European Community Seventh Framework Programme
Theme FP7-ICT-2011-7
Computing Systems

Automatic Online Tuning (AutoTune)

D3.1
Extended Monitoring System

Robert Mijaković

Date of preparation (latest version): October 10th, 2012

PROJECT INFORMATION

Project acronym	AutoTune
Project full title	Automatic Online Tuning
Grant agreement no	288038
Call (part) identifier	FP7-ICT-2011-7
Funding scheme	Collaborative project

DOCUMENT INFORMATION

Deliverable Number	D3.1
Deliverable Name	Extended Monitoring System
Authors	Robert Mijaković (TUM), Martin Sandrieser (UNIVIE), Carmen Navarette (BADW-LRZ), Michael Gerndt (TUM)
Responsible Author	Robert Mijaković (TUM) e-mail: mijakovi@in.tum.de phone: +49 89 289 17677
Keywords	Periscope Tuning Framework, Monitoring
WP/Task	WP3 / Task 3.1

DISSEMINATION LEVEL

PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
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)	

COPYRIGHT NOTICES

ABSTRACT

Some of the tuning plugins currently under development in AutoTune for the Periscope Tuning Framework (PTF) require special performance and energy data. This document describes the extensions to the PTF monitor that were implemented in the first year of AutoTune. The extensions enable performance analysis of GPGPU codes written in OpenCL and CUDA, energy measurements, and performance measurements for parallel patterns-based applications.

This document starts with an overview of the relevant components of PTF. Next, the individual extensions are presented. It covers the provided metrics, the external components used in the implementation, and their integration into the PTF monitor. Each of these sections ends with a short demonstration of the measurement support. The final section gives a short summary and outlines future work.

TABLE OF CONTENTS

1 Introduction

During the last few years supercomputer systems have entered the petascale range. These systems consume several megawatt of power and require powerful cooling systems to keep them cool and reliable. The constant increase of the price of electric energy has raised operational costs of these systems such that they now represent a significant fraction of the total cost. In order to achieve exascale computing, those systems must be significantly more energy efficient. This requires a careful tuning of applications as well as the introduction of new technologies. One promising way is to add very energy efficient accelerators to the systems, such as GPGPUs.

The AutoTune project develops the Periscope Tuning Framework (PTF) including several tuning plugins targeting performance improvement as well as reduced energy consumption of applications. To support the plugins with dynamic information, PTF has to monitor performance and energy data of the applications at runtime. The already existing monitoring system had to be extended for those measurements.

AutoTune supports programming of GPGPUs via HMPP generating automatically OpenCL and CUDA code as well as via manual coding. The manually coded kernels are used as components in high-level parallel patterns that are written in a novel programming model currently under development at University of Vienna. For both approaches, tuning plugins require performance data on the kernels executed. The extended PTF monitor allows to measure execution time of kernels, data transfer overhead between CPU and GPGPU, and provide hardware performance counter data for the executed GPGPU kernel. To further support analysis and tuning of high-level parallel patterns, additional information is collected from the pattern runtime library implemented in Vienna.

In addition, the extended PTF monitor provides energy measurements on HPC systems. Although the recently installed three Petaflop supercomputer (SuperMUC) at LRZ is the main target, the implementation can be ported to other systems if those provide the appropriate measurement support at hardware level.

This report is structured as follows. Section 2 introduces fundamental components of PTF's runtime monitoring. The third section presents the extensions of Periscope's monitoring infrastructure, i.e., a general refactoring of the monitor, performance measurement support for GPGPUs and high-level pattern programming, as well as energy measurement support. The final section gives a short summary and presents future work.

2 Periscope Tuning Framework

This section gives a short overview of the main components of the monitoring infrastructure of PTF. PTF is based on Periscope, a distributed automatic online performance analysis system developed by Technische Universität München (TUM) at the Chair of Computer Architecture (LRR). In contrast to most other performance analysis systems, Periscope executes an online analysis based on aggregated runtime information. It uses a hierarchy of analysis agents to search for performance bottlenecks in a distributed fashion. It is also capable of exploiting the application structure to automatically refine the discovered problems according to their code region and type.

2.1 Periscope architecture

PTF consists of a frontend and a hierarchy of communication and analysis agents, shown in Figure 1. Each of the analysis agents, i.e., the leaves of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes.

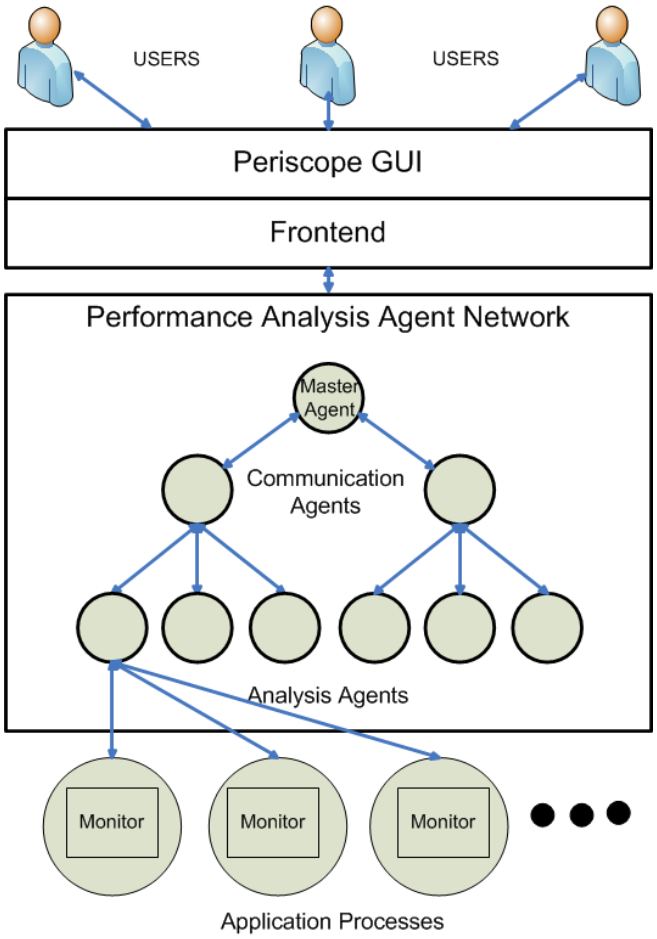

Figure 1: Architecture of Periscope

The root of the hierarchy is the master agent which takes commands from the frontend, propagates the commands to the analysis agents, and provides the list of found performance properties to the frontend. The agents in the middle of the hierarchy are responsible for forwarding information among the analysis agents and the master agent.

The search is controlled by the frontend. It invokes the application and creates the agent hierarchy. How many agents are created depends on the number of application processes and of additional processors provided in the batch job. The processes and agents are mapped to the available processors in a way that communication is local with respect to the physical interconnection network topology.

After the application and the agent hierarchy have been launched, the frontend starts the search by propagating a command to all the analysis agents. Many of Periscope's search strategies are multistep strategies, i.e., they consist of multiple search steps. Each search step evaluates performance properties via an experiment. If an agent requests a new experiment, the master agent starts a next experiment via another command.

The analysis agents dynamically configure the measurements and control the application's execution via the monitor which is linked to the application (Section 2.3). This configuration and control is performed through the Monitoring Request Interface (Section 2.2).

On top of the frontend, Periscope provides a graphical user interface based on Eclipse and the Parallel Tools Platform (PTP). The GUI allows the programmer to define a project with all the source files, start a performance analysis via the frontend and, most important, to investigate the performance properties found.

2.2 Monitoring Request Interface

The Monitoring Request Interface (MRI) defines the protocol used for the communication between the analysis agents and the monitor. The agents attach to the monitor via sockets. They send configuration and control requests to the monitor, and receive measured data via this socket connection.

The MRI provides configuration requests to trigger certain measurements by the monitor. These measurements specify the metric and the program region for which the metric should be measured. The additional control requests allow the agent to define a breakpoint at which the application processes are suspended in the monitor and to start or continue the application's execution. In addition to the configuration and control requests, MRI provides a request to retrieve the measurements.

2.3 PTF monitor

The monitor is highly configurable via MRI. It measures for program regions only the requested metrics. It configures hardware and software sensors and keeps track of measurements that were not possible due to measurement resource limitations. Together with the reported measurement data information about ignored requests is reported back to the analysis agent.

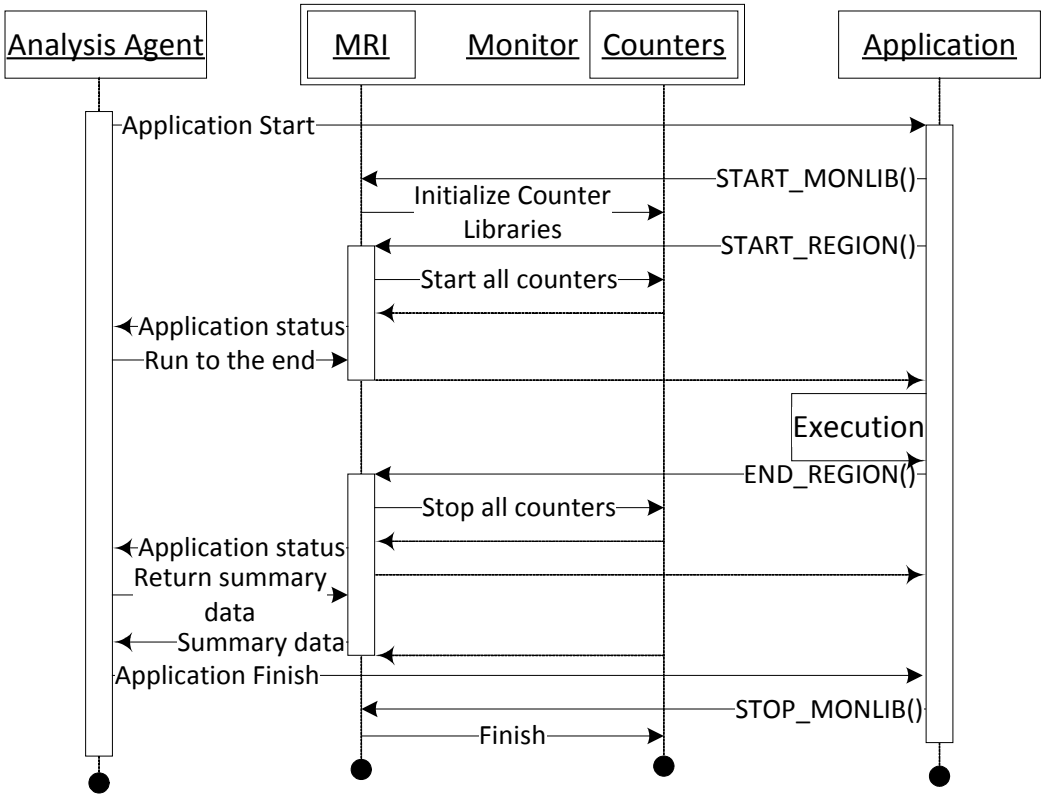

Figure 2: Sequence diagram presenting the interaction between the analysis agent, the PTF monitor and the application.

Figure 2 depicts the interaction between the analysis agents, the application and the monitoring system. The very first statement executed by the application after entering the main region is a call that initializes the monitor. The application processes are blocked in this call until Periscope specifies MRI requests and releases the application.

Once the tool sent all measurement requests, it specifies a breakpoint where the program should stop (usually the end of a region) and notifies the monitor to start the application's execution. At the entry and exit points of each instrumented region a call to the monitoring library is executed. The monitor then checks the configuration table for MRI requests for the current region. If such a request exists, the measurement part of monitor will be configured accordingly at the start of the region. After that, the

control is returned to the application. At the end of a monitored region, the monitor stops the measurements and aggregates the results before the execution continues.

After reaching the predefined breakpoint, the execution is suspended and the analysis agents retrieve the measurements. If the performance analysis is finished, the agent instructs the application to terminate.

When the analysis agent transfers the requests to the monitor, it first sends the BEGIN_REQUEST command. At the end of the requests it sends an END_REQUEST command. The BEGIN_REQUEST command triggers the initialization of data structures for storing subsequent requests.

For each request, the monitor creates a data structure that points to the region for which the metric is requested, specifies the metric itself and the data structure for the aggregated measurement.

The following measurement types are supported by the monitor:

- Time measurements
- Hardware counters
- Software counters
- Overhead measurements

Time measurements are used to determine how much time is spent in the specific program region. Hardware performance counters are used to count events in the processor or the processor core. The PAPI library is used to access the hardware performance counters in the CPU. The monitor currently supports PAPI (Performance Application Programming Interface) metrics for Itanium 2, Power6, Nehalem and other x86/x64 based architectures. Software counters are, for example, used to count the number of invocations of program regions. Overheads of the monitoring itself are measured in the monitor to enable PTF to reduce program perturbation by automatically optimizing program instrumentation.

The END_REQUEST command triggers the processing of all the requests and generates optimized data structures that reduce the program perturbation during execution.

At a breakpoint, the analysis agent can request the measured performance data. It sends the GET_SUMMARY_DATA command to the monitor (Figure 2) which then executes the return summary data function. This function collects all the measurements in a table and returns this table in binary format to the analysis agent.

2.4 Static program representation (SIR)

A **SIR** (**S**tandard **I**ntermediate Program **R**epresentation) file is a XML document representing the program structure. The SIR document contains information about the code structure (main routine, subroutines, loops, their nesting, etc.) and the used data structures. This document is used by the frontend, the analysis agents, and the Periscope GUI. The SIR file is generated by the Periscope instrumenter.

3 PTF monitor extensions

This section describes the following extensions of the monitoring system implemented in AutoTune:

- Performance measurements for GPGPU kernels
- Energy measurements based on the Intel Sandy Bridge RAPL counters
- Performance measurements for parallel patterns for heterogeneous parallel systems

To facilitate the implementation of the extensions a major refactoring of the existing PTF monitor was required.

3.1 PTF monitor refactoring

The refactoring of Periscope's monitor solved two major issues:

1. The monitor implemented two modes of data collection, namely profiling and time series profiling. In the latter, the aggregated information, e.g. cache misses, is output every n-th invocation of a region and not only at the end of the measurements. This approach is similar to a tracing approach, required substantial source code, and determined the design of the handling of metrics.
2. Metrics that were supported by the monitor were defined in a header file that was included into all the other software components of Periscope and thus ensured consistency among all the components. However, the implementation of the measurement support of those metrics in the monitor was distributed across a larger number of files and extensions needed to be done to all those files which resulted in a big and error prone effort.

During the refactoring, the generation of time profiles was removed from the monitor so that its complexity was reduced and the implementation of the measurements of metrics could be redesigned. In addition, all the required information about metrics is now declared in a central place, i.e., the metric header file and the metric implementation file.

Metrics are now sorted into twelve metric groups. Each metric group consists of a number of semantically related metrics that are measured in the same way, e.g., via hardware performance counters.

PTF currently supports the following counter groups:

- GROUP_PAPI_COUNTER - native PAPI counters
- GROUP_PAPI_POWER6_COUNTER - counters specific to Power 6
- GROUP_PAPI_NEHALEM_COUNTER - counters specific to Nehalem
- GROUP_PAPI_ITANIUM2_COUNTER - counters specific to Itanium 2
- GROUP_LIBCALLS_OVERHEAD_COUNTER - overhead specific counters
- GROUP_TIME_MEASUREMENT - time measurement
- GROUP_MPI - MPI related metrics
- GROUP_OMP - OMP related metrics
- GROUP_OPENCL – OpenCL metrics
- GROUP_CUDA - CUDA metrics
- GROUP_ENERGY - Energy metrics
- GROUP_PIPELINE – Pipeline pattern metrics

The introduction of the counter groups significantly simplified the code implementing the measurements. The extension of the monitor with new metrics is now centralized and changes have to be done in most cases only to those two new files.

3.2 GPGPU extensions

The extension of the monitor for GPGPUs described in this section enables performance measurements for GPGPU kernels written in OpenCL or CUDA. These kernels can be manually programmed as in the high-level pattern approach of Vienna or automatically be generated from HMPP code.

GPGPUs are used as accelerators in current parallel systems, i.e., there is a standard CPU as host processor and one or more GPGPU devices. Figure 3 illustrates the execution of a kernel on such a GPGPU accelerated system. The application and the GPGPU driver are running on the CPU while the kernel is executed on the GPGPU. The application starts by initializing the GPGPU device. Then, it sends a command to the driver to transfer the input data of the kernel to the device memory. Commands sent to the driver are executed asynchronously. Thus, the actual data transfer happens later in the execution, but the sequential order of the commands is respected. After the command to transfer the data, the application enqueues the kernel launch command. The application continues with some other operations and submits a get results command to the driver when it needs the kernel's results. This submission blocks until the results were fetched by the driver from the GPGPU memory after the end of the kernel execution.

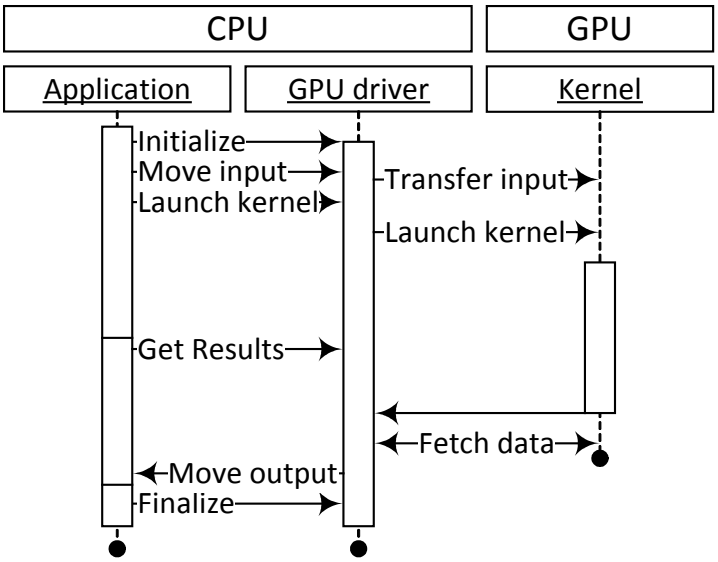

Figure 3: GPGPU execution model

Tuning applications for such heterogeneous systems requires the following measurements:

- Size and time of data transfers
- Execution time of the kernel
- Performance data of the kernel execution on the GPGPU

The extensions added to the PTF monitor enable those measurements for OpenCL and CUDA.

3.2.1 Metrics

The metrics related to GPGPU processing supported by the extended PTF monitor are:

- Data transfer metrics (OpenCL and CUDA)
    - Bytes transferred between the host and the device memory or inside the device memory. (CUDA only)
    - Execution time for the transfers.
- Kernel execution time (OpenCL and CUDA)

  - Time for the execution of the kernel including the overheads for transferring the code and starting it on the device.

- Performance counter based metrics (CUDA only)
  - Metrics for streaming multiprocessors: These metrics provide for example the number of L1 cache misses, branch instructions, and load/store operations to the streaming multiprocessor's shared memory.
  - Metrics for the device memory: These metrics are related to the L2 cache of new Nvidia cards and to the graphics memory. They include the number of hits and misses as well as the total number of accesses.
  - Application metrics: Each streaming multiprocessor provides eight counters that can be incremented from within the application (Profile trigger).

The metrics available on the different versions of Nvidia GPGPUs are documented in Appendix A. Their measurement might require multiple executions of the kernel due to limitations in the hardware support.

3.2.2 OpenCL Profiling Interface

The OpenCL specification provides a profiling interface that allows to access performance data of individual driver commands. Those measurements are based on an Event object defined in the OpenCL specification. A unique Event object is created for each command submitted to the GPGPU driver. This event object stores information about the command. Part of this information is the execution status:

- CL_QUEUED - This indicates that the command has been added to the command queue.
- CL_SUBMITTED - The command has been submitted by the host to the device.
- CL_RUNNING - This indicates that the device has started executing this command. In order for the execution status of an enqueued command to change from CL_SUBMITTED to CL_RUNNING, all commands that this command is waiting on must have completed successfully, i.e. their execution status must be CL_COMPLETE.
- CL_COMPLETE - This indicates that the command has successfully completed.
- Error code - This indicates that status of the command, i.e. if it was executed successfully.

In addition, the event object can store profiling information. To enable profiling, the CL_QUEUE_PROFILING_ENABLE flag of the command queue in the driver has to be switched on. With the command clGetEventProfilingInfo(…) the following time stamps can be retrieved from the event object:

- CL_PROFILING_COMMAND_QUEUED
- CL_PROFILING_COMMAND_SUBMIT
- CL_PROFILING_COMMAND_START
- CL_PROFILING_COMMAND_END

These time stamps are 64 bit values storing the device time in nanoseconds when the command reached the appropriate state.

3.2.3 CUDA Profiling Interface

The CUDA Profiling Tools Interface (CUPTI) enables the creation of profiling and tracing tools that target CUDA applications. It provides four APIs:

- Activity API
- Callback API

- Event API
- Metric API

In the PTF monitor, the Activity API, CUPTI Callback API, and the Event API are used. In the following text, a description of these CUPTI APIs is given.

3.2.3.1 Activity API

The Activity API allows the monitor to asynchronously collect information about CUDA driver and runtime functions on the CPU and on the GPGPU. Each CPU/GPGPU activity is reported in form of activity records. To collect activity records, the CUPTI API uses activity buffers that are automatically filled with activity records when the activities occur on the CPU and GPGPU. Activity buffers, i.e. the memory area allocated for activity records, have to be provided by the application/performance tool. They are inserted into one of three queues of activity buffers. Each queue can hold multiple buffers. Once the buffers are inserted into a queue, they should not be accessed by the application. They are filled by CUPTI with event records. Only after a buffer is returned from the queue, the application can process its content.

CUPTI maintains queues of three types of activity buffers:

- Global queue, which collects all activity records that are not associated with a valid context. The context basically identifies the device on which kernels can be launched.
- Context queue, which collects activity records for a specific context.
- Stream queue, which collects memcpy, memset and kernel activity records associated with a specific stream. A stream is basically a forest of commands submitted to a context that are executed sequentially.

When a record has to be stored, CPUTI first checks for free buffers in the appropriate queue of the stream or the context. If no free buffers are available, it checks the global queue. If there are also no free buffers, the record is dropped.

The activity records store for example the start and end time of driver and runtime functions in nanoseconds, detailed information for CUDA kernels, e.g., number of registers requested, and the size and the start/end time stamps of memory transfer operations.

3.2.3.2 Callback API

The CUPTI Callback API allows a callback function to register for calls to CUDA runtime or driver functions as well as for certain event in the CUDA driver. Callbacks are grouped into domains to make it easier to associate your callback functions with a group of related CUDA functions or events. In the current implementation (4.2) of CUPTI four callback domains exist: a domain for CUDA runtime driver functions, a domain for CUDA driver functions, a domain for CUDA resource tracking, and a domain for CUDA synchronization notification. In order to associate a callback function with one or more CUDA API functions in a domain, a subscriber is used.

Using the callback API a tool can associate a callback function with one or more CUDA API functions. The callback function will be called twice when the CUDA function is invoked and shortly before the function exits. For each invocation of the callback function, the ID of the CUDA function is passed to it as well as some additional data which provide function-specific information.

3.2.3.3 Event API

The CUPTI Event API allows querying, configuring, starting, stopping, and reading the hardware performance counters (event counters) on a CUDA-enabled device. Events on the GPGPU are countable activities, actions or occurrences on a device. Events available across different devices are described in the Appendix A. Events have an identifier, the event name, and are grouped into event categories. To start the measurements, the required events are combined into an event group. An event group can contain events that can be measured simultaneously by the hardware. If the required events

cannot be measured together they have to go into separate groups and be measured in multiple executions of the kernel.

The Event API supports the measurement of the events during the execution of a kernel. After the execution, the counter values can be read. The measurements are done per warp, which is a block of threads that are executed in a SIMD fashion by the streaming multiprocessor (SM). For example, the number of branches will be incremented each time a warp on that streaming multiprocessor executes a branch instruction. Depending on the event and the device, the events are counted on just a single SM, on all, or on some SMs. At the end of the kernel the individual values for the measured events can be accessed.

3.2.3.4 Metric API

The metric API allows the computation of derived metrics from multiple events. Several derived metrics, such as branch_efficiency, are defined. With the API, the appropriate events can be easily combined in an event group. After the measurement with the Event API, the Metric API allows to access the derived metric based on the obtained basic event values.

3.2.4 Integration

In this subsection, the integration of GPGPU extensions for OpenCL and CUDA into the monitor is described.

3.2.4.1 Base definitions

The following new code region types were added to the monitoring system:

- RT_OPENCL_MEM_CMD_REGION - OpenCL memory commands
- RT_OPENCL_KERNEL_ CALL_REGION - OpenCL kernel invocation
- RT_CUDA_MEM_TRANSFER_REGION – CUDA memory transfer
- RT_CUDA_KERNEL_CALL_REGION - CUDA kernel invocation

To be able to call different measurement functions, new counter groups were added as well. These groups are the following:

- GROUP_OPENCL – OpenCL metrics
- GROUP_CUDA - CUDA metrics

3.2.4.2 OpenCL support

The implementation of OpenCL measurements is based on the OpenCL Profiling Interface. The measurements are triggered when an instrumented OpenCL region is executed. The instrumentation consists in a call to the monitor function start_region(…) before the OpenCL region and a call to end_region(….) after the region. The implementation requires that the instrumenter adds to the OpenCL region the retrieval of the event object that is generated when a command is inserted into the command queue. This event object is then passed to the end_region(…) function.

```
start_region(RT_OPENCL_KERNEL_ CALL_REGION,…)
cl_int  clEnqueueNDRangeKernel(cl_command_queue command_queue,
                               cl_kernel kernel,…,
                               &event);
end_region(RT_OPENCL_KERNEL_ CALL_REGION,…,event)
```

In the end_region(…) function the monitor executes a wait operation that blocks until the command is finished and accesses the start and end timestamps via the clGetEventProfilingInfo(…) command.

The instrumentation of the OpenCL regions will be done by the HMPP compiler and the pattern language compiler of Vienna. For now, we rely on a manual instrumentation of the OpenCL call site.

3.2.4.3 CUDA support

The CUDA support is realized using the CUPTI Callback API and the Event API. Again, we assume an instrumentation of the CUDA region with start_region(…) and end_region(…) calls.

To enable measurements of CUDA codes, CUPTI has to be initialized. This initialization is done during the initialization process of the PTF monitor. The monitor looks for GPGPU compute cards and determines the available hardware events.

In the start_region (RT_CUDA_MEM_TRANSFER_REGION, ….) function of the instrumented CUDA region a CUDA context is created and a callback function of the monitor is registered with the CUDA driver. For metrics that require counter measurements, an event group is created and measurements are enabled.

The end_region(….) function only waits for the kernel to be finished. The access to the measured metrics is implemented in the callback function that was registered during start_region(…). From the data structures that are passed to the callback function, the execution time of the memory operation or a kernel can be determined and accumulated in the monitor's internal data structure. The data structure also contains the size of the data transferred for memory transfer operations.

For the measurement of events, the callback function reads the event values for the constructed event group and accumulates the measurements in the code region data structure of the monitor.

3.2.5 Limitations

The execution time is measured indirectly from the host side, not by the GPGPU which executes the kernel. Because of that, various driver overheads might introduce some imprecision into the measurements.

On NVIDIA GPGPU cards only some of the streaming multiprocessors are equipped with hardware counters. Therefore, some of the metrics are not measured on all the multiprocessors. Only if the kernel is running on all of the multiprocessors, reasonable but not precise measurements can be obtained for all metrics.

3.2.6 Measurements

The PTF monitor's GPGPU extension was tested on the Nehalem systems with an NVIDIA GeForce GTX 480 GPGPU. Measurements were done on two implementations of a small application that basically adds two vectors. One implementation is done in OpenCL, while the other one is done in CUDA.

```
  // OpenCL Kernel Function for element by element vector addition
  __kernel void VectorAdd(__global const int* a, __global const int* b,
                          __global int* c, int iNumElements)
 {
        // get index into global data array
        int iGID = get_global_id(0);

        // bound check (equivalent to the limit on
        // a 'for' loop for standard/serial C code)
        if (iGID >= iNumElements) {
          return;
        }


        // add the vector elements
        c[iGID] = a[iGID] + b[iGID];
}
```

Figure 4: OpenCL vector addition

```
// Device code
__global__ void VecAdd(const int* A, const int* B, int* C, int N) {
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < N)
C[i] = A[i] + B[i];
}
```

Figure 5: CUDA vector addition

Number of elements	1M		4M		16M	
Programming Inteface	OpenCL	CUDA	OpenCL	CUDA	OpenCL	CUDA
Transfer time of vector A	825536	1096576	3518976	3336512	13956800	14097504
Transfer time of vector B	1092128	1044448	3407328	3193504	14583296	12669472
Kernel execution time	99104	234400	346784	566016	1301920	1445024
Transfer time of vector C	1711264	1107008	10010816	3365536	34335008	12839968
Instructions executed	N.A.	48576	N.A.	191136	N.A.	771584

Figure 6: Measurements in nanoseconds for the OpenCL and the CUDA version.

Figure 6 presents the obtained measurements on the target system for the two kernels. For this simple kernel it is obvious that the major bottleneck is the transfer of the input and the result data. The instruction count could only be obtained for CUDA. The measurement has to be multiplied by the warp size of 32.

3.3 Energy measurement extensions

In this subsection, we present the energy measurement support integrated into the PTF monitor. It is based on a library that integrates the PAPI-RAPL component and the cpufreq Linux kernel infrastructure developed at LRZ.

The increasing popularity of Green Computing has qualified energy consumption and optimization as a primary concern. The key to energy efficient applications lies in the definition of energy models that minimize, or at least improve, the energy consumption at any instant of the runtime of the applications. Deeply related to these energy models are the energy policies provided by the Linux kernel cpufreq infrastructure. These policies allow the management of the energy consumption based on frequency and processor utilization criteria.

On the other hand, recently server and desktop vendors have equipped their products with on-board energy sensors and tools for obtaining the energy consumption of on-core hardware components and, therefore, of the code that runs on these components. In the case of Intel, these sensors are combined in the so called RAPL (Running Average Power Limit) interface in their SandyBridge microarchitecture.

3.3.1 Metrics

The library currently provides the energy consumption of the cores on the entire CPU socket. Some unit conversions are performed inside the library to treat the energy reads as valid metrics.

3.3.2 Energy measurement library (libenopt)

The implemented library has been written in C++ with wrappers for C and FORTRAN codes. It integrates the cpufreq and PAPI-RAPL features, allowing the optimization of the energy consumption of running instrumented applications by changing the frequency and governor policy of processors on certain instrumented code regions.

The library is mainly composed of two modules: the cpufreq module, which provides a mapping of the cpufreq kernel subsystem and is in charge of changing the frequencies and governors; and the PAPI counters module, which provides a high level API for hiding the intrinsic operations and elements of the PAPI interface and stands for the energy and performance measurements. Other modules of this library are, for example, the exception management module for sanity checks and wrappers for the different input user code flavors.

Instrumented codes written in C++, C or FORTRAN and parallelized either with MPI (processes), OpenMP (threads), or hybrid solutions can be linked against this library.

3.3.2.1 RAPL interface

The PAPI-RAPL component is used as a high-level interface for the RAPL (Running Average Power Limit) energy sensors available at recent Intel chips for measuring energy consumption.

The specific RAPL domain counters available in a platform vary across "product segments":

- Platforms targeting the client segment support the following RAPL domain hierarchy:
  - Package (PKG).
  - Two power planes (PP0 and PP1). In this case, PP0 refers to the processor cores and PP1 reflects the other devices on the chip.
- Platforms targeting the server segment support:
  - Package (PKG)
  - The power plane PP0. In this segment, PP0 refers also to the processor cores, whereas PP1 domain is not supported.
  - DRAM.

The package domain PKG, regardless of the targeted segment, is defined as the processor die. The Model-Specific Registers (MSR) defined for the RAPL domain are:

- MSR PKG POWER LIMIT: that allows software to set power limits for the package.
- MSR PKG POWER INFO: that reports the package power range information for the RAPL usage.
- MSR PKG ENERGY STATUS: that reports measured energy usage. This read-only MSR is updated every 1ms and has a wraparound time of about 60s when the power consumption is high; otherwise it may be longer. That means that, in order to avoid a register overflow, PAPI has to read the event related to this MSR at least once per minute.

PAPI-RAPL provides a set of PAPI native events to interact with the RAPL interface. These events are, among others:

- MINIMUM POWER:PACKAGE0: Minimum power for package 0.
- MINIMUM POWER:PACKAGE1: Minimum power for package 1.
- MAXIMUM POWER:PACKAGE0: Maximum power for package 0.
- MAXIMUM POWER:PACKAGE1: Maximum power for package 1.
- PACKAGE ENERGY:PACKAGE0: Energy used by chip package 0.
- PACKAGE ENERGY:PACKAGE1: Energy used by chip package 1.
- DRAM ENERGY:PACKAGE0: Energy used by DRAM on package 0. Unavailable for client segments.
- DRAM ENERGY:PACKAGE1: Energy used by DRAM on package 1. Unavailable for client segments.
- PP0 ENERGY:PACKAGE0: Energy used by all cores in package 0.
- PP0 ENERGY:PACKAGE1: Energy used by all cores in package 1.
- PP1 ENERGY:PACKAGE0: Energy used by all uncore devices in package 0. Unavailable for server segments.

- PP1 ENERGY:PACKAGE1: Energy used by all uncore devices in package 1. Unavailable for server segments.

Based on the RAPL counters, the library currently returns the sum of PP0_ENERGY:PACKAGE0 and PP0_ENERGY:PACKAGE1. This assumes that the whole socket is given to a single application, which is true on the target system SuperMUC.

The MSR_PKG_ENERGY_STATUS register has a wraparound time of about 60 seconds. To fix this, the library sets an alarm (SIGALRM) to refresh the PAPI-RAPL counters at least every 60 seconds and to accumulate the temporal values.

3.3.3 Integration

The PTF monitor was extended with a special energy counter group that includes the appropriate metrics provided by the energy measurement library. These metrics can be requested by the analysis agent via the MRI. Inside of the instrumentation functions start_region(…) and end_region(…), which are inserted around code regions by the instrumenter. These metrics are then measured as any other metric that is measured by a counter, e.g., execution time. Since the energy metrics are in a different counter group, the special functions of the libenopt library for starting, stopping, and accessing the current energy value are used instead of the standard PAPI functions.

3.3.4 Limitations

The library currently only supports the measurement of the energy consumption of the cores. It will be extended for node level measurements based on hardware in the power supplies and a library provided by IBM.

3.3.5 Measurements

The library has been tested with the two applications provided in the application repository by the LRZ group:

- A SIP (Strongly-implicit procedure) test-bench application developed in FORTRAN and parallelized with OpenMP.
- The Seissol application implemented in FORTRAN and parallelized with MPI.

The two applications have been instrumented with Periscope and then linked with the implemented library. Then, the applications were run in SuperMUC with its SandyBridge EP Intel(R) Xeon(R) CPU E5-2680 at 2.70GHz cluster. For the SIP procedure the alarm refresh time has been set to 50 seconds. This value has been determined during the experiments, being the maximum value in which there were no register overflows. The Seissol application has been run using 32 tasks, whereas the SIP application has been run using eight threads.

The following energy measurements where obtained for the two applications:

```
Seissol with Ondemand governor at 2.7GHz:
     Energy consumed: 35713.6  J
     Time: 473.14 s

SIP with ondemand governor at 2.7GHz:
     Energy consumed: 8360,32 J
     Time: 102,63 s
```

3.4 High-level patterns for heterogeneous systems

This section presents the new support of the PTF monitor for measuring performance data of applications featuring high-level patterns for heterogeneous machines equipped with GPGPUs. As presented also in deliverables D4.1 and D5.1, our approach for programming such applications is

based on language and compiler support as well as a runtime library currently supporting the pipeline programming pattern.

We specify a high-level computational pipeline as a series of computational stages (Figure 7) with data passed between stages. For each of the stages in our framework, different implementation variants tailored for different target architectures (e.g., GPGPU, CPU) may be provided.

```
#pragma pph pipeline with buffer(FIFO,4)
while (data.size == bs) {
     stage1(ifile, data);
    #pragma pph stage replicate(2)
    stage2(data , cblock);
    #pragma pph stage buffer for port(cblock, PRIORITY)
    stage3(cblock , ofile);
}
```

Figure 7: Pseudo code of an application based on the pipeline pattern.

Our main optimization goal is to maximize the overall pipeline throughput. This throughput is defined as the average number of processed results (of the last stage) for a given time-span.

For maximization of pipeline throughput we identify the following tuning points that influence pipeline performance:

- Adjustment of stage replication factor
- Adjustment of pipeline buffer sizes
- Stage implementation variant selection
- Stage granularity (i.e. merging/unmerging of stages)
- Selection of scheduling and resource allocation strategy of underlying runtime systems

3.4.1 Metrics

We identified a set of metrics that are important for automatic tuning of pipelined applications. Those metrics are now automatically measured by the PTF Monitor and the Pattern Coordination Runtime Library. Based on these measurements, tuning scenarios can be evaluated. Hence, we identify the following initial set of related metrics for pipeline application tuning:

- **Stage execution time (PIPE_STAGEEXEC_TIME)**
  This metric comprises execution time of pipeline stage invocations.
- **Buffer input processing time (PIPE_BUFINP_TIME)**
  This metric comprises time to process input objects of one buffer.
- **Buffer output processing time (PIPE_BUFOUTP_TIME)**
  This metric comprises time to process output objects of one buffer.
- **Buffer size (PIPE_BUF_SIZE)**
  This metric comprises the current size of individual buffers.
- **Pipeline execution time (PIPE_EXEC_TIME)**
  This metric comprises the full execution time of one pipeline region including all individual stage invocations and data operations.

3.4.2 Pattern Coordination Runtime Library

Our pattern coordination runtime library is implemented in C++ and provides dedicated classes for representation of pipeline stages and buffers, as well as a management class for configuring and managing the execution of a pipeline construct. For each stage, a C++ stage object coordinates the execution of its associated computational kernels. In this context, stage input buffers are queried and, if all required buffers for a stage are ready, the stage dispatches its computational task to the underlying heterogeneous task scheduling system (cf. PEPPHER heterogeneous runtime based on StarPU[1]).

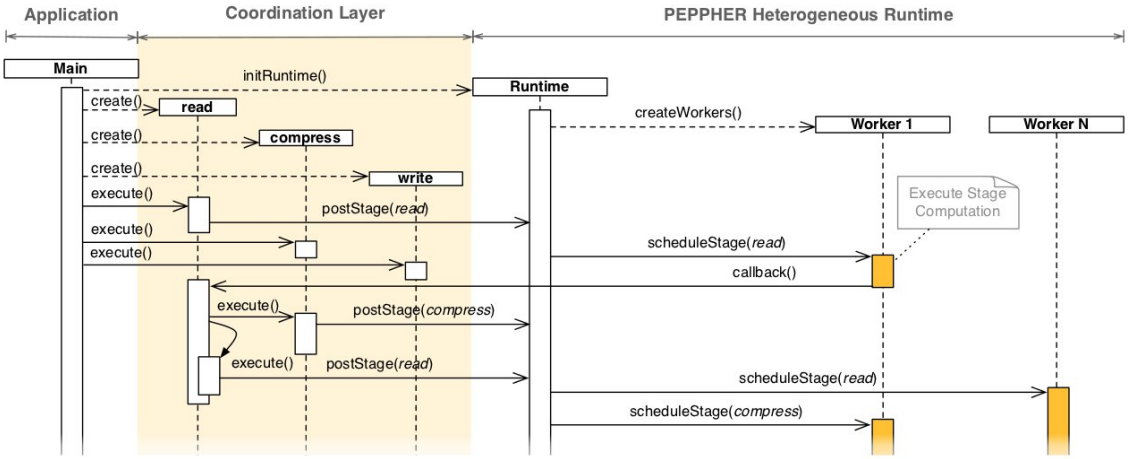

Figure 8: Runtime coordination for 3 stage pipeline

Figure 8 depicts the described runtime coordination for an exemplary 3 stage (read, compress, write) pipeline. An application's main thread instantiates stage objects for each stage in the pipeline. Each stage individually posts its computational kernels for execution on heterogeneous processing units managed by corresponding worker threads.

The pipeline management class (not shown in Figure 8) handles the creation of individual pipeline stages and associated buffers. For providing feedback to performance monitoring tools, it also stores measurements about the pipeline performance.

The stage coordination and execution is implemented locally at each pipeline stage object. Those stage objects as well as buffer structures for stage communication are created by the pipeline management class.

Hence, we implemented storage for stage execution time and buffer metrics in the management layer. Upon each stage invocation, the measured stage run-times are accumulated in the management object and measurement counters are incremented. In addition, associated buffer metrics, are stored for each buffer operation of stages.

3.4.3 Integration

To invoke measurements for pipeline regions, an initialization method (startmonlib) must be called at application start. This call initializes required data structures for metric storage and interchange with the PTF monitor. Subsequently, on request of PTF's analysis agents, the PTF monitor triggers the measurements in the pattern coordination runtime library. At the end of the phase, the monitor calls a function to stop the measurements and retrieves the results from the library via a return summary function.

To map the measurements back to the original source code, we follow PTF's approach to attach region information to each measurement. This information comprises required data, such as region-fileid, line-number and region-type. Together with information in the external SIR file, the mapping can be obtained. For now, the information results from manual instrumentation of the pipeline and the stages,

while in the future it will be generated via the application development framework that has been initially developed at UNIVIE within the EU project PEPPHER. It also includes a source-to-source transformation system that translates a high-level input program with code directives for delineating pipeline regions to an output program utilizing the pipeline coordination layer.

3.4.4 Limitations

In the current preliminary implementation, we restrict feedback granularity to full pipeline region execution. This means that measured result metrics can be processed and delivered to Periscope after a pipeline-region has been finished.

In addition, we will investigate performance feedback at finer granularities such as single stage-region invocations. We further envision a user-defined sampling factor for stage invocations that can be set at runtime by the measurement caller. Such fine-grained feedback can enable full runtime pipeline reconfiguration for individual pipeline invocations. This could be used for implementing tuning-strategies that enable to automatically adapt a pipeline configuration to highly dynamic runtime-parameters such as unexpectedly changing input-data characteristics during a single pipeline execution.

3.4.5 Measurements

Measurements have been performed for an exemplary application utilizing the pipeline layer with enabled measurement functionality. The application consists of a simple three stage pipeline with additional FIFO and PRIORITY buffer structures between stages (Figure 7). The middle stage (stage2) is computationally most demanding and therefore replicated once.

In what follows, an exemplary PIPE_EXEC_TIME metric result for the application in Figure 7, utiling the internal Periscope framework's *MeasurementType* structure (adapted from mrimonitor/return_summary_data.h) for storage, is listed:

```
rank: 0
thread: 0
fileId: 1
rfl: 107
regionType: RT_PIPE_FULL_REGION
samples: 99
metric: PIPE_EXEC_TIME
ignore: 0
fpVal: 0
intVal: 4852686
```

The depicted output comprises the metric measurement in microseconds for the pipeline-region in file with id 1 beginning at line 107. For comparison: The full application execution time (for the whole program) was measured 5.419 seconds.

4 Summary and future work

In the first year of the AutoTune project, the PTF monitor has been extended to provide measurements needed by the tuning plugins. These extensions are based on three libraries, i.e., the CUPTI library provided by NVIDIA for CUDA, the Energy Measurement Library provided by Leibniz Supercomputing Centre (BADBW-LRZ), and the pattern coordination runtime library provided by University of Vienna (UNIVIE).

The extensions for GPGPUs enable access to performance counters, measurement of the kernel execution time, and of the data transfer time and transferred data sizes. This information is needed by the HMPP Codelet Tuning plugin and the Pattern Tuning plugin.

The energy measurements are based on the RAPL counters that enable energy measurements on newer Intel processors created around Sandy/IvyBridge cores. This information is provided for the Energy

Tuning plugin. The measurements will be extended based on a library developed by IBM with measurements of the energy consumption at the power supplies of the SuperMUC nodes.

The monitoring of high-level patterns provides metrics related to the pipeline pattern. These metrics are pipeline throughput, stage execution time, buffer wait time, and buffer occupation. They are used in the corresponding Pattern Tuning plugin.

The implemented extensions require manual instrumentation and adaptation of the SIR program representation. During the second year the necessary extensions to the Standardized Intermediate Program Representation (SIR) will be defined as well as the code instrumentation that generates the SIR file. This work will be based on the compiler infrastructure for high-level patterns developed in Vienna and the HMPP compiler of CAPS. The information will enable to map performance measurements back to individual HMPP codelets as well as individual pipelines and pipeline stages.

Furthermore, this workpackage will develop performance properties and performance search strategies for the automatic analysis of the performance of GPGPU-based codes and of the energy efficiency of applications.

5 Bibliography

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187-198, February 2011.

6 Appendix A.

Metric name	Description	Capability
Texture cache hits	Number of texture cache hits	< 2.0
Texture cache miss	Number of texture cache hits	< 2.0
Branch	Number of branches taken by threads executing a kernel.	All (1.0-3.0)
Divergent branches	Number of divergent branches within a warp.	All (1.0-3.0)
Instructions	Number of instruction executed.	All (1.0-3.0)
Warp serialize	Number of thread warps that serialize on address conflicts to either shared or constant memory.	All (1.0-3.0)
Compute Thread Array (CTA) launched	Number of threads blocks launched on an SM	All (1.0-3.0)
Profile Trigger	User programmable triggers that can be inserted in any place of the code to collect the related information	All (1.0-3.0)
L1 local/global load/store hit/miss	Number of local/global hits/misses in L1 cache.	2.0-3.0
(Uncached) global load/store transaction	Number of global load/store transactions	2.0-3.0
L1 shared bank conflicts	Number of bank conflicts caused due to addresses for two or more shared memory requests fall in the same memory bank.	2.0-
Texture unit 0/1/2/3 cache sector queries/misses	Number of texture cache requests/misses on units 0/1 for compute capability 2.x .and 0/1/2/3 for 3.x	2.0-3.0
Global load/stores of 8/16/23/64/128 bit	Number of 8/16/23/64/128 global load/store instructions that are executed by all the threads across all thread blocks.	2.0-3.0
Warps/Threads launched	Number of warps/threads launched	2.0-3.0
Active warps	Accumulated number of active warps (0-48) per cycle.	2-0-3.0
Active cycles	Number of cycles a multiprocessor has at least one active warp	2.0-3.0
Local/Global load/store	Number of local/global load/store instructions per warp.	2.0-3.0
Shared load/store	Number of shared load/store instructions per warp.	2.0

Instructions issued	Number of instructions issued including replays.	2.0
Instructions issued by group 0/1	Number of times instruction group 0/1 issued one/two instructions.	2.0-3.0
Thread instructions executed in pipeline 0/1	Number of instructions executed by all threads, not including replays in pipeline 0/1.	2.0
Global sub32/32/64/128 load and store instructions executed	Number of sub 32, 32/64/128 of load and store instructions executed.	3.0
Thread instruction executed excl. predicated	Number of instructions executed by all threads, not including predicated instructions.	3.0
Warp did not issues due to barrier	Number of active warps that did not issue due to barrier	3.0
L1 local/global/shared load/store transactions	Number of local/global/shared transactions per warp	3.0
Local/Global load/store divergence replays	Number of replays due to local/global load/store divergence.	3.0

Table 1. Streaming Multiprocessor related metrics

Metric name	Description	Capability
Global (in)coherent loads/stores	Number of coalesced/non-coalesced global memory loads/stores.	1.0-1.1
Global loads/stores 32/64/128 byte	Number of 32/64/128 byte global memory load/store transactions, incremented by 2 for each 32 bytes.	1.2-1.3
Local load/store	Number of local memory load/store transactions, incremented by 2 for each 32 byte transaction.	< 2.0
CTA launched	Number of thread blocks launched on TPC	< 2.0

Table 2 Texture Processing Cluster related metrics

Metric name	Description	Capability
L2 write/read misses in slice 0/1 for compute capability 2.x, 3.x 0/1/2/3	Number of write/read misses in slice 0/1 for compute capability 2.x .and 0/1/2/3 for 3.x of L2 cache. Incremented by 1 for each 32-byte access.	2.0-3.0
L2 read requests from L1/ TEX in slice 0/1 for compute capability 2.x, 3.x 0/1/2/3 of L2 cache	Number of read requests from L1/TEXT that hit in slice 0/1 1 for compute capability 2.x .and 0/1/2/3 for 3.x of L2 cache. Incremented by 1 for each 32-byte access.	2.0-3.0
DRAM read/write requests to FB0/1 for compute capability 2.x, 3.x FB subpartition 0/1	Number of DRAM read/write requests to sub partition 0, increments by 1 for 32-byte access.	2.0

Table 3 Frame Buffer related metrics