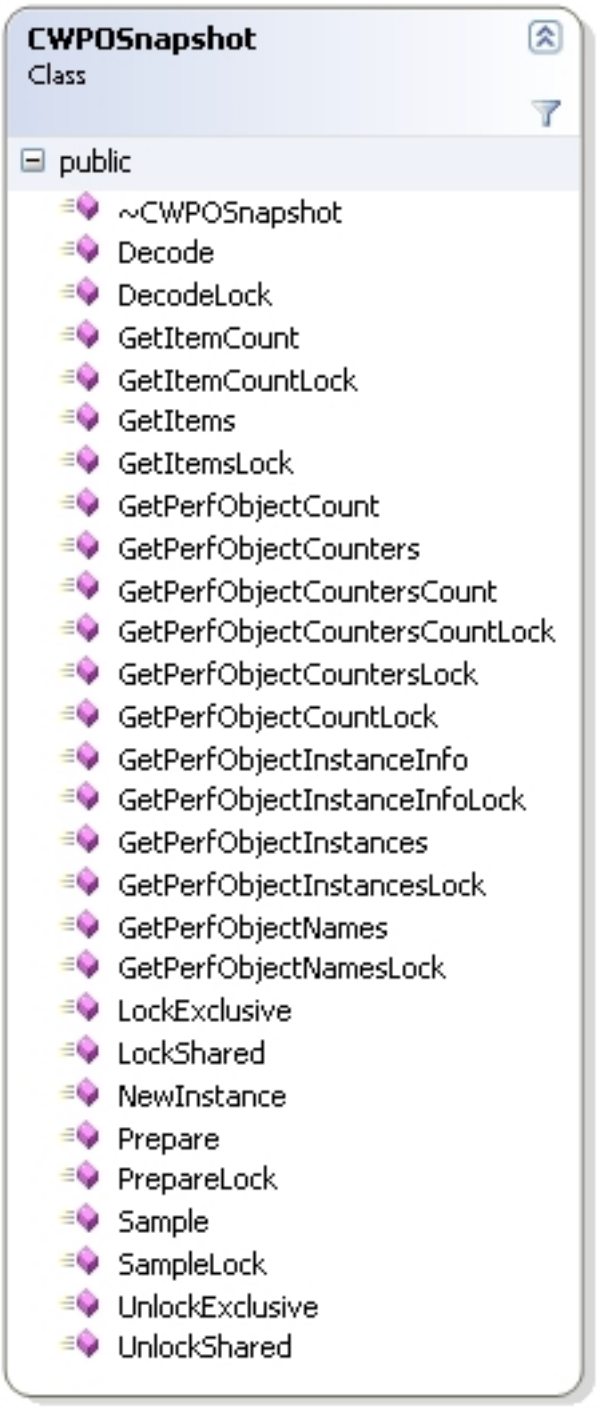


Figure 4.2: `CWPOSnapshot` class

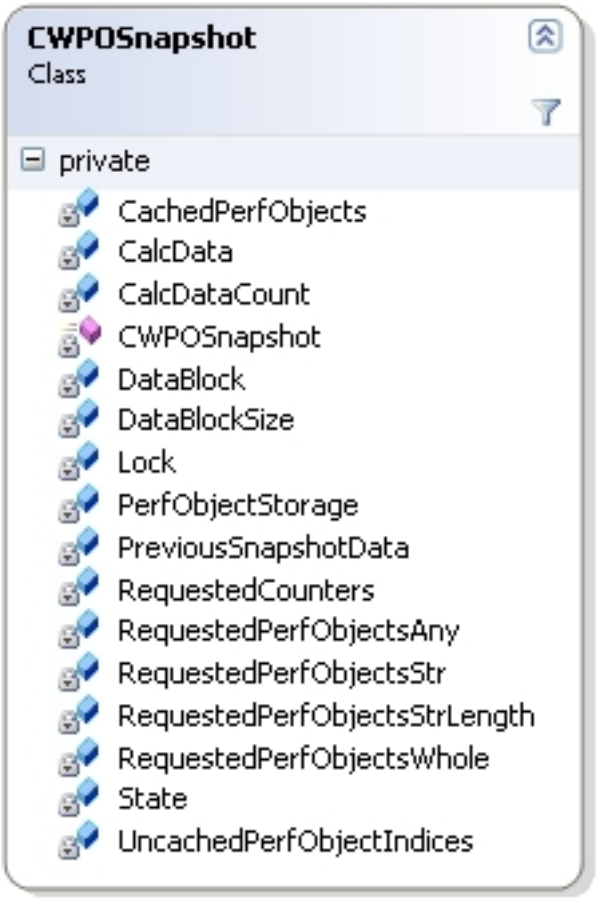


Figure 4.3: Private members of `CWPOSnapshot` class

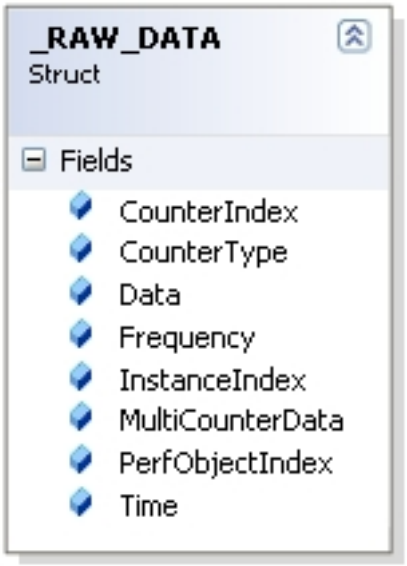


Figure 4.4: `RAW_DATA` structure

value is greater than the older one, the object reports anomaly during the data measurement.

The contents of `RequestedPerfObjectsStr` string is used when querying `HKEY_PERFORMANCE_DATA` pseudo hive. This string contains indices of all performance objects for which the performance data need to be collected because they have not been found in the performance object cache. The buffer to hold the string is preallocated during the initialization of WPOSnapshot object and is set large enough to last the whole lifetime of the object. The size of the buffer is stored in `RequestedPerfObjectsStrLength` field of `CWPOSnapshot` class.

WPOSnapshot object stores information about performance objects and counters to measure in three fields: `RequestedCounters`, `RequestedPerfObjectsWhole` and `RequestedPerfObjectsAny`. All these fields are implemented as STL sets and are filled during the initialization of the object. `RequestedCounters` contains information about performance counters the creator of the object wants to measure. This information is represented by pairs of performance object and counter name index. However, the caller might also request to monitor the whole performance object. Such requests are stored in the `RequestedPerfObjectsWhole`. The third set contains indices of performance objects which data must be collected, because either the caller requested to monitor some of its counters, or monitoring of the whole performance object has been requested.

Two fields of the `CWPOSnapshot` class are related to the performance object cache. `UncachedPerfObjectIndices` contains indices of the performance objects the data of which were not found in the cache. On the contrary, `CachedPerfObjects` field contains addresses of `PERF_OBJECT_TYPE` structures of cached performance objects. These members of the `CWPOSnapshot` class are used during sampling and decoding operations.

Some performance counters need more data snapshots, two at least, in order to calculate the displayable values from their raw data. Hence, field `PreviousSnapshotData` saves performance data from the previous snapshot. The data are

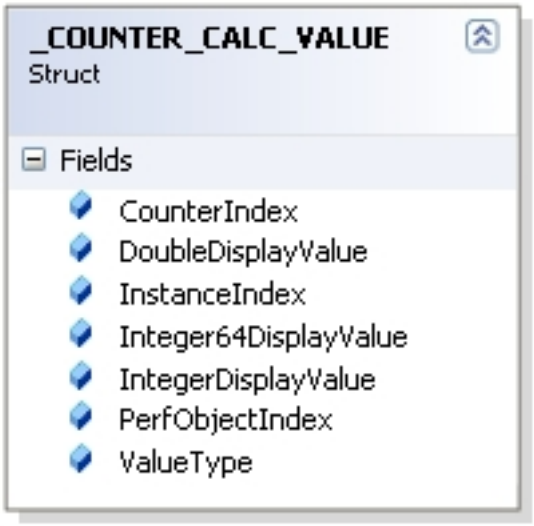


Figure 4.5: `COUNTER_CALC_VALUE` structure

Table 4.1: Places of displayable value storage, depending on its type

Type of value	`ValueType` **value**	**Place of storage**
32-bit integer	`ValueTypeInteger`	`IntegerDisplayValue`
64-bit integer	`ValueTypeInteger64`	`Integer64DisplayValue`
Double	`VauleTypeDouble`	`DoubleDisplayValue`

stored in form of `RAW_DATA` records. Every such record, the structure of which is shown in Figure 4.4, identifies one counter or counter instance, and contains enough information to calculate its displayable value from that snapshot. Some counters need two such records from two different snapshots to calculate their value.

Decoding operation transforms contents of one or two `RAW_DATA` records into one `COUNTER_CALC_VALUE` structure, shown in Figure 4.5. This structure is intended to hold displayable value for the particular counter and instance, uniquely identified by `PerfObjectIndex`, `CounterIndex` and `InstanceIndex` fields. Table 4.1 shows in which member of the structure the displayable value is actually stored, depending on its type.

Calculated counter values, which are the result of successful decoding operation, are stored in array of `COUNTER_CALC_VALUE` records pointed by `CalcData` member of `CWPOSnapshot` class. The length of the array is stored in `CalcDataCount` field. The contents of this array is visible outside WPOSnapshot object and may be collected by upper layers of code.

State of the WPOSnapshot object is stored in `State` member of the class. This member is an enumeration of `EWPOSnapshotState` type, shown in Figure 4.6. Individual states of the object were described in Section 3.3.

Figure 4.2 shows methods, the `CWPOSnapshot` class provides to other entities in the library scope. We do not plan to describe semantics of these method here. The reason is that the methods are, with certain preprocessing and postprocessing, directly used by routines, forming the JNI bridge between JPMF and the library. And semantics of these routines is described in Section 4.3. Table 4.2

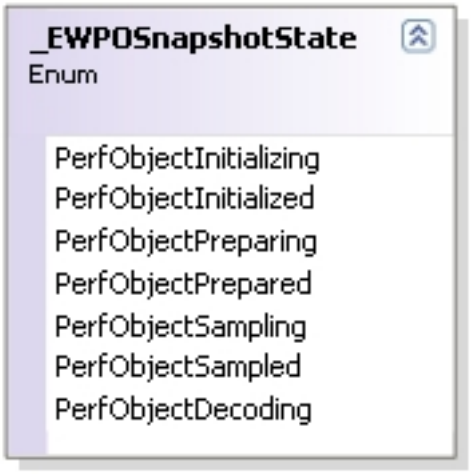


Figure 4.6: `EWPOSnapshotState` enumeration that stores actual state of the WPOSnapshot object

links corresponding routines and methods together.

The noticeable fact is that many methods are provided by the `CWPOSnapshot` class twice. Their names differ only by the `Lock` suffix. Methods with the suffix in the name are thread-safe variants of the others. The synchronization is performed via reader-writer lock. Shared access is allowed for methods that read the structure of the performance data. The lock is acquired exlusively when executing one of the methods responsible for the measurement operation.

4.2.2 Object Operations

Initialization

At first, the new instance of the object determines which performance objects is the object requested to monitor. The creator of the object expresses its demand by two input arguments passed to the `NewInstance` static method of `CWPOSnapshot` class. The first argument is of `EWPOSnapshotType` (displayed in Figure 4.7) and may contain one of the following values:

- `wposGlobal`. The caller requests to monitor all performance objects which data can be collected without high consumption of memory and processor time. Such performance objects are members of Global category.
- `wposCostly`. The caller requests to retrieve performance data collection of which is very expensive and may take some time. Such performance objects are members of Costly category.
- `wposCustom`. The caller requests to monitor only certain performance object and single performance counters.

The second argument of the `NewInstance` method is an array of `WPOSNAPSHOT_INPUT_INFO` (Figure 4.8) records and is used only in case the first argument is set to `wposCustom`. Otherwise, the caller can set the value of the argument to `NULL`. Each of the records specifies one performance counter, or the whole performance

Table 4.2: `CWPOSnapshot` class methods and corresponding routines of the JNI bridge

Class method	**Routine of the JNI bridge**
`GetPerfObjectCount`	`getSnapshotPerfObjectCount`
`GetPerfObjectNames`	`getSnapshotPerfObjectNames`
`GetPerfObjectCounters`	`getSnapshotPerfObjectCounters`
`GetPerfObjectCountersCount`	`getSnapshotPerfObjectCountersCount`
`GetPerfObjectInstanceInfo`	`getSnapshotPerfObjectInstancesLength` `getSnapshotPerfObjectInstancesCount`
`GetPerfObjectInstances`	`getSnapshotPerfObjectInstances`
`GetItemCount`	`getSnapshotItemCount`
`GetItems`	`getSnapshotItems`
`NewInstance`	`createSnapshot`
`~CWPOSnapshot`	`destroySnapshot`
`Prepare`	`prepareSnapshot`
`Sample`	`sampleSnapshot`
`Decode`	`decodeSnapshot`

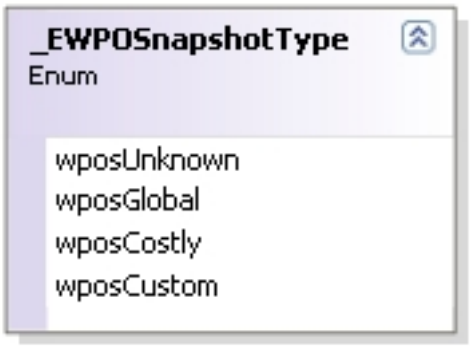


Figure 4.7: `EWPOSnapshotType` enumeration

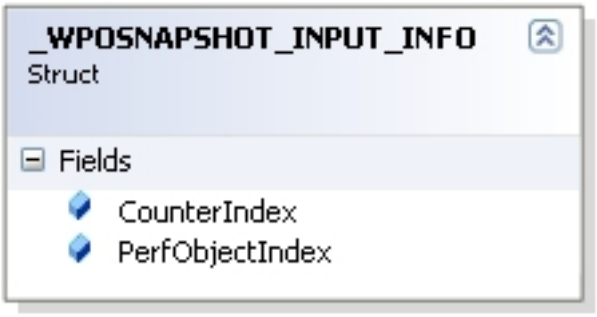


Figure 4.8: `WPOSNAPSHOT_INPUT_INFO` record

object, which the Wposnapshot entity must monitor. The record contains only two fields: `PerfObjectIndex` to specify name index of the target performance object, and `CounterIndex` to determine exact counter. If the latter is set to `0`, the whole performance object with name index stored in the `PerfObjectIndex` field is monitored.

According to the values of these input arguments, the WPOSnapshot object fills `RequestedCounters`, `RequestedPerfObjectsWhole` and `RequestedPerfObjectsAny` sets. All sets are queried during the Decode operation and determine which which counter values should be calculated and copied through JNI to JPMF. The third set is also used during Prepare and Sample operations because it contains name indices of performance objects that should be measured.

When the parsing of input arguments is finished, it is time to allocate storage to hold the performance data snapshot. The whole snapshot is not stored in `DataBlock` buffer because the registry interface is used only to collect data for performance object that are not present in the cache. Before the Decode operation, performance objects from the cache and from the data block are copied into Performance Object Storage represented by `CPerfObjectStorage` class in field `PerfObjectStorage` of the `CWPONames` class. The main purpose of this class is storing contents of performance objects and their time information.

To initialize the `PerfObjectStorage` field, snapshot of all performance objects with name index in `RequestedPerfObjectsAny` set is performed. Then, buffer of every performance object, found in the snapshot, is allocated and stored inside the `CPerfObject` class instance. During the life of the WPOSnapshot object, the buffers are reallocated only when the size of certain performance object grows. This means that the number of instances changed and this situation is considered as measurement anomaly that should be reported to JPMF.

Before the snapshot is taken, `RequestedPerfObjectsStr` is preallocated large enough to hold the name of value that instructs the registry interface to retrieve data of all performance objects specified by the creator. The buffer is never reallocated during the life of the object because the name of the value cannot be longer. Actually, it might be often shorter when some performance data are cached.

When the Performance Object Storage is successfully initialized, `PreviousSnapshotData` field which is actually a vector of `RAW_DATA` records, is allocated and filled with performance data obtained during the initialization of Performance Object Storage. This ensures ability to compute counter values just after the first

snapshot taken after the initialization phase, even for counters that need more snapshot for successful value calculation.

Determining the Structure of the Snapshot

When the WPOSnapshot object is initialized completely, it allows, apart from measurement, to explore the structure of the snapshot. External entities might use its interface to find out which performance counters are present in the monitored performance objects and which instances they consist of. These information are taken from Performance Object Storage class where the last snapshot is saved.

Measurement

This operation is divided into three phases: preparation, sampling and data decoding. These phases match exactly `prepare`, `sample` and `decode` methods of the Java interface built on top of the library by the `ProbeContext` entity.

The Prepare operation is quite simple and straightforward. The WPOSnapshot object only initializes members of its class related to the performance object cache, and sets its state to `PerfObjectPrepared`. The Prepare operation can be performed only when the object is in either `PerfObjectInitialized`, or `PerfObjectSampled` state.

The Sample operation is more complex, however, it is designed to proceed quickly. At first, the object retrieves as much data of requested performance objects from the cache as possible. It builds a list of performance objects the data of which were not found in the cache, and converts it to the string value which is passed to the registry interface. During construction of this value, no memory allocation or other operations are needed because the buffer holding the string is preallocated large enough during the object initialization.

When the data block is filled with fresh data of performance objects not found in the cache, WPOSnapshot object checks whether the data block was reallocated or not. If yes, a measurement anomaly is signalled.

When the Sample operation is successful or even when measurement anomaly occurs, WPOSnapshot object enters `PerfObjectSampled` state. Its data block contains newly measured performance data and performance objects retrieved from the cache are stored in `CachedPerfObjects` member of `CWPOSnapshot` class. When the object is in `PerfObjectSampled` state, it is not possible to retrieve information about the structure of individual performance objects. As explained earlier, this information is taken from its instance of Performance Object Storage class, which, after successful sampling, might not reflect characteristics of newly measured performance objects. The Sample operation can be performed only when the object is in `PerfObjectPrepared` state.

Data decoding operation starts with merging cached performance objects with contents of the data block. In this step, the fresh performance data from the data block are also inserted into the the cache. After the merge operation, the instance of Performance Object Storage contains both cached and fresh performance data for the new snapshot.

Then, the performance data are converted to array of `RAW_DATA` records, each contains data related to one particular requested counter, or one particular instance of it in case of counters with instances. Counters which values are not

requested by the creator of the Wposnapshot object, are skipped. After that, WPOSnapshot object starts to compute values for individual counters and instances. Performance data from previous snapshot, needed in some cases, are stored in `PreviousSnapshotData` member of the `CWPOSnapshot` class instance.

The records in both `RAW_DATA` arrays (the new array created from the current snapshot and the old one stored in `PreviousSnapshotData` field of WPOSnapshot object structure) should be identical except the performance values themselves. Violation of this condition implies that the structure of the snapshot has changed, for example, an instance had been added or deleted. In such a case, new Prepare and Sample operations are requested. This request, however, does not reach Java portion of the project because it is resolved by the code just below Java Native Interface (JNI) which provides connection between Java and components written in C++.

WPOSnapshot object may enter data decoding operation only from `PerfObjectSampled` state. When the operation finishes, the object is put into `PerfObjectInitialized` state. When the decoding operation succeeds, newly calculated counter values might be obtained by external entities.

4.3 The JNI Interface

Our data source for performance measurement is mainly written in C++ language which cannot be directly used in Java applications. However, JNI allows to build a bridge between these languages. Java end of this bridge is represented by native methods with their counterparts exported by dynamic link library written in C++. This section contains definitions of these methods and describes their behaviour.

The library exports functionality of its two main abstractions: index-to-name translation and WPOSnapshot object. Exported functions are connected to Java classes: `JWPONames` and `JWPOSnapshot`, which represent the abstractions in the world of JPMF. Instances of these classes are used to build the whole infrastructure of our data source.

4.3.1 Name Index Translation Interface

`JWPONames` class is connected to the library through three native methods that provide all functionality needed to translate name indices to corresponding string values. Definitions of the methods are the following:

```
int getCount();
int getByteLength();
int getNames(char[] aNames, int[] aIndices);
```

The first method (`getCount`) returns number of entries in the translation table. The second method retrieves total length of all strings in the table, including their terminating null characters plus one extra null character which will be used by the `getNames` method to signal the real end of valid data.

The third method (`getNames`) serves the most complicated task of the three ones. Its purpose is to copy the whole contents of the translation table to Java

part of our data source. The method accepts two arguments:

- `aNames`. Caller-allocated character array that the library fills with all strings found in the translation table. The strings are separated by null characters and the end of the valid data is signaled by two null characters following immediately after each other. The caller must allocate this array large enough to hold all strings of the translation table. Routine `getBytelength` should be used to retrieve required size of the array.

- `aIndices`. Array of integers that the library fills with name indices for the individual strings stored in the first argument of the method. The array must be preallocated large enough to hold indices for all strings in the translation table. Required length of the array should be obtained by invocation of `getCount` method. The order of retrieved name indices respects the order of strings stored in the `aNames` argument. This means that the first name index correspond to the first string stored in the character array, and so on.

The method returns `0` when the operation succeeds and negative value otherwise.

As mentioned earlier, the translation table is in library context represented by `CWPONames` class. The library, however, does not export any routine that can be used to create or destroy instance of this class. There is also no exported function providing name index translation service. The reason of absence of such routines is that only one instance of `CWPONames` class is needed to serve all translation requests. This instance is created during library initialization, and is destroyed just before the library is unloaded from the process‘ address space.

The library does not export any routine for name index translation because calls through JNI are believed to be more expensive than solving the problem purely in Java. The library exports routine that allows to copy the whole translation table to internal members of instance of `JWPONames` (Java) class. All translation requests are then handled by the instance of the class without need to pass them through JNI. And this is exactly, how `JWPONames` class works.

4.3.2 WPOSnapshot Object Interface

`JWPOSnapshot` class is connected to the library through much more native methods than `JWPONames` one. The methods can be divided into three main categories by their purpose. The first category consists of methods responsible for creation and destruction of the WPOSnapshot objects. Those in the second category provide information about the structure of WPOSnapshot object stored in the scope of the library, such as number of performance objects, their names and counter details. Members of the third category allow our data source to collected performance data and decode values of individual counters.

Creation and Destruction

Single instance of `CWPOSnapshot` class represents one WPOSnapshot object in the library scope. The library exports two methods that allow our data source to create and destroy these objects. Their declarations look as follows:

```
int createSnapshot(int aSnapshotType, WPOSnapshotInputInfo[]
aInputInfo);
void destroySnapshot(int aSnapshotHandle);
```

An instance of the `JWPOSnapshot` class is created together with creation of an instance of the class, implementing `ProbeContext` interface for our data source. Constructor of `JWPOSnapshot` class calls `createSnapshot` method in order to create new WPOSnapshot object in the library scope. This method accepts two arguments: an integer describing type of the snapshot,and an array of `WPOSnapshotInputInfo` classes that contains information about the requested performance data. Both arguments are directly passed to the call of `CWPOSNapshot ::NewInstance` method. Values of the first argument directly correspond to the `EWPOSnapshotType` enumeration. `WPOSnapshotInputInfo` instances are equivalent to `WPOSNAPSHOT_INPUT_INFO` records.

If the call to `createSnapshot` succeeds, positive integer value is returned and stored inside the instance of `JWPOSnapshot` class. This value is called *a handle*.

Handles uniquely identify `CWPOSnapshot` objects created in the library scope. Native methods discussed in this subsection, with exception of `createSnapshot`, always require to specify target object by handle value passed in `aSnapshotHandle` parameter. Handle values are converted to addresses of corresponding WPOSnapshot objects as the very first operation of routines exported by the library to satisfy JNI requirements. We decided to use this concept because of the following reasons:

- Identification of WPOSnapshot object by their direct addresses gives JPMF ability to indirectly read or change contents at potentially arbitrary memory addresses. We think that Java applications should not posses such a power.
- We are able to determine whether the caller works with an existing WPOSnapshot object and when it attempted to use nonexistent one for some reason.

Indirect identification of WPOSnapshot objects is also the reason why many methods accepting handles have return value, although JPMF definition forbid them to fail. Such methods can fail mainly because of invalid handle passed in their arguments. For example, `decodeSnapshot` method described later follows this rule.

Semantics of `destroySnapshot` method is quite straightforward. It instructs the library to destroy WPOSnapshot object, identified by its handle. This method should be called explicitly by our data source when the instance of WPOSnapshot class is no longer needed.

Object Structure Browsing

The second category contains the following methods:

```
int getSnapshotPerfObjectCount(int aSnapshotHandle);
int getSnapshotPerfObjectNames(int aSnapshotHandle, int[] aIndices);
int getSnapshotPerfObjectCountersCount(int aSnapshotHandle,
```

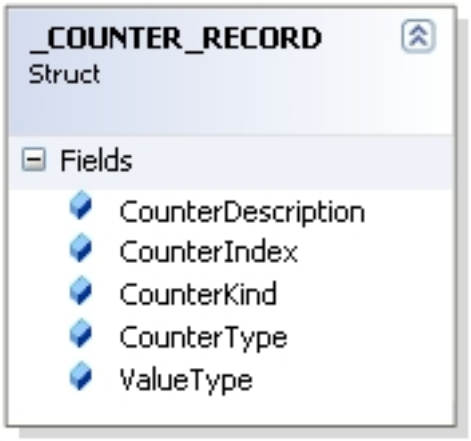


Figure 4.9: `COUNTER_RECORD` structure

```
int aPerfObject);
int getSnapshotPerfObjectCounters(int aSnapshotHandle,
int aPerfObject, CcounterRecord[] aCounterRecords);
int getSnapshotPerfObjectInstancesCount(int aSnapshotHandle,
int aPerfObject);
int getSnapshotPerfObjectInstancesLength(int aSnapshotHandle,
int aPerfObject);
int getSnapshotPerfObjectInstances(int aSnapshotHandle,
int aPerfObject, char[] aNames, int[] aIds);
```

The first two methods retrieve basic information about about performance objects present in the current snapshot of the performance data. Number of these groups is returned by `getSnapshotPerfObjectCount` method, while `getSnapshotPerfObjectNames` routine is responsible for retrieval of name indices of the performance objects. The indices are written to array passed in the `aIndices` argument. The array must be preallocated large enough to hold the indices of all performance objects present in the current snapshot.

The second pair of methods provide information about counters belonging to the certain performance object. The performance object is specified by its name index passed in `aPerfObject` argument. Again, the first method of the pair returns number of counters present in the given performance object, whereas the second one, named `getSnapshotPerfObjectCounters`, writes detailed information about individual counters of the performance object into the array passed in `aCounterRecords` argument. The array must be large enough to hold information about all the counters. Each performance counter is represented by single `COUNTER_RECORD` structure, depicted in Figure 4.9. Table 4.3 briefly describes meaning of individual structure members.

The purpose of the last three methods is to retrieve information about counter instances. Name index of the target performance object must be specified in `aPerfObject` parameter. The semantics of these methods is very similar to the native ones stored in WPONames class. Number of counter instances is returned by `getSnapshotPerfObjectInstancesCount` method and total length of their names in bytes, including terminating null characters plus extra null character to

Table 4.3: Description of the COUNTER_RECORD structure members

Member name	Description
CounterDescription	Index of the description of the counter.
CounterIndex	Index of the counter name.
CounterType	Raw type of the counter that is stored in the counter definition in the performance data structure.
CounterKind	JPMF classification of the sensor representing the counter. It can be either counter, or gauge.
ValueType	Type of counter value. There are constants defined for 32-bit integer, 64-bit integer and decimal value types.

signal end of valid data, is retrieved by the `getSnapshotPerfObjectInstancesLength` method. Method named `getSnapshotPerfObjectInstances` writes names of the instances into `aNames` array and their parent performance object indices into `aIds` one. Both arrays must be preallocated large enough to hold information about all instances of the performance object. Retrieved instance names are separated by null character and the last one is followed by two such characters to indicate the end of the valid data in the array.

Instance names, retrieved by `getSnapshotPerfObjectInstances` method are the strings written in the the instance definition structures inside the structure of their performance object. This means that such names are not guaranteed to be unique and, in some cases, they might not match format declared in Section 3.2. Their transformation to this format is performed when the sensors representing performance counters of the performance object are being created.

Method `getSnapshotPerfObjectInstances` does not explicitly retrieve information about instance order within the target performance object. However, this information is implicitly contained in the order of information written in `aNames` and `aIds` array which is the same as the order of instance definition structures within the performance data snapshot.

Performance Data Retrieval and Decoding

There are three native methods responsible for three phases of performance data measurement operation, and two methods for data transfer to JPMF world:

```
int prepareSnapshot(int aSnapshotHandle);
int sampleSnapshot(int aSnapshotHandle);
int decodeSnapshot(int aSnapshotHandle);
int getSnapshotItemCount(int aSnapshotHandle);
int getSnapshotItems(int aSnapshotHandle, CalcValue[] aValues);
```

Actually, the `prepareSnapshot` and `decodeSnapshot` methods, responsible for the first and the last stage of performance data measurement, have been already described in the beginning of the subsection. They accept only one argument, handle of the target WPOSnapshot object, and should always return `0`, meaning

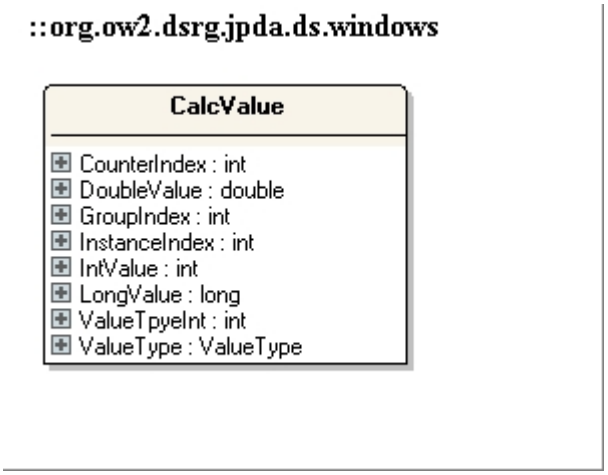


Figure 4.10: `CalcValue` record definition

Table 4.4: Meaning of individual members of the `CalcValue` structure

Field Name	**Description**
`DoubleValue`	If the counter is of decimal type, this field contains its calculated value.
`IntValue`	If the counter type is 32-bit integer, this field contains its calculated value.
`LongValue`	If the counter type is 64-bit integer, this field contains its calculated value.
`ValueType`	Type of the calculated value. This member is filled after the structure is fetched from JNI.
`PerfObjectIndex`	Name index of performance object the counter belongs to.
`CounterIndex`	Name index of the counter.
`InstanceIndex`	Order index of the group instance.
`ValueTypeInt`	Contains type of the calculated value during transport through JNI. Values of the field reflect ordinal values of the `ValueType` enumeration.

success of the operation. The second stage of the measurement – performance data sampling – is implemented by the `sampleSnapshot` method which return value indicates result of the sampling operation. Success is signaled by returning `0`, anomaly by returning `-6`, and complete failure by other negative value.

The return values correspond to the internal library error codes. Success of the operation is always signaled by returning zero. Errors are signaled by negative values. Value of `-6` is hardcoded to indicate a measurement anomaly.

Library exports two routines to allow transport of calculated (decoded) sensor values into JPMF environment. They can be called only after the Decode operation. Each calculated value is represented by `CalcValue` record, which structure is depicted in Figure 4.10 and its individual members described in Table 4.4. Semantics of the methods is the same as in case of `getSnapshotPerfObjectCount` and `getSnapshotPerfObjectNames` methods.

4.4 Performance Object Cache

This section describes details related to implementation of performance counter caching. It covers data structures used by the cache and more in-depth description of supported operations.

4.4.1 Data Structures

As mentioned in Section 3.4, the performance object cache consists of three data structures: Cache Table, Banned Entity List and Outdated List.

Outdated List is implemented just as a dynamic array of cache items. The reason of this decision lies in fact that this list would never contain many items. The list serves only as a storage for cache items that contain old performance data, however, they cannot be freed because some WPOSnapshot objects keep references to the items. And this state never holds for long.

Banned Entity List is represented as an instance of `std::multimap`. Name index of a performance object is used as a key, entity identifier (which is, in fact, address of the `CWPOSnapshot` class instance of the entity) is stored as a data. This organization makes the operation of blocking entity from accessing certain performance objects very fast. The similar applies to access unblocking.

The responsibility of the Cache Table is to perform fast translations between performance object name indices and corresponding cache items. This can be effectively handled by unordered map. The table stores at least one cache item per name index, so an unordered multimap is not required.

The structure of the cache item is named `CPerfObjectCacheItem` and is depicted in Figure 4.11. The `PerfObject` member contains address of the performance object associated with the item. Name index of the object is stored in `PerfObjectIndex` item. Actual size of the performance object is held by `PerfObjectSize` member. Time information related to the measurement are stored in the form of `DATA_BLOCK_TIMES` structure (shown in Figure 4.12) in `DataBlockTimes` field. Time of insertion to the cache is stored in `Time` field. Every cache item is also provided with reader-writer lock (stored in `Lock` field) which ensures atomicity of its changes and other operations. `ReferenceCount` field is used to keep track of number of active pointers to the associated performance object (the cache uses direct pointers to the cache items, other entities work with pointers to performance objects, because cache items are not visible outside the cache scope). When the number of references reaches zero, memory occupied by the cache item and associated `PERF_OBJECT_TYPE` structure is freed.

Cache Table stores at most one cache item for each performance object. This item represents the newest performance data for the performance object obtained through Windows Performance Objects interface. When an entity attempts to insert new performance data for already cached performance object, the address of `PERF_OBJECT_TYPE` structure is replaced with the address of the new structure and the old one is, in case some active pointers outside the cache reference it, moved to the Outdated List. Attempt to insert data for performance object, that is not yet present in the cache, results into creation of a new instance of the cache item, and ist association with newly measured data. The cache never uses location of new data given on the input of the operation in its internal data

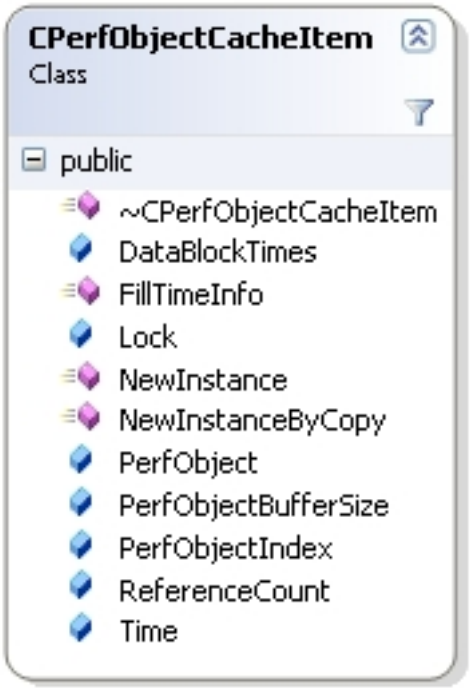


Figure 4.11: `CPerfObjectCacheItem` structure

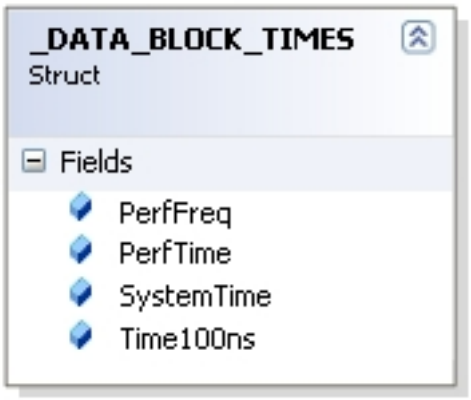


Figure 4.12: `DATA_BLOCK_TIMES` structure

structures, it copies these data to newly allocated buffer instead.

It may happen however, that the size of the new `PERF_OBJECT_TYPE` structure is equal or less than the size of the structure which is about to be replaced. In such cases, data are simply copied from the new structure to the location of the old one, effectively destroying its previous content, and no buffer allocation for the copy of the new data is performed.

4.4.2 Operations

The cache as a whole supports five operations: Insert, Find, Release, Lock and Unlock.

Find

An entity attempts to perform the Find operation when it tries to retrieve cached data for certain performance object. On input, the entity specifies name index of the performance object, its entity unique identifier (address of its `CWPOSnapshot` class instance) and address of the `DATA_BLOCK_TIMES` structure which the routine, in case it finds cached data, matching the request, fills with timing information for retrieved performance object. Output of the operation is either address of `PERF_OBJECT_TYPE` structure with requested cached data or null pointer, if there is no matching performance data in the cache. The Find operation does not copy content of the performance object, it just returns address of the data associated with matching cache item.

The operation proceeds as follows: at first, the Banned Entity List is searched for pair that would disallow the entity from retrieving cached measurement data for performance object of given index. If such pair exists, the the cache treats as if no cached data of that object were present.

If the entity is allowed to obtain data of the performance object, the Cache Table is searched for item with the same value of performance object index as the input one. If such item does not exist, the Find operation fails. Outdated List is not searched because the entity expects fresh measurement data not the an useless old stuff.

When the Cache Table contains matching cache item, its insertion time is compared with value of the current time which effectively determines age of the performance data associated with the item. If the age reaches beyond the threshold (500 ms), the data are treated as old and the Find operation fails again.

If the performance data are not considered too old, the entity is banned from further attempts to retrieve the same data from the cache again, until the cache item is updated by other entity. Input time information structure is filled and address of the performance object associated with the cache item is returned. Reference count of the cache item is increased by one.

Insert

An entity performs this operation after successful measurement made through Windows Performance Objects interface, in order to fill the cache with fresh performance data. On input, the entity specifies address of the performance

object with newly measured data, address of data block of the measurement and address of the `CWPOSnapshot` class instance as an entity identifier.

At first, the cache attempts to locate cache item associated with performance data of the given performance object. If such an item exists, the cache updates its time information from the data block given by the entity. If the storage for the performance object data, pointed by `PerfObject` member of cache item structure, is large enough to hold the new data, the content of the supplied `PERF_OBJECT_TYPE` structure is copied to the performance object storage of the cache item. Otherwise, new buffer is allocated and the new performance data are copied to it. Then, this buffer is associated with the cache item (the `PerfObject` member is directed at the buffer). If there are no outside references to the cache item, the old performance object storage is simply freed. However, if there is at least one entity working with the old storage, the storage is moved to the Outdated List under the appropriate name index. Moving to this data structure consists from copying contents of the cache item instance to newly allocated memory and inserting this copy to the list.

When the Insert operation succeeds, the cache also updates the Banned Entity List to reflect the performance data update. All pairs of form (pi, x) where pi is the index of the input performance object and x stands for arbitrary entity identifier are removed, and pair of form (pi, eid), where eid represents unique identifier of entity that performed the Insert operation, is added. This effectively allows all entities to access the newly inserted performance data with exception of the one that actually has a copy of the data.

The Insert operation does not change the reference count of the cache items. Newly created cache items are referenced only inside data structures internal to the cache.

Release

Every successful Find operation increases reference count of a cache item, representing certain performance object, by one. An entity performs a Release operation when it no longer needs to work with the cached data. the Release operation is therefore dual to the Find operation.

The Release operation causes the cache to locate the item connected to the performance object represented by `PERF_OBJECT_TYPE` structure in its internal data structures. At first, the cache table is searched. If the item is found here, its reference count is decremented by one. In this case, the item is not freed even if only internal references to it remain.

If there is no suitable cache item located in the Cache Table, the Outdated List is traversed. The item must lay within this data structure. Otherwise, the entity attempted to release performance object that is not cached, and this behaviour is considered as a bug of the library. The reference count of the corresponding cache item is decreased by one. If only internal references remain active, the item is removed from the Outdated List and freed.

Locking

It was already mentioned that every cache item has its own reader-writer lock to synchronize operations on its content. This lock can be acquired also by entities

from outside the cache through the Lock and Unlock operations.

The Lock operation causes the cache to find the cache item corresponding to the given performance object, either in the Cache Table, or in the Outdated List. The lock is acquired in shared mode and ensures that the cache will not manipulate with the performance object data until the lock is released. Other entities are allowed to read the cached performance object as well.

The Unlock operation works in the following way: the cache locates cache item responsible for the given performance object and releases its reader-writer lock. Again, the Cache Table is searched first, and if the corresponding item is not there, the cache resorts to search the Outdated List.

The implementation of the cache assumes that the entities attempt to lock or unlock only performance objects (represented by `PERF_OBJECT_TYPE` structure) actually present either in the Cache Table, or in the Outdated List. Attempt to lock and/or unlock performance object, not present in the cache, is treated as a bug in the library and should never happen.

4.5 Reader-Writer Lock Implementation

The library is designed to allow multiple threads to access snapshot of performance data in parallel. Access to WPOSnapshot objects and performance object cache is synchronized by the reader-writer lock primitives. A thread acquires the lock for shared access when it intends to read the content of the cache or the performance data snapshot. It uses exclusive mode when modifying the cache or in case of new measurement.

Unfortunately, not all currently supported versions of Windows provide reader-writer lock synchronization primitive to user mode applications (kernel mode drivers can benefit from Executive Resources or Pushlocks). Windows Vista introduces Slim Reader Writer Locks (SRWLs). However, Windows XP does not provide anything like that.

To ensure compatibility of the library with different versions of the Windows operating system, we had to create our own implementation of the reader-writer lock primitive which will be used on Windows XP. On Windows Vista and newer versions of the operating system, the library benefits from SRWLs. This section describes our implementation of the primitive.

4.5.1 Helper Primitives

Our implementation of the lock is built with help of two synchronization primitives – *critical sections* and *keyed events*. The former is used to synchronize access to the lock structure, the latter is used to implement passive waiting when a reader thread attempts to acquire the lock, already owned exclusively, or when a writer thread wants to enter critical region already occupied by some other threads.

Critical Sections

Critical sections are one of the main synchronization primitives that Windows provides to usermode applications on top of the kernel-level synchronization prim-

itives [3, p194]. Their main advantage over standard kernel synchronization primitives (events, mutexes, semaphores, keyed events...) lies in saving a round-trip to kernel mode in case the critical section is not contended. In other words, a critical section can be acquired and released without need of any system call. This holds when there are no threads waiting for its ownership.

Critical sections use architecture-specific test and set instruction to perform the acquire and release operations. When a thread attempts to acquire a critical section which is already owned, it waits for synchronization event object that is part of the internal state of the critical section. The event is set to signaled state by the actual owner of the section when it resigns to the ownership. The critical section creates this event object when it needs to suspend a thread for the first time.

Critical sections provide only mutual exclusion. According to [3, p194] they can also be acquired in shared mode, however, other resources, including the MSDN Library, do not confirm this statement. [4, p201-2], however, clarifies the statement, which is now used to describe characteristics of an undocumented primitives called *user-mode resources*. These primitives are present in the operating system from Windows NT 3.1 and can be understood as a port of the Executive Resources to usermode. Probably, If we knew this earlier, we probably would not implement our own reader-writer locks and used user-mode resources, although our primitive has certain minor advantages, like smaller size of the data structure.

So, the critical sections cannot provide good implementation of the reader-writer lock only on their own. In many aspects, their performance is comparable to spin locks. Actually, they can be configured in a way that blocking threads perform a period of busy-waiting (spinning) rather than waiting for the synchronization event instantly. This behavior can avoid additional system calls. The semantic similarity to spin locks predetermines the critical sections to be used for synchronizing access to smaller data structures.

Keyed Events

Keyed events were originally implemented to help programs to deal with low memory situations when using critical sections [4, p194] [5, p268]. As mentioned above, the critical section object, when being acquired, might allocate a synchronization event. In case of full process handle table or when there is not enough free memory in paged and nonpaged pool, the allocation might fail. Windows 2000 did not handle such a situation very well; the critical section object was left in inconsistent state [4, p194-5] and an exception was generated. Windows XP introduced keyed events that solved the problem.

Keyed events are kernel synchronization objects. This means that they support passive waiting like , e. g., standard events or semaphores. However, their semantics is quite different. Keyed events supports two operations: *wait* and *release*. The operations are not intended to guard critical regions, however, they can help to implement more sophisticated synchronization primitives, like condition variables or reader-writer locks. And Windows Vista takes advantage of it; keyed events are used to implement condition variables and SRWLs.

When a thread attempts to perform a *wait*on a keyed event, it must specify a special value called *a key*. The thread is suspended and, together with the key

Table 4.5: Native functions working with keyed events

Function name	Description
`NtCreateKeyedEvent`	Creates a new keyed event. Because they are executive objects, keyed events can have names and security descriptors.
`NtOpenKeyedEvent`	Retrieves handle of the existing keyed event.
`NtWaitForKeyedEvent`	Waits for the keyed event. The wait can be alertable and it is also possible to wait only for certain time interval.
`NtReleaseKeyedEvent`	Releases a keyed event. This operation is blocking. The wait can be alertable and it is also possible to block only for certain time interval.
`NtClose`, `CloseHandle`	Closes handle to the keyed event object, retrieved by `NtCreateKeyedEvent` or `NtOpenKeyedEvent`.

value, it is stored within internal structure of the keyed event object.

Release operation is the opposite to the wait operation. Again, a thread must specify value of the key. The internal keyed event structure is searched for the given key value. If there is a thread waiting on the same value, it is resumed and removed from the internal structure. If no such thread exists, the thread that is performing the release operation, is suspended and stored in the internal keyed event structure under the given key until another thread performs the wait operation on the same key.

The key value is pointer-sized and unique in the context of the process which owns the thread that uses the value. This means that multiple processes can use one keyed event object without any unwanted interactions, except possibly increased time complexity. Keyed events can only be used to synchronize threads of the same process.

The above description suggests that keyed events are designed to solve rendezvous problems and are really helpful when we need to atomically release a lock (for example, a critical section) and start passive waiting. Which is exactly the operation needed for condition variables. Semantics of the keyed events ensures that early resume signals are never missed.

Interface to the keyed events is neither officially documented, nor exported via Windows API. Native API functions represent the only way of accessing this kind of objects. Relevant functions are listed in Table 4.5 together with their short characteristics.

Slim Reader Writer Locks

Slim Reader Writer Locks are implemented with help of keyed events and a test and set instruction. Their behavior is nearly identical to pushlocks, undocumented lightweight primitives used by the OS kernel. They are pointer-sized and prefer neither readers, nor writers. The only difference from pushlocks is that acquire mode of the SRWL cannot be converted (it is not possible to convert shared mode to exclusive and vice versa).

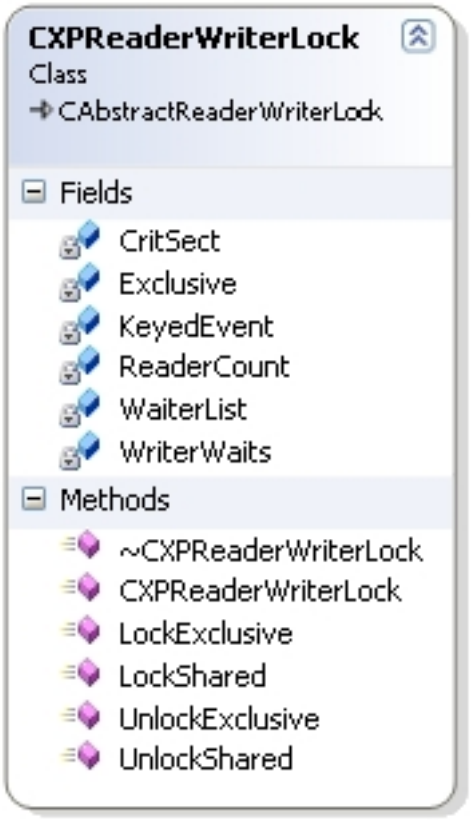


Figure 4.13: `CXPReaderWriterLock` class

4.5.2 Custom Reader-Writer Lock Implementation

`CXPReaderWriterLock` class that represent our reader-writer lock is displayed in Figure 4.13. The class contains mainly a critical section (`CritSection`), a handle to a keyed event object (`KeyedEvent`) and a list of waiting threads (`WaiterList`), which is used as a FIFO (waiters are inserted at its tail and removed from the head). The lock also counts actual number of reader threads (`ReaderCount`), and number of writers (`WriterWaits`), waiting to take its ownership. There is also a flag indicating whether the lock is exclusively owned (`Exclusive`).

Figure 4.13 shows that `CXPReaderWriterLock` class is derived from `CAbstract-ReaderWriterLock` class. This abstract class serves as an interface that defines methods every kind of the reader-writer lock must implement. There is also a `CVistaReaderWriterLock` class which implements reader-writer lock on the top of Slim Reader Writer Lock primitive. This class is not described here because its implementation is not interesting from any point of view – its methods work as redirectors to corresponding Windows API routines working with SRWLs.

The critical section object and the keyed event handle are initialized during creation of the lock. If an attempt to create its own keyed event fails, handle of the global keyed event is used instead. The global keyed event is allocated during initialization of the library.

Every lock prefers to use its own keyed event because internal implementation of the object on Windows XP keeps waiters in doubly linked list. Hence, it is desirable to keep the number of waiters in the list as small as possible. Keyed

event code traverses the whole list during both wait and release operations. In case, our reader-writer lock uses its own keyed event (not the global one), the release operation is typically performed in constant time. If it happens after the wait operation, thread being resumed is guaranteed to be placed as the first entry in the list. The reader-writer lock deallocates the keyed event object and deletes the critical section during its cleanup.

Acquiring in Shared and Exclusive Mode

When a thread attempts to acquire the lock, it first locks the critical section. Then it checks, whether it should block. If not, the thread successfully acquires the lock (increases the number of readers, or sets exclusivity flag to `1`) and releases the critical section. No keyed event is needed for this scenario. Additionally, the critical section is configured to spin some time before suspending the thread. Hence, the probability of a trip to kernel is very low.

Different scenario occurs when the thread realizes that it should block on the lock. This happens in one of the following situations:

- A writer thread attempts to acquire the lock already owned either by another writer, or by one or more threads in shared mode.
- A reader thread attempts to acquire the lock that is owned exclusively.
- A reader thread attempts to acquire the lock that is owned in shared access by other readers, and one or more writer threads are waiting for exclusive ownership. This case prevents starvation of the writer threads.

In such cases, the thread inserts its wait record to the end of the list of waiters, releases the critical section and waits on the keyed event. The thread specifies address of its wait record as a key value.

A thread wait record is a small structure that contains information about the mode the thread wants to acquire the lock for. Every thread owns one such record. This is sufficient because every thread can wait for at most one of our reader-writer locks at any time. The library maintains a mapping between TIDs and corresponding wait records.

Thread wait records are allocated neither during reader-writer lock creation, nor during its acquisition or release. The allocation takes place at the following moments:

- During the initialization phase, the library enumerates all threads of the current process and creates thread wait records for them.
- If the thread is created after the initialization of the library, its wait record is allocated in response to `DLL_THREAD_ATTACH` callback.

Deallocation of the wait records follows similar rules. If a thread exits, its record is deleted as a part of execution of the `DLL_THREAD_DETACH` callback. Wait records of other threads are deleted during the library cleanup.

Release

The first step of the release operation is very similar to lock acquisition. The thread acquires the critical section and updates the lock structure to reflect its new state (decrements number of readers, sets the exclusivity flag to zero). Then, it wakes one or more waiters and releases the critical section. The number and the type of resumed waiters depend on the type of the release operation. There are two resume scenarios:

- If the operation is performed by a reader thread, only the first waiter is removed from the list and resumed. The waiter must be a writer because no reader threads are blocked until there are any writers waiting for the lock.
- If the thread owns the lock exclusively and the first wait record in the list of waiters represents another writer, the scenario is very similar to the case described in the previous point. The record is removed from the list and the lock ownership is transferred directly to the thread described by the record. When the first entry in the list represents reader, the list is traversed from its head until the first wait record of writer thread is found. All preceeding entries are removed and their owners resumed.

Resuming a thread waiting on our reader-writer lock is done by performing a release operation on the keyed event. Address of the target wait record is used as a key value.

Characteristics of the Lock

From the above description and from the source code of the reader-writer lock implementation, it is possible to derive the following characteristics of the primitive:

- It works in low memory conditions. No memory allocation is required during the acquire and release operations.
- It cannot be acquired recursively.
- In case of no waiting readers and writers, acquire and release operations perform no system calls.
- It prefers neither readers, nor writers.

4.6 Changes Made to JPMF Framework

In theory, integration of our data source to JPMF should not be very complicated task. The framework attempts to load and initialize all data source classes using `ServiceLoader` approach. So it seemed that adding full name of our data source class to the list in the `META-INF` file will do the job. However, the reality showed to be a little bit different.

The main problem was that JPMF contained no support for sensors of decimal type. The framework supported only sensor with 32-bit or 64-bit integer values.

Decimal type of sensors is crucial for Windows Performance Objects because this interface uses performance counters of this type quite often. Integration of the new sensor value type was required.

At first, we created a new class named `DoubleValueHandle` and placed it into `src\jpda\main\org\ow2\dsrg\jpda\DoubleValueHandle.java` file in the source code tree. The class implements `ValueHandle` interface, as another value types do, and is very similar to the `LongValueHandle` and `IntValueHandle` classes. Classes that implement this interface serves as storage for performance data of certain type; `Intvaluehandle` for 32-bit integers, `LongValueHandle` for 64-bit integers and `DoubleValueHandle` for decimal numbers.

We had to update `ValueHandle` interface itself to reflect addition of the new value type. The interface forces classes, that implement it, to contain methods that allow converting values of one type to another, whenever possible. For example, the interface contains `longValue` method, which purpose is to convert the value inside the class to 64-bit integer. This method makes sense in context of `IntValueHandle` because every 32-bit integer value can be easily and loselessly converted into the 64-bit integer. The method makes much less sense for the `DoubleValueHandle` class. In such case, it throws an exception, instead of returning the converted value. A `doubleValue` method was added to `ValueHandle` interface to allow conversion of values to the newly supported type. Update of the interface requested to implement the method in all implementing classes.

After these changes, time came to add decimal type into the `ValueType` enumeration. The new field has been added in a way similar to the already present members, in order not to destroy mechanisms of JPMF, working with sensors independently of their type.

Another update has been made to the `assingStorage` method of the `SensorReading` class. The class provides read-only access to sensor, their instances and data. The method is responsible for creating storage which is then passed to one of the classes that implement `ValueHandle` interface (`IntValueHandle`, `LongValueHandle` and `DoubleValueHandle`), and used to store decoded sensor values.

The last change was made to the `__AllocateStorage` method of the `BaseMeasurementContext` class. The task of the method is to determine the type of sensors that the user of the interface wishes to measure, and allocate storage of appropriate value types for them.

To make these changes easily traceable, newly added lines of code were enclosed with the following comments:

```
// CHANGE-MARTIN-DRAB
//\CHANGE-MARTIN-DRAB
```

5. Conclusion

We have successfully implemented a new data source for JPMF. The module is able to collect performance data through the Windows Performance Objects registry interface. The data source should work well when measuring larger number of performance counters, however, it also provides a way to access the single counters.

5.1 Registry Interface Abstraction, Caching and JNI Bindings

We have built an abstraction over the registry interface, that makes it more suitable for use in JPMF. The abstraction allows to consumer performance data at level of performance counters, which reduces amount of data copy operations. The abstraction is implemented in standalone dynamic link library that is connected to JPMF via Java Native Interface.

The main requirement put on the implementation of the data source was to minimize requests made to Windows Performance Objects interface. The interface works only at level of performance objects, not single counters or instances. Such behaviour implies copying of potentially large data structures even in case, the user is interested only in a value of one performance counter. The effect of this unwanted behaviour was reduced by implementing a cache that stores copies of recently collected performance data. The cache is then used to handle requests originally designed for Windows Performance Objects interface during sampling (data collection) phase of the measurement. The cache helps only in case, when more entities attempt to obtain performance data for the same performance objects in short period of time. It has no use when the registry interface is being accessed only by a single entity.

Because the library, that provides abstractions over Windows Performance Objects interface, had to be written in different programming language than Java, which is the one used to implement JPMF, it was also required to create appropriate bindings between these subjects. It was decided to divide their responsibility in the following manner: the library (written in C++) should solve problems related to performance data measurement at single counter granularity, and Java part of the data source should increase the access granularity to the level of counter instances. Implementation of the decision showed to be quite straightforward and reduced amount of data being copied from the scope of the library to JPMF.

5.2 The Instance Naming Problem

Another problem arised with unique identification of counter instances. Some performance objects, i.e. the Process object that reports statistics related to running processes, do not name their counter instances in the way, that suits our needs, especially in the area of unique identification. The Process object names its instances by image names of corresponding running processes. Such

names do not provide unique identification of the instances. We decided to solve this problem by renaming these instances to PIDs of the running processes. The operation is performed in the scope of JPMF.

The method showed to be quite reliable in case of the running processes. Problems arised for the Thread performance object, which task is to report statistics about running threads. Because threads are created and terminated quite often (several thread creations and exists per second may happen), the number of counter instances of the performance object varies frequently, which makes our method quite unreliable in this case. For this reasons, we used the counter instance renaming method only in case of the Process object.

General solution to the counter instance changing problem is to build an ability of the realtime changing of number and names of the counter instances into JPMF. The data sources should inform the framework of such changes and it should reflect them into the structure of the performance data.

However, this goal is not really easy. Even detection of the changes in counter instances might be nontrivial, if we do not wish to always traverse the while snapshot of performance data and compare it with the previous one, which might be quite time consuming. To demonstrate the truth of this statement, we can examine the ways how changes of the running processes and threads can be detected.

Probably the easiest way is to scan the list of the running processes and threads periodically and detect the changes by comparison of two most recent snapshot. This approach, however, might be quite time consuming, especially in case of the running threads, because of their quantity. And it does not guarantee to capture all the changes. Another, and quite elegant, documented and portable way is to register callback routines that the operating system invokes when any change of the running processes and threads occurs. The main problem of this solution lies in the fact that such callbacks might be registered by kernel drivers only, which implies ownership of high privileges to the system under test, and a digital certificate when used on 64-bit version of Windows.

Our data source is, to certain extent, able to resist realtime changes in counter instances. However, it is not able to measure performance data for instances, created after its initialization, and might assign the performance data to the incorrect instances.

5.3 The Reader-Writer Lock

We have successfully implemented and tested our own implementation of the reader-writer lock synchronization primitive. The primitive is intended to be used on Windows XP where, as we thought for several years, is no such primitive available to usermode applications. Our reader-writer lock benefits of the critical section primitive, which makes the lock to behave nearly as a spin lock, in certain scenarios. The lock implementation uses keyed event synchronization primitive to implement the unloack-and-wait atomic operation.

However, the recent research showed our claims about absence of the reader-writer lock primitive for applications on Windows XP as false. The existence of a new primitive, called user-mode resource, has been discovered. This primitive

is a port of one kind of the kernelmode reader-writer locks into userspace, and is present not only on Windows XP, but it existed already on Windows NT 3.1.

After taking all things into account, we decided not to use our reader-writer locks to synchronize accesses to our Windows Performance Object interface abstraction, that are made from JPMF. The JPMF part of the data source uses primitives available for the Java programming language. Our locks are used mainly to synchronize data structures of the performance object cache, and are ready to be used in case, the library will be used as a standalone component in various environments.

5.4 Future Work

There are several directions of further improvements on this topic. Possibly, the mechanism of performance data caching can be enhanced. Standalone Windows Performance Objects interface should be tested against other alternatives, especially our library and Performance Data Helper library, in means of speed and influence on the system under test. This comparison would show the real efficiency of our implementation.

Although the library can be also used on its own (without JPMF), it provides no comfortable way of access to individual counter instances because this feature is implemented as part of the data source module in Java. If someone wishes to use the library in his or her own application, this feature must be ported to that environment, which may trigger some changes of abstraction made over the registry interface.

The library performs caching at the level of raw performance data. It could be interesting to make an attempt to cache at other places of the library and data source module, and determine which of such approaches (or their combinations) is more effective than the current one. For example, it is possible to cache calculated counter values, or even place the cache into low level code of JPMF. This would save some trips through JNI. The question is, how much the suggested approaches would be effective for performance counters which value changes dramatically in short period of time.

References

[1] Bulej Lubomír. *Connector-based Performance Data Collection for Component Applications*. issertation Thesis, Dept. of SW Engineering, Charles University, Prague, September. 2007.

[2] Shende S. and Malony A. D. *The TAU Parallel Performance System*. Intl. Journal of High Performance Computing Applications, 20(2):287-331, SAGE Publications, 2006.

[3] Russinovich M., Solomon D., Inoscu A. *Windows Internals Covering Windows Server 2008 and Windows Vista*. 5. edition Microsoft Press, 2009. ISBN 0735625301.

[4] Russinovich M., Solomon D., Inoscu A. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. 6. edition Microsoft Press, 2012. ISBN 0735648735.

[5] Duffy J. *Concurrent Programming on Windows*. Addison-Wesley Professional, 2008. ISBN 032143482X

[6] Tůma P. and Bulej L. *Xampler: Application for Detailed Benchmarking of Middleware*. http://dsrg.mff.cuni.cz/ccpsuite, Distributed Systems Research Group, Charles University, December 2006.

[7] Performance Research Lab. *University of Oregon. TAU: Tuning and Analysis Utilities*. http://www.cs.uoregon.edu/research/tau.

[8] *Performance Data Format* [online], available at http://msdn.microsoft.com/en-us/library/windows/desktop/aa373105%28v=vs.85%29.aspx. [cited 3. 12. 2011].

[9] *Using the Registry Functions to Consume Counter Data* [online], available at http://msdn.microsoft.com/en-us/library/windows/desktop/aa373219%28v=vs.85%29.aspx, [cited 3. 12. 2011]

[10] *Retrieving Counter Names and Help Text* [online], available at http://msdn.microsoft.com/en-us/library/windows/desktop/aa373181%28v=vs.85%29.aspx [cited 3. 12. 2011]

List of Tables

List of Figures