Deliverable D3.1

Requirements for the Interface between System-level and Application-level Performance Analysis

CONTRACT NO	HOPSA-EU 277463
INSTRUMENT	CP (Collaborative project)
CALL	FP7-ICT-2011-EU-Russia

Due date of deliverable: August 1st, 2011

Actual submission date: August 15th, 2011

Start date of project: 1 FEBRUARY 2011 Duration: 24 months

Name of lead contractor for this deliverable: Felix Wolf, GRS

Abstract: This report provides the requirements for the interface between system-level and application level performance analysis. It documents the understanding of the interface between both parts by all members of the project including our Russian project partners.

Project co-funded by the European Commission within the Seventh Framework Programme (FP7/2007-2013)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	X
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Table of Contents

Glossary

Abbreviation / acronym	Description
API	Application Programming Interface
GUI	Graphical User Interface
HOPSA	HOlistic Performance System Analysis
HPC	High Performance Computing
I/O	Input/Output
MPI	Message Passing Interface (Programming Model for Distributed Memory Systems)
OpenMP	Open Multi-Processing (Programming Model for Shared Memory Systems)
REST	Representational State Transfer (Inter-process Communication Protocol)

1. Executive summary

The objective of this work package is to combine and integrate the work on the HPC system-level performance and on application-level performance into a coherent and holistic performance analysis environment. This environment will provide a performance report compiling essential information from system-level monitoring and application-centric measurements. When an application is found to have performance problems, an automated workflow guides system administrators and application developers in conducting more detailed analysis using a variety of mature performance tools. These tools also take advantage of the tight integration, correlating data from both system-level and application-level sources.

As an important prerequisite, this deliverable provides the formal requirements of an interface enabling the exchange of performance-related results between the system-level, job-level, and low-level application analysis on the one hand and high-level performance tools on the other hand. More precisely, this deliverable

- identifies the key performance metrics which should be maintained in a system performance database after job completion,
- defines the requirements of the interface to interchange these metrics between the system-level, job-level, and low-level application analysis on the one hand and the high-level performance tools on the other hand, and
- outlines the design of a low-overhead end-to-end performance analysis for all jobs running on a given system from their submission to their completion.

2. Introduction

To maximize the scientific output of a high-performance computing system, different stakeholders pursue different strategies. While individual application developers are trying to shorten the time to solution by optimizing their codes, system administrators are tuning the configuration of the overall system to increase its throughput. Yet, the complexity of today's machines with their strong interrelationship between application and system performance presents serious challenges to achieving these goals with non-correlated *application* and *system-level tuning* processes.

One of the goals of the HOPSA project is to close this gap and connect application and system-level tuning by collecting system-level and application-level performance metrics and making them available to both sides. At the interface between these two facets of our holistic approach, which is illustrated in Figure 1, is the system-wide performance screening of individual jobs, pointing both at system-related performance issues such as above-average waiting time in the queue and at inefficiencies of individual applications such as high communication overhead. Once the screening pinpoints an application for more detailed analysis, system administrators and application developers are provided with hints on how to track down the source of the inefficiency effectively using our set of mature performance tools. These tools can provide an enriched view of the application performance by correlating data from both sources, e.g. imbalance in file I/O time (application-level data) caused by heavy load on the I/O system by other jobs at the same time (system-level data). In this deliverable, we summarize the formal requirements of this interface enabling the exchange of performance-related results between the system level, job-level, and low-level application analysis on the one hand and high-level performance tools on the other hand.

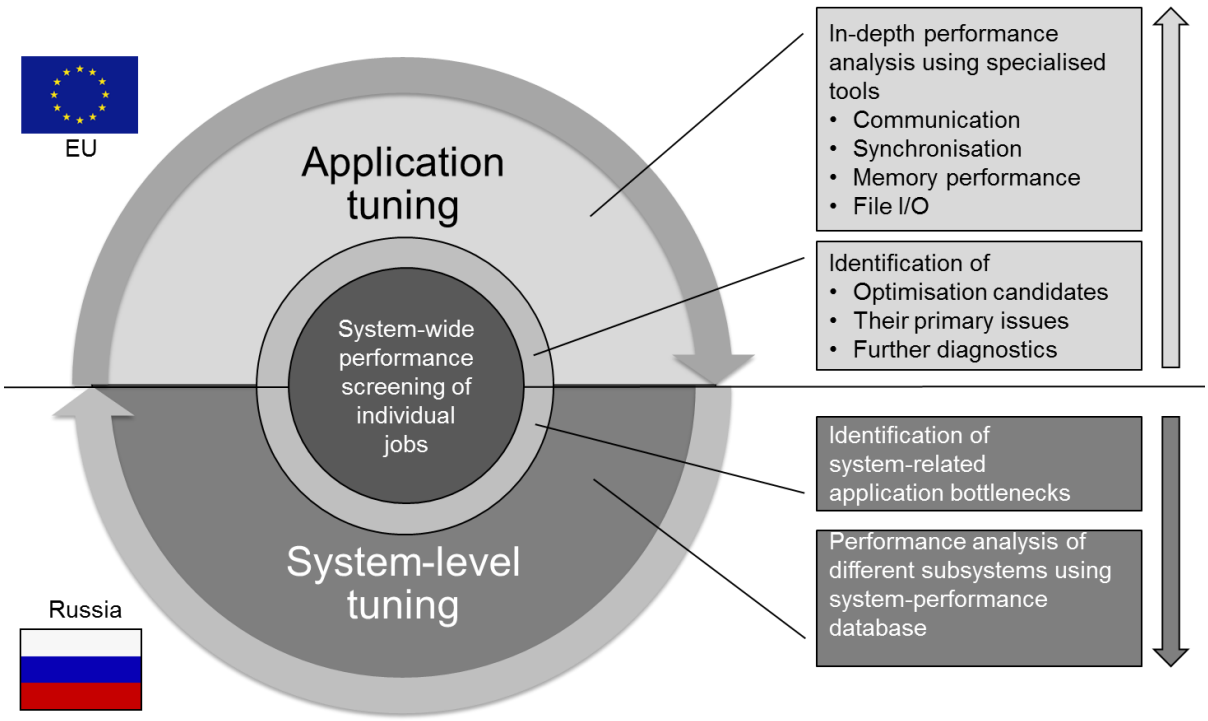

Figure 1: System- level tuning (bottom), application-level tuning (top), and system-wide performance screening (center) use common interface for exchanging performance properties.

3. Main section

3.1 Technical interaction of systems and tools

The following sections explain in which way system-level metrics are being collected and how they are made available to be used by the application-centric performance tools in order to enrich their presentation of performance data to the user.

3.1.1 System-level monitoring

System-level tuning tries to identify system-related bottlenecks, such as network problems (congestion, driver issues, etc.) or hardware failures, which could potentially slow down an application. On a more technical level, these bottlenecks can be assessed through node-level and system-level metrics. The Russian partners are developing a system to collect and aggregate both kinds of metrics in a system-performance database. The overall system consists of two parts, HOPSA-I refers to *data collection* and HOPSA-II refers to *analysis*. Figure 2 shows the general architecture of the system. **Agents**, **aggregation layer**, and **agent modules** together form the data collection part and the other components belong to the analysis part.

Design of HOPSA-I – Data collection

The agents in the system are low-overhead processes running on the individual nodes. They collect different kind of metrics and pass this data to the aggregation layer. **Node-level metrics**, as listed in Table 1, will be collected by agents using *ClustrX Watch*, which is part of the *ClustrX* cluster-monitoring suite developed by T-Platforms [5]. These kinds of agents interface with various modules collecting sensor information. Among the metrics measured by the sensors are hardware performance counters for all relevant subsystems in a particular node.

Another type of agents, interfacing with the batch system itself, provides further **system-level metrics**, as listed in Table 2. This includes metrics for system components that cannot host a monitoring agent themselves, but can be queried, e.g. Network switches, SAN front-ends, RAID controllers, power meters, room climate information. Again, these are also sent to the aggregation layer from where the metrics are made accessible to the system. This aggregation layer combines the metrics collected by the different agents and provides some first, simple analysis, for example, to check the range of the values in the aggregated metrics. For different data collectors, or different locations (different clusters) the layer can be made up of several processes. It is worth noting that additional metrics might be identified in the next steps.

Design of HOPSA-II - Analysis

The logic part of the system is shown in Figure 2 in the upper part of the diagram. The **control server** keeps information about the system configuration, i.e. available modules, agents, connection topology etc. To accomplish this, it connects to the aggregation layer and retrieves information about all sensors and agents. For fault tolerance and performance reasons there can be more than one control server in the system. It is also responsible for managing all modules in the system, i.e. starting when needed as well as terminating them if not needed anymore. Modules encapsulate functionality for storing the metrics (database modules), performing some kind of analysis (analysis modules), or visualizing data for some user.

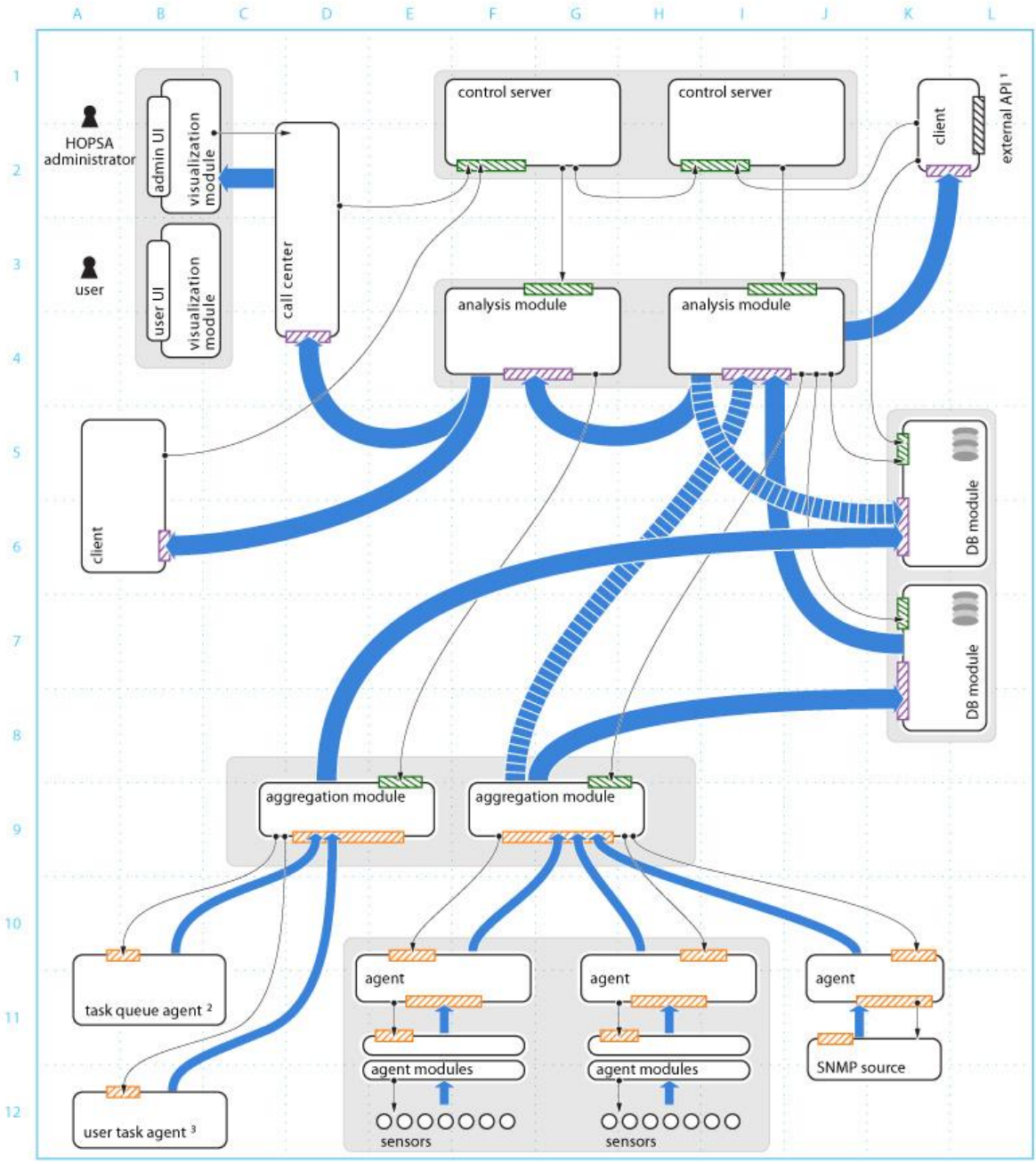

Figure 2. Architectural overview of the cluster-wide measurement system. Green rectangles denote the Control API while purple rectangles denote the API Consumer-Agent.

While one aspect of the system is the collection and storing of these metrics, it is also possible to perform post-mortem analysis, especially historical analysis to allow investigating job related properties over time beyond a single execution. In order to process large amounts of data occurring in large-scale high-performance computing installations, special tools are required. Currently, special data-processing definition language, HOPLANG, is being developed. The main goal of this language is scalable processing large amounts of data, which can be obtained from several data sources, even in the scope of a single query. Such data sources include, among others, SQL databases, LDAP bases, log files, custom databases, distributed databases such as Cassandra [6] and direct monitoring data. Several instances of ordinary SQL databases may be joined into a single data source, in order to increase performance and scalability of data processing.

HOPLANG is aimed to be scalable, and to process data quickly, so that it would be possible to get fast response for queries which process small amounts of data.

Table 1: Node-level metrics.

Identifier	Alias name	Identifier	Alias name
1010	MON_CPU_TEMP	1234	MON_V_1_4
1018	MON_CPU_FREQ	1235	MON_V_1_5
1020	MON_CPU_FAN	1236	MON_V_3_3VSB
1030	MON_CPU_VCORE	1237	MON_V_5VSB
1050	MON_CPU_USAGE_USER	1238	MON_V_BAT
1051	MON_CPU_USAGE_NICE	1239	MON_V_1_1
1052	MON_CPU_USAGE_SYSTEM	1240	MON_V_1_8
1053	MON_CPU_USAGE_IDLE	1241	MON_V_N12
1054	MON_CPU_USAGE_IOWAIT	1280	MON_A_INSTANT
1055	MON_CPU_USAGE_IRQ	1281	MON_W_INSTANT
1056	MON_CPU_USAGE_SOFTIRQ	1250	MON_MEMORY_TOTAL
1080	MON_CPU_MCE_TOTAL	1251	MON_MEMORY_VMALLOC
1130	MON_V_MEM	1252	MON_MEMORY_SWAP_TOTAL
1210	MON_SYS_TEMP	1253	MON_MEMORY_SWAP_FREE
1211	MON_MEM_TEMP	1254	MON_MEMORY_TOTAL_FREE
1220	MON_SYS_FAN	1300	MON_PS_INP_VOLTS
1225	MON_CHASSIS_FAN	1303	MON_PS_INP_WATTS
1230	MON_V_3_3	1310	MON_PS_OUTP_VOLTS
1231	MON_V_5	1314	MON_PS_OUTP_LOAD
1232	MON_V_12	1320	MON_PS_TEMP
1233	MON_V_1_2	1330	MON_PS_FAN

These metrics – both online and post-mortem – are made available in different ways. One way of accessing the data will be using well-known and proven interface technologies, for example, using a REST-style web service. Especially for the performance tools in the HOPSA project, however, there will also be a C API provided to access the metrics in an efficient manner. It is particularly important that the C API has a minimal impact on the client, i.e., not blocking the control flow, not requiring extra threads, or high memory consumption.

Table 2: System-level metrics.

Identifier	Alias name
2000	MON_RX_PACKETS
2001	MON_TX_PACKETS
2002	MON_RX_BYTES
2003	MON_TX_BYTES
2004	MON_RX_ERRORS
2005	MON_TX_ERRORS
2006	MON_RX_DROPPED
2007	MON_TX_DROPPED
2008	MON_MULTICAST
2009	MON_COLLISIONS
2100	MON_FS_BYTES_USAGE
2101	MON_FS_INODES_USAGE

3.1.2 Augmented application-centric measurement

Application tuning has been performed using specialized tools which measure, for example, communication patterns of distributed, parallel programs; memory performance and memory access patterns; as well as how processes utilize file I/O. With these measurements, many application-caused performance problems can be identified, diagnosed and presented to the user. Unfortunately, these metrics only look at the operations an application performs itself. Based on the system-level monitoring system, however, the holistic performance analysis environment will provide a performance report including not only application-centric metrics but also system-level metrics. Such a report will give an overview about the essential performance properties, for instance, by visualizing the runtime behavior in time-line browsers such as Vampir [3] and Paraver [2].

To give examples of the current capabilities of these tools and to exemplify potential visualization options of system-specific performance metrics, Figures 3 and 4 show traces displayed in time lines using the Paraver tool. Figure 3 compares the amount of memory accesses in computation regions before and after optimization in the Gromacs molecular dynamics code, while Figure 4 shows the level 1 data cache misses of the same application in the same interval. The original version shows more variability and higher (dark blue) values in both metrics, with the optimized version showing smoother patterns and lower (bright green) values. Figure 5 shows a different view of the same traces in Paraver, a histogram displaying the level 1 data cache misses. Here the wide dispersion of values in the original traces shows suboptimal performance, with the optimized version showing much less variability.

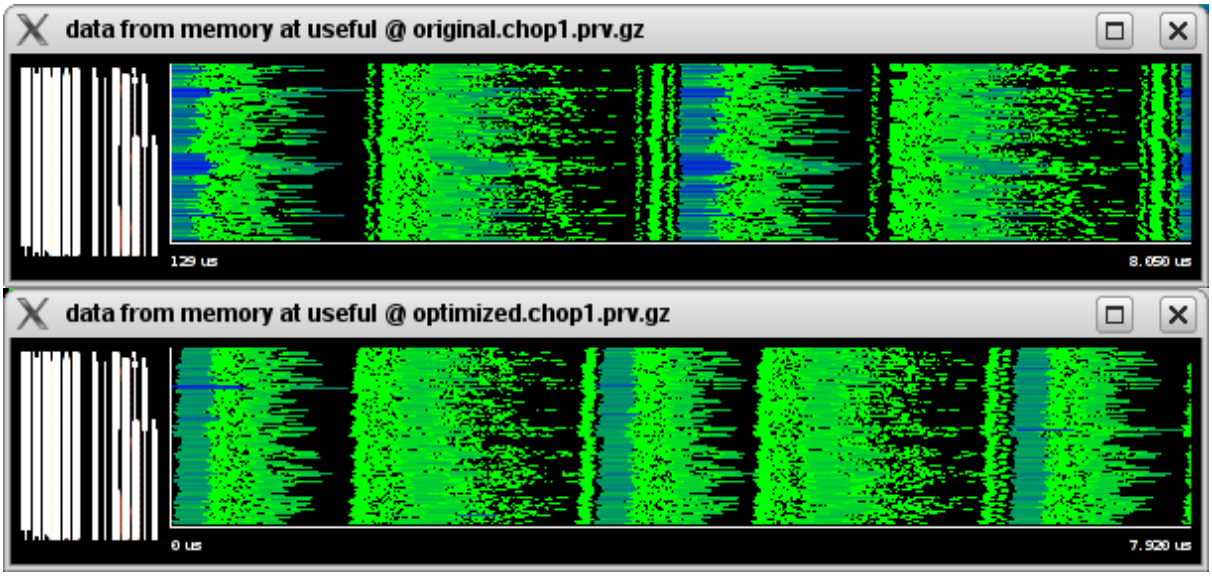

Figure 3: Paraver displaying traces collected from the Gromacs code before (top) and after (bottom) optimization. The metric displayed shows the amount of data loaded from main memory during computation regions.

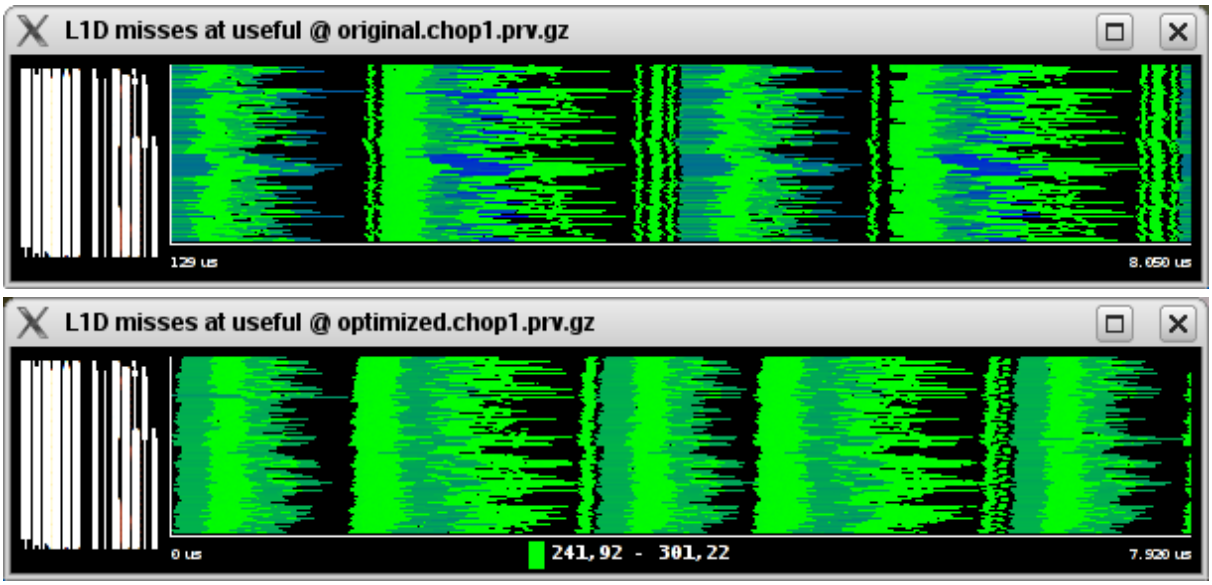

Figure 4: Paraver displaying traces collected from the Gromacs code before (top) and after (bottom) optimization. The metric displayed is the level 1 data cache misses during computation regions.

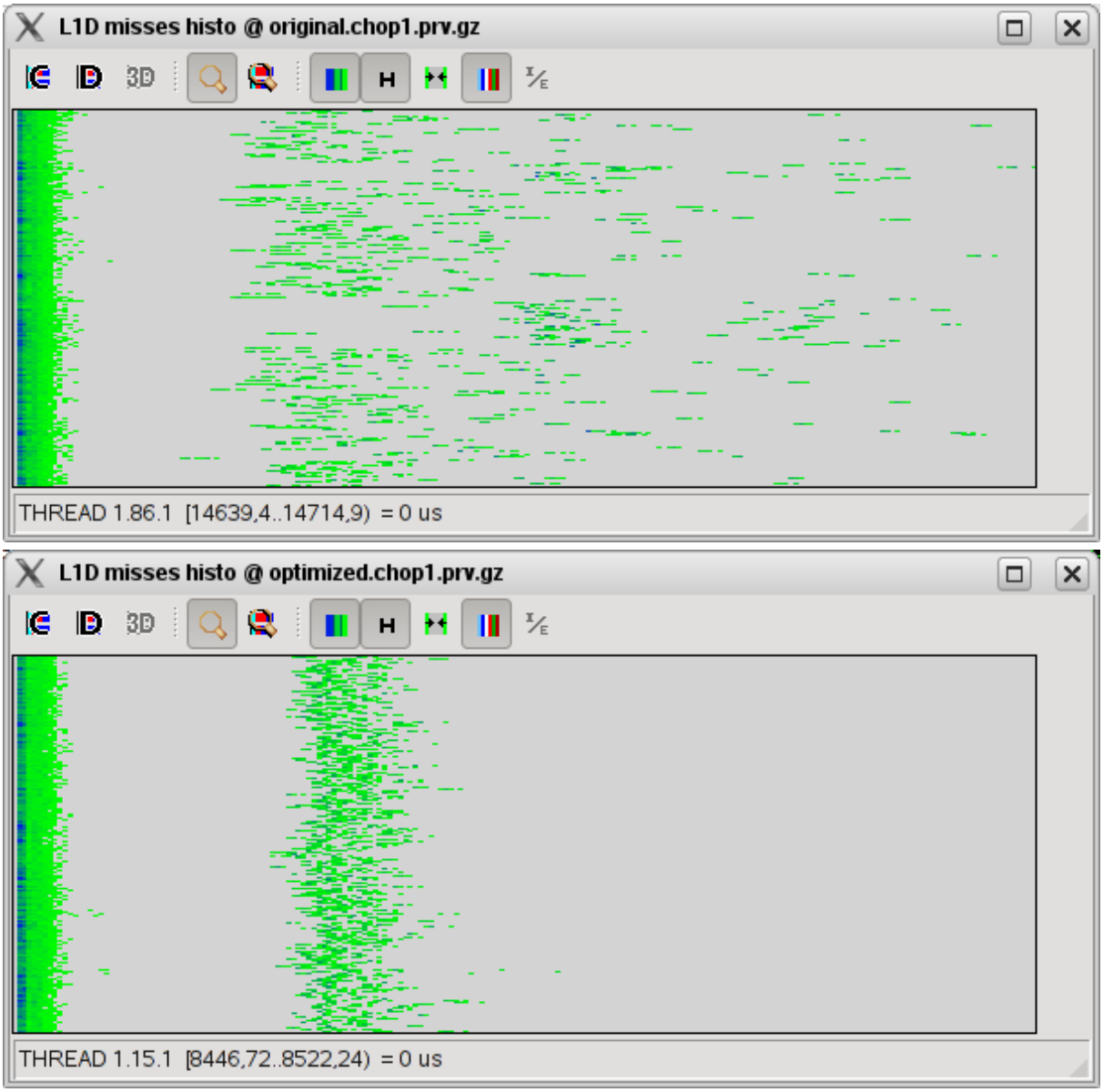

Figure 5: Paraver displaying a histogram view of traces collected from the Gromacs code before (top) and after (bottom) optimization. Displayed are level 1 data cache misses.

Figure 6 gives an example of displaying traces of the NPB BT benchmark in Vampir, showing a variety of different views. In the top left it shows an overview time line of the different processes with computation in yellow and communication in red. In the bottom left time lines of two hardware counter values are displayed, number of floating-point operations and level 2 cache misses. In the lower right corner a communication matrix is included, showing the amount of data transmitted between each pair of processes. In HOPSA, these same displays can be used to display system-level data collected and provided by HOPSA-I and HOPSA-II, for example network or I/O system load, in parallel with the usual application-level metrics. Correlation both data sources will give developers new insight into their applications. In addition to visualizing trace data, the performance report shall include hints about potential performance deficiencies and how to verify their presence with other tools, for instance, by applying Scalasca [1] or ThreadSpotter [4]. Again to give examples of the current capabilities of these tools and to exemplify potential visualization options of system-specific performance metrics, Figure 7 shows an example of a trace displayed in CUBE, the GUI component of Scalasca. After automatically analyzing the traces to identify patterns of inefficient communications behavior, the time wasted in the "Late Sender" inefficiency pattern has been separated from the actual message transfer part of the point-to-point communication time. The pane on the right shows the distribution of this useful communication time on an important call path among all the processes of the communication.

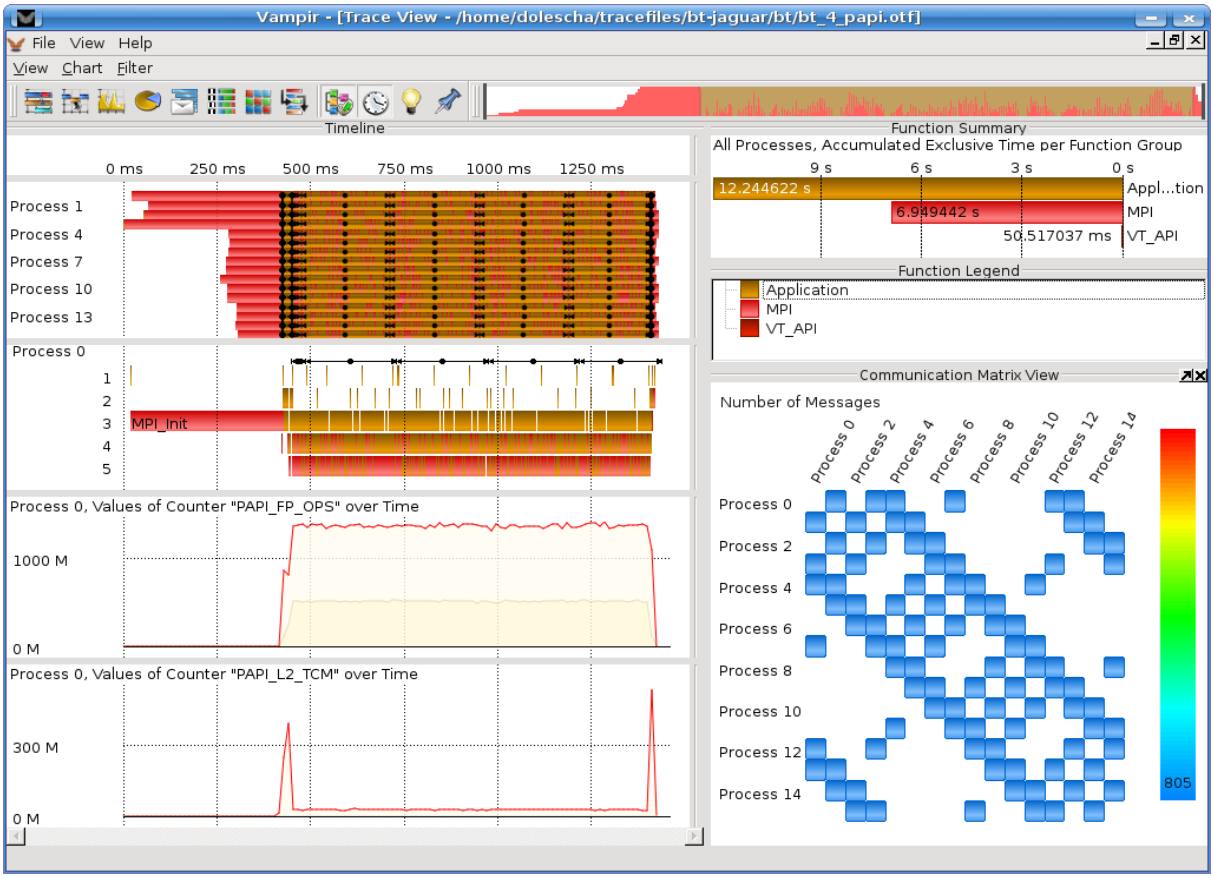

Figure 6: Vampir displaying traces collected from the NPB BT benchmark, showing a variety of different views of the same execution.

Furthermore, other than displaying traces in time lines and doing automatic analysis looking for higher-level inefficiency patterns, the report should also allow comparing performance behavior with past reports of the same application to survey the success of analysis and tuning.

Of course, these performance tools require certain functions of the system-level monitoring system. The first is retrieving a hierarchical inventory of available metrics and their locations. This inventory will contain all available metrics using a consistent naming scheme. The metrics will be associated with a set of system components (locations) e.g. CPU core, CPU socket, node, rack, switch, SAN component or power supply unit. Furthermore, details about the individual metrics will be provided. These details will include a description of the metric's unit, a specification of the mode, depicting whether the metric is an absolute value or accumulated over a period of time, and the time scope, i.e., if the sample is valid (since the last sample, since the very beginning of the measurement, until the next sample, or only right now). For metrics with a fixed sampling interval this is also supplied.

Performance data is accessed by specifying the metric and a location. For general access, different modes are required: synchronous, where the client fetches the momentary or most recent value for the local entity, asynchronous, where the system sends updated samples to the client, and post-mortem, where the performance tool requests a sequence of past samples of metrics for a given time period. For the latter case, it is assumed that the metrics are constantly stored in the system's database module. If this will not be the case, an additional mechanism to start and stop the storage of metrics is required.

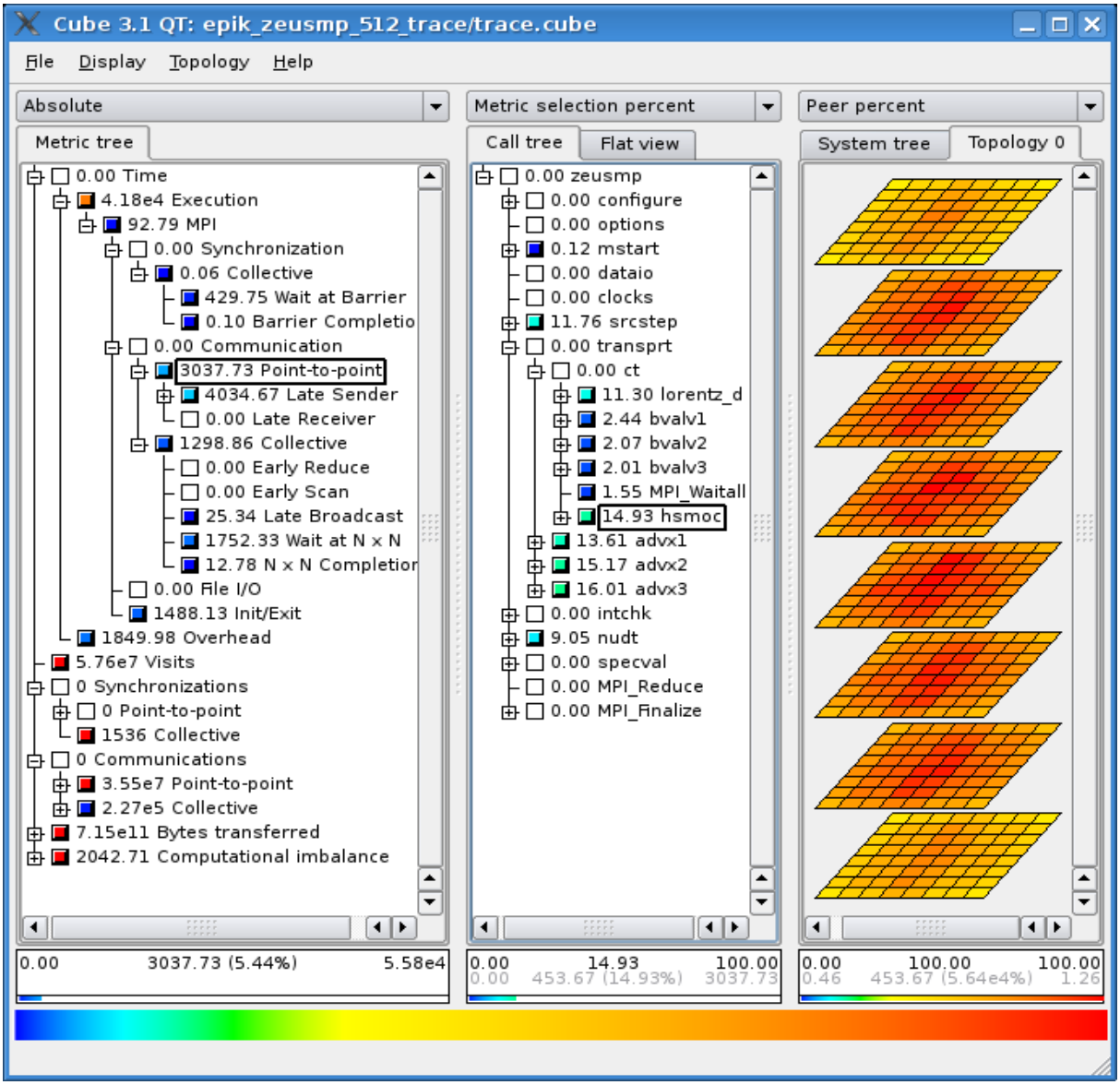

Figure 7: Cube, the GUI component of Scalasca displaying a trace of the 132.zeusmp2 benchmark from the SPEC MPI 2007 benchmark suite.

Moreover, it should be possible to allow the performance tools to specify a combination of metrics, i.e. arithmetic operations with different metrics or to receive aggregated metrics, i.e. the minimum/maximum/average of a set of metrics. Another important point is adequate **time synchronization** between all locations and monitoring clients in order to accurately attribute metrics to tracing events especially in the case of post-mortem retrieval of metrics for a particular time period.

3.2 Lightweight application performance monitoring module

As the goal of the project is to define a comprehensive performance measurement and analysis methodology, a lightweight parallel performance collection module, capturing basic performance metrics such as execution time or message-passing metrics, is necessary to cover an end-to-end performance analysis from job submission to completion. This module will be implemented as a shared library, so that it can be preloaded before the execution of a parallel job by the job launcher (or linked statically using a compiler wrapper if dynamic linking is not available).

More precisely, the lightweight module will run in parallel to applications on the cluster and collect basic performance information about the running application. It will not only enable the basic end-to-end performance analysis of potential performance bottlenecks but will also provide the general system state along with information about the scheduling process a given job went through. The end-to-end performance analyses will be mostly performed using measurements gathered by the module itself. Measured metrics include

- the message-passing behavior of distributed applications,
- the runtime profile of multi-threaded codes (OpenMP, pthreads), and
- memory access.

Further information about the general system state and about the handling of individual jobs will be retrieved from the system-level metric interface discussed above. This enables the module to inform the user about the time his job spent in a batch queue, potentially why it spent time there, and give reasons for certain scheduling decisions. In addition, it will be possible to notify the user about certain system events affecting running jobs. For example, if a job using message passing fails because of a physical network interruption between nodes the user will be informed that it was an external problem and not necessarily a problem with the application itself.

Light-weight module architecture

The module itself is composed of several sub-modules, which enables a user to select on a fine-granular level which metrics shall be collected during an application's run or which analysis shall be performed post-mortem. A schematic overview of the architecture is shown in Figure 8.

Measurement modules are responsible for measuring and collecting information about the application's runtime behavior. With the main requirements being maintaining low-overhead and having as less influence on the application itself as possible, statistical sampling is used instead of, for example, event tracing. This results in a lot less measurement points (*signals*), which have to be collected, stored, and managed.

Information about the usage of libraries is collected using a combination of instrumentation and sampling. In the case of the message passing behavior of an application, signals are collected using an MPI wrapper, which intercepts function calls to MPI functions using the PMPI profiling interface. In these instrumented calls a flag is set when the intercepted function is currently active. A so-called *signal handler* is called repeatedly after a certain amount of time by the operating system, and – if an instrumented function is active during the invocation of the signal handler – creates a signal indication that the instrumented function was active at a certain point of time.

In a similar way information about the behavior of multi-threaded applications is collected using interception of certain functions of the *pthreads* threading library, which is available on most high-performance computing platforms. In most cases it is also the underlying library of implementations of the *OpenMP* standard, a widely used shared-memory parallelization paradigm, thus allowing the module to collect metrics about applications using OpenMP as well.

If the high-performance computing system supports it, hardware counter information can also be collected, for example, to allow the module to gain insights into the memory usage pattern of the application. Hardware performance counters can either be collected using the cross-platform PAPI library, retrieved from the system-wide measurement system, or other libraries/interfaces can easily be attached to the monitoring module due to well-defined interfaces.

All measurement modules store their collected signals in an aggregated form in a central **data repository**. This repository stores all similar signals in a compressed form, which allows for efficient storage and fast access times.

Analysis modules operate just before the end of an application and mine the data collected by the measurement modules in order to detect (potential) performance problems. In addition they provide advice and guidance to the user on how to proceed in the performance analysis with more advanced performance analysis tools.

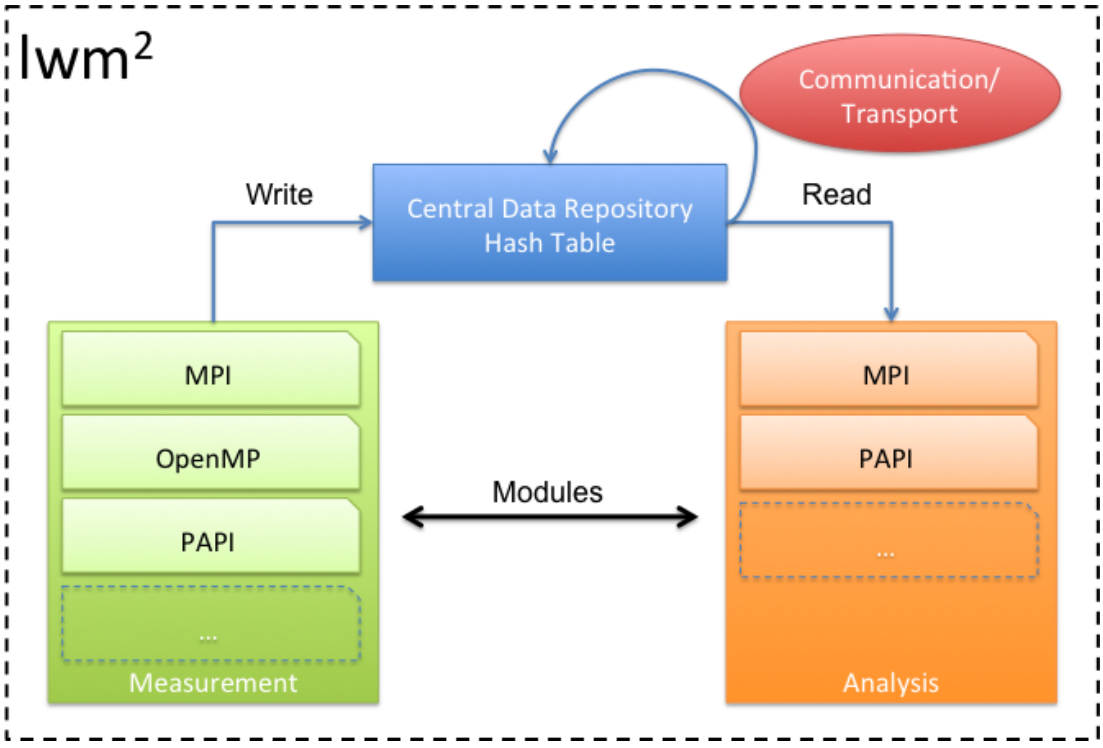

Figure 8: General architecture of the light-weight monitoring module.

4. Conclusions

The HOPSA project creates an integrated diagnostic infrastructure for combined application and system tuning. Starting from system-wide basic performance screening of individual jobs, an automated workflow routes findings on potential bottlenecks either to application developers or system administrators with recommendations on how to identify their root cause using more powerful diagnostics. The objective of this work package is to combine and integrate the work on the HPC system-level performance and on application-level performance into a coherent and holistic performance analysis environment. As an important prerequisite, this deliverable provides the formal requirements of an interface enabling the exchange of performance-related results between the system-level, job-level, and low-level application analysis on the one hand and high-level performance tools on the other hand. On a more technical level, this deliverable (i) identifies the key performance metrics which should be maintained in a system performance database after job completion, (ii) defines the requirements of the interface to interchange these metrics between the system-level, job-level, and low-level application analysis on the one hand and the high-level performance tools on the other hand, and (iii) outlines the design of a low-overhead end-to-end performance analysis for all jobs running on a given system from their submission to their completion.

In future work, we will implement an interface with the described functionality. The interface designed in this way will enable a system-wide performance screening without exception which will distinguish the codes that utilise the underlying hardware well from those which do not and could therefore benefit from optimisation. Although many application performance problems can and should be addressed by the developer himself, for example, via re-coding relevant parts or replacing components with more efficient alternatives, some issues are in fact symptoms of a system-level bottleneck that may affect more than one application. Given that the tools being part of this project will also have a much wider sets of metrics available for their individual analysis, the tools will also be substantially enhanced, allowing the user to gain deeper insights into performance issues and, thus, to yield better optimisation results. In addition to the enhancement of individual tools, their effectiveness will also be promoted through closer integration.

5. Bibliography

[1] *M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.*

[2] *J. Gimenez, J. Labarta. Parallel Processing for Scientific Computing, chapter 2: Performance Analysis: From Art to Science. SIAM, 2006.*

[3] *A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, Tools for High Performance Computing, pages 139–155. Springer Verlag, July 2008.*

[4] *Rogue Wave Software AB. Acumem performance productivity tools. http://www.acumem.com.*

[5] *T-Platforms. Clustrx product family. http://www.t-platforms.com/products/software/clustrxproductfamily.html.*

[6] *Apache Foundation. Apache Cassandra http://cassandra.apache.org/*

