Extending the Functionality of Score-P Through Plugins: Interfaces and Use Cases

Robert Schöne, Ronny Tschüter, Thomas Ilsche, Joseph Schuchart, Daniel Hackenberg, and Wolfgang E. Nagel

Abstract Performance measurement and runtime tuning tools are both vital in the HPC software ecosystem and use similar techniques: the analyzed application is interrupted at specific events and information on the current system state is gathered to be either recorded or used for tuning. One of the established performance measurement tools is Score-P. It supports numerous HPC platforms and parallel programming paradigms. To extend Score-P with support for different back-ends, create a common framework for measurement and tuning of HPC applications, and to enable the re-use of common software components such as implemented instrumentation techniques, this paper makes the following contributions: (1) We describe the Score-P metric plugin interface, which enables programmers to augment the event stream with metric data from supplementary data sources that are otherwise not accessible for Score-P. (2) We introduce the flexible Score-P substrate plugin interface that can be used for custom processing of the event stream according to the specific requirements of either measurement, analysis, or runtime tuning tasks. (3) We provide examples for both interfaces that extend Score-P's functionality for monitoring and tuning purposes.

1 Introduction and Related Work

There are numerous tools for monitoring and tuning High Performance Computing (HPC) applications. All of them use similar techniques to gather information about the executed hardware and software environment. Ilsche et al. classify performance analysis tools by three different layers: data acquisition, recording,

R. Schöne (✉) • R. Tschüter • T. Ilsche • D. Hackenberg • W.E. Nagel
Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany
e-mail: robert.schoene@tu-dresden.de; ronny.tschueter@tu-dresden.de; thomas.ilsche@tu-dresden.de; daniel.hackenberg@tu-dresden.de; wolfgang.nagel@tu-dresden.de

J. Schuchart
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 70569 Stuttgart, Germany
e-mail: schuchart@hlrs.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2016*,
DOI 10.1007/978-3-319-56702-0_4

and presentation [10]. In this paper we focus on the monitoring of applications, which includes the first two layers. The two proposed data acquisition techniques are sampling and instrumentation, which Ilsche et al. define in more detail in [10, Sect. 2.1]. Monitoring tools for HPC applications like Score-P, VampirTrace [14], Scalasca 1.x [7], Extrae [3], Open|Speedshop [22], and TAU [23] use different instrumentation frameworks for parallelization paradigms, for example MPI (via PMPI [6, Sect. 14.2]), OpenMP (via Opari [13] or OMPT [5]), CUDA (via CUPTI [15]), as well as automatic and manual user instrumentation.

These frameworks are also used to tune parallel applications, for example for energy efficiency. The Periscope Tuning Framework (PTF) [8], for example, can apply concurrency throttling and frequency scaling to a user instrumented function. Bhalachandra et al. instrument MPI parallel programs [4] to perform load balancing via clock modulation. Rountree et al. use dynamic voltage and frequency scaling (DVFS) instead, but also use MPI instrumentation via MPI's profiling interface [18]. Wang et al. also apply DVFS, but balance OpenMP parallel applications via an Opari instrumentation [27]. On a different scale, the Linux operating system has its own tuning mechanisms, that rely on instrumentation or even sampling which influence the performance and efficiency of parallel programs. The cpuidle kernel infrastructure [17] instruments the Linux scheduler and applies specific power states to idling hardware threads based on the presumed future behavior. The Linux ondemand governor [16] interrupts the workload of a CPU periodically to re-evaluate frequency decisions. Table 1 summarizes the different methods and tools.

Table 1 Examples of existing monitoring and tuning tools, their data acquisition techniques and the supported recording or tuning options

Tool	Data acquisition	Recording/tuning
Monitoring		
Score-P [12]	Instrumentation, sampling	Summarization, logging
VampirTrace [14]	Instrumentation	Summarization, logging
Scalasca 1.x [7]	Instrumentation	Summarization
Extrae [3]	Instrumentation	Logging
HPCToolkit [1]	Sampling	Summarization, logging
Open|Speedshop [22]	Instrumentation, sampling	Summarization, loggings
TAU [23]	Instrumentation, sampling	Summarization, logging
Tuning		
Renci/UNC [4]	Instrumentation	Clock modulation
Adagio [18]	Instrumentation	DVFS
ENAW [27]	Instrumentation	DVFS
PTF [8]	Instrumentation	Various plugins
ondemand gov. [16]	Sampling	DVFS
cpuidle menu gov. [17]	Instrumentation	Idle states
Green Governors [24]	Sampling	DVFS

Data acquisition techniques are not the only aspect that such tools have in common. Both, monitoring and tuning tools collect metrics like performance counters to enrich the information about the executed application with additional data that can be used to optimize its execution. Since essential components of these tools are shared, a common infrastructure that can be used for monitoring *and* tuning is desirable. This is for example done by Score-P, which supports tuning (via PTF) and recording (profiling and tracing).

With open interfaces, the existing infrastructure can be used to implement new functionalities with little effort. In Sect. 3, we describe an interface of Score-P that can be used to capture additional information. We show how the additional data can help to interpret performance results with three examples. Another extension of Score-P that enables programmers to write additional back-ends for Score-P is presented in Sect. 4. This can exploit the capabilities of the existing infrastructure to optimize the execution of the workload or write alternative performance information which is shown in three examples. Section 5 summarizes our paper and outlines future work.

2 Score-P Overview

Score-P is a highly scalable performance measurement tool that supports various HPC architectures and parallel programming paradigms to enable users to interpret the performance of their parallel applications. To do so, Score-P provides different *adapters*. Adapters interrupt the monitored application to capture and record its current status. Available adapters include the instrumentation of parallel programming paradigms, user instrumentation, and sampling. However, some information about the hardware and software environment is independent of the chosen data acquisition method. Hence, Score-P includes different *services* that collect such independent data. These services include for example system trees, which describe the hardware layout, and metrics like performance monitoring counters (PMCs), which can be used to monitor the utilization of processor resources. The data that is collected by adapters and services is then passed to *substrates*, which represent the recording layer in the classification given by Ilsche et al. Existing substrates implement tracing and profiling.

One major target of Score-P is to provide high code quality and a robust infrastructure. Thus, designing and merging new functionality is a protected process that requires multiple steps. Additionally, some functionality targets only specific architectures or projects and is abandoned once the funding has expired. To increase the flexibility of the sophisticated Score-P infrastructure, we implemented two interfaces that enable users to easily provide additional metrics and implement new substrates. The basic structure of Score-P including our extensions is depicted in Fig. 1.

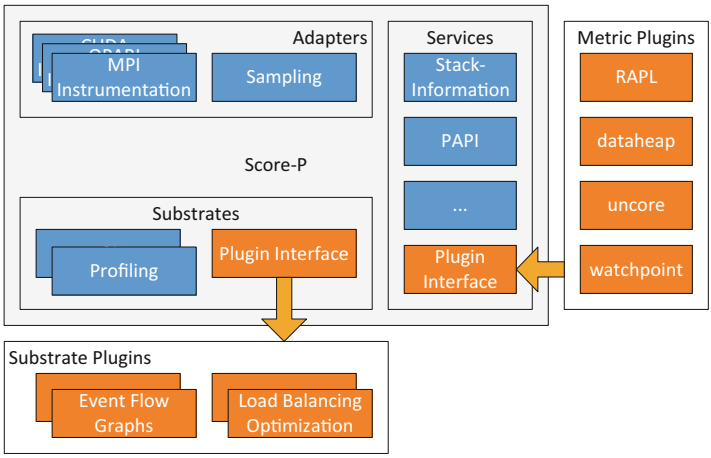

Fig. 1 Score-P overview. Described interfaces and possible extensions are marked *orange*

3 The Metric Plugin Interface

In this section, we describe the Score-P metric plugin interface. We illustrate different design criteria for metric plugins and how Score-P supports them in Sect. 3.1. Section 3.2 lists the calls from Score-P to a plugin in detail. In Sect. 3.3, we measure the overhead for the interface on a contemporary system. Two examples for metric plugins are given in Sects. 3.4 and 3.5.

Historically, Score-P metric plugins succeed the VampirTrace plugin counters that we introduced in [20]. The previous interface has been used in several publications to incorporate new metrics into application performance traces, e.g., power and energy measurements. We translated this interface to Score-P 1.2 and further refined it in Score-P 2.0 in a backward compatible way.

3.1 Metric Design Criteria

Metrics can have different *spatial scopes*, *value ranges*, *information types*, and *temporal scopes*. The spatial scope of a metric can be any software instance or hardware device. Score-P focuses on applications and does not provide detailed hardware topology descriptions like core or NUMA mappings. Therefore, the interface supports four scopes: *per thread*, *per process*, *per computing node*, and *global*. Hardware metrics should be assigned to one of the latter: either to a node or the total monitored system. Examples for the different scopes are *per-thread* stack size, *per-process* allocated memory, *per-node* inlet temperature, and *total system* power consumption. Additional scopes have to be used informally, e.g., if the performance analyst knows that the thread has been pinned to a specific core

and simultaneous multithreading is not used, he can relate all hardware events of a core to the thread that is pinned to it.

Score-P supports different value ranges for metrics: `uint64_t`, `int64_t`, and `double`. The attributes `base`, `exponent`, and `unit` describe the numerical semantics of a metric in more detail: `base` can be either 2 (*binary*) or 10 (*decimal*) and `exponent` specifies the prefix, e.g., -3 with a base of 10 represents *milli*. This allows us to cover a wide range of values with 64-bit integers. In addition, the plugin description contains a human-readable `unit` string. Taken all together a measurement of a metric can be interpreted as: $value * base^{exponent}$ *unit*. For example, to define a memory bandwidth metric in *GiB/s* `base` has to be set to *binary*, `exponent` to 30, and `unit` to *"B/s"*.

The temporal scope of metrics can be defined with a *next*, *last*, *start*, or *point* semantic. The values of *next* metrics are valid from the associated timestamp to the next measurement point. Writing the current amount of allocated memory directly after (de)allocation operations would result in a *next* metric. Generally, *next* metrics represent state changes that are captured directly. By contrast, *last* metrics contain values that are valid from the previous timestamp to the timestamp associated with the current value. This can be the count of operations since the last measurement point. The special case of operations since the start of the measurement, is described with the *start* semantic. Measurements with instantaneous characteristics are described as *point* metrics. For instance taking a instantaneous samples of the current processor voltage without any averaging would be recorded as a *point* metric. It is important to distinguish the temporal scope when correlating metrics with applications measurements, both for visualization and statistical analyses.

Metric plugins can provide their measurement data either synchronously or asynchronously. Synchronous data is gathered when an adapter of the measurement system interrupts the analyzed application. If the plugin defines the metric to be *strictly_sync*, it has to supply a new measurement value on each of these events. Other *sync* plugins can specify a minimum time delta between queries e.g., to account for the underlying measurement resolution. Synchronous plugins should be able to provide data very quickly, otherwise the perturbation can spoil the measurement. Since the reported value will be associated with the current time, it should not be outdated.

For asynchronous (*async*) plugins, measurements are acquired at arbitrary points in time. All values are collected once at the end of the execution. As a result, the plugin is responsible for buffering the measurement data at runtime. Either a background thread, a different process, or even a separate system collects the measurement values and timestamps during execution. Measurements that occur independently from the running application, especially those with a fixed update rate (e.g. average power over 10 ms) should be recorded with an *async* plugin. In the special case *async_event*, a plugin is queried for series of timestamp/value data more frequently during execution. Due to the mismatch between the timestamps from metrics and application events, asynchronously collected data cannot easily be mapped to the application events. One possibility would be trace-replay which sorts the different events and metric values according to the spatial scope of the used

locations and location groups.[1] However, this would rely on trace records as profiles do not store timing information. Thus, asynchronous metrics are not supported when profiling is enabled.

3.2 Calls to Plugins

The interface has been designed to account for the many degrees of freedom that metrics can have. A plugin has to implement five functions for basic functionality. The *entry point* is the only function that has to be exported by the plugin. It passes the necessary function pointers to the Score-P runtime system.

In the *initialization* function of a plugin, all processes can check for the availability of required resources and initialize appropriate data structures. Afterwards, the function `get_event_info` should provide a mapping between the user-supplied metric specification strings and actual metric names, e.g., to resolve wildcards in the specification. Thus, multiple metric names can be returned for each metric specification. Based on the specification of the spatial scope of the plugin, the function `add_counter` is called once per thread, process, host, or once globally. It is used to set up the measurement of the requested metric and should return an identifier that is later used to reference this metric. The last mandatory callback function is the *finalization* call.

Additional functions may be implemented by a plugin depending on the characteristics of its metrics. For (strictly) synchronous plugins, the functions `get_current_value` and `get_optional_value`, respectively, should return the current value of the metric. For asynchronous plugins, the function `get_all_values` is called to provide all collected values at the end of the application run. The values should be timestamped according to Score-P's internal clock. A reference to this clock can be acquired through the `set_clock` callback. Timestamps from external sources need to be converted by the plugin, e.g. using linear interpolation. The optional `synchronize` callback is called for all threads and processes, both at the beginning and at the end of the application run.

A C++ interface is available[2] in addition to the native C interface. The C++ wrapper enables the development of plugins in a more high-level and object-oriented manner. The synchronicity and spatial scope are defined as policies. The plugin class inherits from a base class with policies as template parameters. Facilities for id management, message logging as well as type-safe timestamps (ticks) are provided. All abstractions are done with runtime-efficient in mind (Fig. 2).

[1] In the Score-P syntax locations define scopes that are monitored. Typically a single location is a thread that is executed on a CPU (CPU location) or an external device. Multiple locations can be grouped to location groups, e.g., all OpenMP threads within a process or all processes within a compute node.

[2] https://github.com/score-p/scorep_plugin_cxx_wrapper.

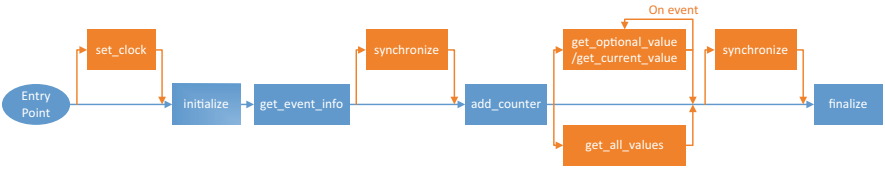

Fig. 2 Order of functions triggered in metric plugins by the Score-P measurement infrastructure. *Blue* elements depict mandatory functions, optional functions are colored *orange*

3.3 Introduced Overhead

This section compares the overhead introduced by plugins by testing minimal strictly synchronous and asynchronous metric plugins. Listing 1 shows the source code of the test program. The workload of this test case is reduced to a main loop generating a predefined number of function calls. The source was compiled with the Score-P instrumenter and automatic compiler instrumentation enabled. With this setup, two events will be recorded for each function call—one event for entering and another event for leaving the function. All experiments were executed on a dual-socket system equipped with Intel Xeon E5-2690 v3 processors running at 2.5 GHz. We run each of the experiments ten times and use the median runtime for further calculations.

In the first experiment, the runtime overhead for minimal strictly synchronous metric plugins is investigated. The plugin is implemented to not take any measurements but to return 0 as current value. The program was executed with the Score-P infrastructure attached in profiling mode. Figure 3 depicts the experiment results. The points in this figure represent measured values, the lines indicate best fits generated by linear regression. The baseline for this experiment is an application run without a registered plugin. In additional runs, a plugin provides varying numbers of metrics ranging from 0 to 4. The runtimes were determined by querying the inclusive time of the main function with the *cube_stat* tool. The results show the same runtimes for runs without a plugin registered and runs with a registered plugin that produces no metric. Hence, there is no runtime penalty for just registering a plugin. Nevertheless, there is an initial overhead when the first metric is activated. We denote this initial overhead *activation factor* α. Based on the experiment result α can be determined to 6.67 ns. This initial overhead is more costly than the overhead of adding further metrics. With a linear regression over the slopes of the lines for n metrics ($n \geq 1$) the cost for adding a strictly synchronous metric can be determined. In our experiments the additional cost β for a single metric is 4.97 ns ($\approx$20 cycles).

Generally, the overall costs can be calculated by the term $\alpha + \beta * n$.

In addition, we repeated the measurements and repeat the experiments with one active internal Score-P metric recording the CPU cycles via Linux perf. Since there is always at least one strictly synchronous metric active, α cannot be measured anymore. In these measurements a higher runtime and more variation is noticeable. Both can be related to the perf metric. β increases to 6 ns (24 cycles).

Listing 1 Minimal program to determine overhead

```
void foo()
{
}
int main()
{
  unsigned long long i=0;
  for (i=0;i<NUM_CALLS;i++)
    foo();
}
```

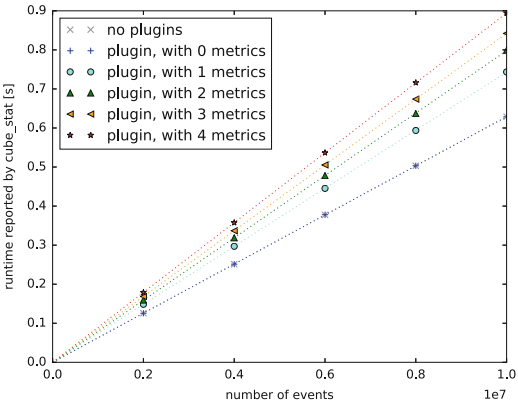

Fig. 3 Measured overhead for minimal strictly synchronous metric

In the second experiment, a minimal asynchronous metric plugin was used. The minimal program was compiled to produce 5,000,000 function calls. The asynchronous metric plugin writes 1, 2, 3, 4, or 5 million elements at the end of the application run. As the profiling mode of Score-P currently does not support asynchronous metrics, we used the `time` command line tool to compare the experiment runtimes. Regardless of the number of supplied elements, no change in the runtimes could be detected. As expected for asynchronous metric plugins, the runtimes are always similar to the ones without plugins.

3.4 Use Case: Uncore Counter

The first example of a metric plugin provides information from Intel uncore performance counters (UPMCs). UPMCs are used to monitor events in uncore devices that are shared by the processor cores, like the integrated memory controller, the last level cache slices, or the power control unit (PCU). The available uncore devices and their respective performance events are described in vendor manuals, e.g. [11]. Linux provides the perf_events interface [28] to access them from user space. This interface is also used by PAPI [25] which relies on libpfm to assign

events to names. However, the support for uncore components depends on the Linux kernel version, e.g., uncore events for Intel Haswell processors are available since kernel 3.18. Older kernels that are often used in HPC do not support such events. Another interface that allows users to poll UPMCs is likwid [26]. However, it relies on accesses that are usually only available for privileged users. To circumvent these restrictions, likwid provides a daemon that can be run as root and polled from userspace applications. While this solves the issue of the restricted access, it also increases the latency for reading values.

Instead, we use a direct access to the perf_event interface or, alternatively, the x86_adapt kernel module [19]. This kernel module exposes save register regions that can be read or written from user space. To provide meaningful names for the events, we use libpfm.

These metrics are registered per-host. Thus, the master thread of one process on each host will set-up the UPMCs and collect their data. Each registered event is measured on all sockets. Thus, on a dual-socket system, one registered event will result in two metrics being included in the trace. To distinguish events from different sockets, the plugin includes the socket ID in the metric name. This information can be used later to match the captured software information if the scheduling of threads and processes is known.

One use case for this plugin is to visualize the number of cores that reside in certain idle states. Such an information can be used to check whether intentionally idling processor cores are placed into a hardware idle state by the operating system. To be able to map the metrics to a group of OpenMP threads, we pin the first twelve threads of the monitored application to the cores on the first socket and the remaining threads on the second socket. In Fig. 4, we show that the operating system correctly uses idle states in OpenMP synchronizing routines. As the threads on the

Fig. 4 Execution of OpenMP parallel NAS benchmark BT (24 threads, Class C). The *top* display depicts the executed regions, the *bottom* displays show the percentage of active cores, based on PCU counter **hswep_unc_pcu::UNC_P_POWER_STATE_OCCUPANCY:CORES_C0:e=0:i=0:t=0**. Within the depicted time frame, the probability that a core in package 0 is not in an idle state is 97.2 % and 88.7 % for package 1 cores, respectively. This corresponds with the time spent in synchronization regions (*cyan*)

second package spend more time in synchronization, the average number of active cores is lower.

3.5 *Use Case: Watchpoints*

Sometimes it is unfeasible or too time-consuming to instrument variables and functions for program analysis. This could be the case if an analyst uses a build system he is not familiar with or if the code is too complex. For these cases, we developed two plugins that enable users to trace local and global variables and the usage of uninstrumented functions.

The first plugin provides information on the number of accesses to a specific memory address, i.e., reading or writing a variable or calling function. Each monitored access to such a variable or function is associated with a specific overhead. The remaining measurement perturbation for Score-P's basic functionality is not influenced. For each registered function or variable, the plugin checks whether it is defined globally, using libbfd. If it found the associated address, it enables performance monitoring via the perf_event interface and watches for accesses to this address. Mapping symbols to addresses is done per process, i.e., in the initialization phase. Thus, in an MPI parallel application each rank can watch a different address. Each monitored variable or function provides a backward-looking per-thread strictly synchronous metric with an `uint64_t` data type. The metrics name does not include address information, which makes it easy to compare values of different processes.

In Fig. 5, we show a resulting trace for the OpenMP parallel NAS benchmark BT in class W. We defined two functions that the plugin should survey for execution: `matmul_sub_` and `matvec_sub_`. The trace indicates that these are executed from all OpenMP threads, but the number of calls to these subroutines is unevenly

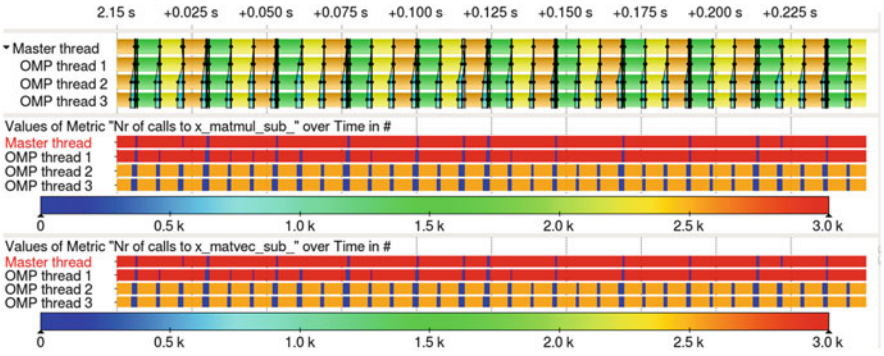

Fig. 5 OpenMP parallel NPB BT (class W, 4 threads), number of calls to sub-functions `matmul_sub_` and `matvec_sub_`. While the first two threads call these functions 3036 times per parallel region (=6*506), the latter threads only call it 2530 times (=5*506), which leads to an imbalance

Listing 2 OpenMP example, which accesses a global variable `d_var`

```
static double d_var=0;
void func(int i){
#pragma omp critical
  {
      d_var=0.5*i;
  }
}

int main(int argc,
char ** argv){
  int i=0;
#pragma omp parallel for schedule(runtime)
  for(i=0;i<100000;i++){
    func(i);
  }
  return 0;
}
```

spread, which creates an imbalance that is depicted by the cyan synchronization phases of the trace. While thread 0 and 1 execute 3036 iterations of the subroutines, thread 2 and 3 only execute 2530 iterations per parallel region. One can assume that the parallel loop assigns *n* chunks of 506 iterations to each thread. A total of 22 chunks are scheduled, where the first and latter two threads execute 6 and 5 respectively, which correlates with the imbalance at the end of the parallel region. This knowledge can be used to assign an optimized number of parallel threads to the workload and predict the scalability of the parallel loops.

The second version of a watchpoint plugin extends the functionality and provides the content of the variable as an asynchronous metric. This means that transitions within the content of the memory region that hold a variable are recorded. To do so we use libbfd and libdwarf to gather the address of a variable whose name is registered by the user. We then set up a hardware breakpoint for this variable using the Linux perf_events interface. In the following, the thread that changes the variable interrupts its execution, gathers the current value and stores it in an array. When multiple threads write the same variable concurrently, the content of the variable cannot necessarily be recorded since another thread can change it before the content has been read by the interrupt handler that is defined by the plugin. Still, the number of recorded transitions matches the number of writes to the variable, even though the recorded values might be flawed.

We show the functionality for a global variable with a short example program (Listing 2). In this example, a number of OpenMP threads access a shared global variable `d_var`. Based on the selected scheduling routine for OpenMP parallel loops, the content of the variable over time changes. The resulting value of `d_var` is depicted in Fig. 6. While for `static` scheduling, the number of iterations are split in a way that one thread executes the first 50,000 iterations and the other thread the remaining 50,000. Thus, while one thread always writes numbers between 0

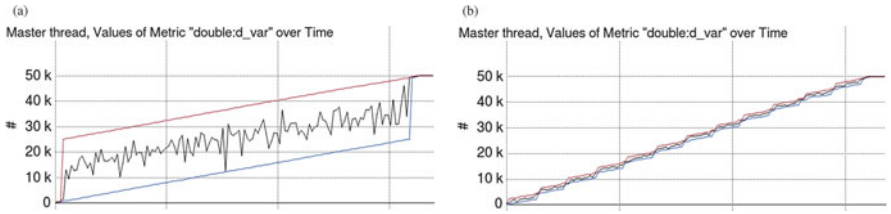

Fig. 6 Value of `d_var` over time for different settings of `OMP_SCHEDULE` and two threads. The minimal value for a time range depicted in one pixel is marked *blue*, the maximal *red*, the average *black*. (**a**) `OMP_SCHEDULE=static`. (**b**) `OMP_SCHEDULE=dynamic,4096`

and 24,999.5, the other thread writes numbers between 25,000 and 49,999.5. For `dynamic` scheduling with a chunk size of 4096 iterations, the written values are much closer as the current chunks of the threads are likely to be close.

In future work, one could implement a monitor for local variables that would be reported per thread. To do so, the plugin would watch for the function that defines the local variable. As soon as the function is entered, the plugin gathers the address of the current stack base, calculates the offset of the local variable via libdwarf and sets up temporary watchpoints for the local variable and the return address. When the return address is executed, the plugin clears the temporary watchpoints.

4 The Substrate Plugin Interface

In addition to the interface for additional metrics, we introduce an interface for substrates. These can use the existing infrastructure in Score-P like adapters and services to implement a new functionality. In previous publications, we described the idea of integrating performance and energy efficiency measurement and tuning [19, 21]. We used VampirTrace where the individual components are tightly coupled. Since the profiling and tracing can not be disabled completely, a significant runtime overhead reduces the applicability of VampirTrace for such an infrastructure.

Score-P already uses an internal substrate interface, which makes it much easier to decouple and integrate additional functionality. However, implementing an internal substrate requires recompilation of the measurement environment and an integration in the Score-P source code tree. This is impractical for experimental and system specific extensions. Thus, we provide a plugin interface to dynamically access the internal substrate functionality. In this Section we describe the interface itself and three plugin implementations, which make use of the new interface to increase Score-P's functionality with new tuning and recording options.

4.1 Substrates Design Criteria

Different substrates put diverging demands on the information that is provided by the monitoring infrastructure. Thus, Score-P must not only *pass the incoming events* to the registered plugins, but must also *provide information about the supplied data*. With the proposed interface, substrate plugins can register for specific types of events. These cover general events like the entering and exiting of a function, but also specialized events that are related to specific adapters. With each of these events, plugins receive a minimal set of information, which is an identifier for the thread whose monitoring issued the event and the timestamp associated with it. Further data depends on the type of the event that is monitored and can for example include information about the communication partner (e.g., for MPI events) or a set of strictly synchronous metrics (e.g., for enter and exit events). Substrate plugins may chose to register only for those events that are relevant to them. Additionally, they can query the Score-P runtime for meta-data about the supplied information, e.g. the type and name of the thread where the current event occurred.

If the monitoring is distributed among different processes, plugins should also be able to *communicate* to enable a global view of the current state. Score-P enables plugins to use an internal interface for multi processing paradigm (MPP) communication. With this interface, processes can synchronize their state independent of the MPP used in the analyzed program.

Substrate plugins receive an event when the monitored application finishes, allowing them to write out the collected information. Likewise, when the monitoring is initialized, an appropriate call enables them to read existing configuration variables.

4.2 Calls to Plugins

We designed the interface in a way that enables programmers to access all relevant data to get a most comprehensive status for their monitoring or tuning implementations. The interface currently consists of three major parts:

1. The plugin definition, which provides callbacks to the substrate plugin for 15 management events,
2. A list of 62 application events that a substrate plugin can register for, and
3. A list of 46 callbacks to Score-P internals, that enable plugins to interpret events and synchronize the distributed state.

To register one or multiple substrate plugins, users set the environment variable `SCOREP_SUBSTRATE_PLUGINS`. When monitoring is initialized, Score-P reads this variable and attempts to load the respective libraries. If for example, the plugin `foo` is registered, Score-P loads the shared object `libscorep_substrate_foo.so`. Afterwards, it retrieves the plugin

definition. Management events that are supplied with the plugin definition are stored for future reference. Afterwards, Score-P initializes the substrate by calling its `initialize` function. If the initialization failed, a warning is prompted and Score-P de-registers the plugin. If the initialization succeeds, plugins are supplied with callbacks to internal functions (`set_callbacks`). These can be used to retrieve internal information (e.g., the scope of a metric or the name of a location) and to access internal functionality like a synchronization mechanism, which transparently maps the calls to the used MPP. The usage of MPP functions should be delayed until the MPP is available, i.e., `initialize_mpp` is called. After Score-P callbacks are provided to the plugin, a list of functions for application events is gathered via the function `get_event_functions`. From this moment on, internal definitions (e.g., metrics or code regions) can be defined. Substrates receive such information via the `new_definition_handle` function. Later in the initialization phase, an identifier is assigned to each substrate plugin via a call to `assign_id`. This identifier can later be used to store and retrieve thread-local data. Afterwards, the measurement is started and the plugin is able to retrieve the same management and application events as the existing substrates, profiling and tracing. When the monitoring ends, substrate can receive calls when Score-P is about to unify the collected monitoring data (`pre_unify`), when it flushes data

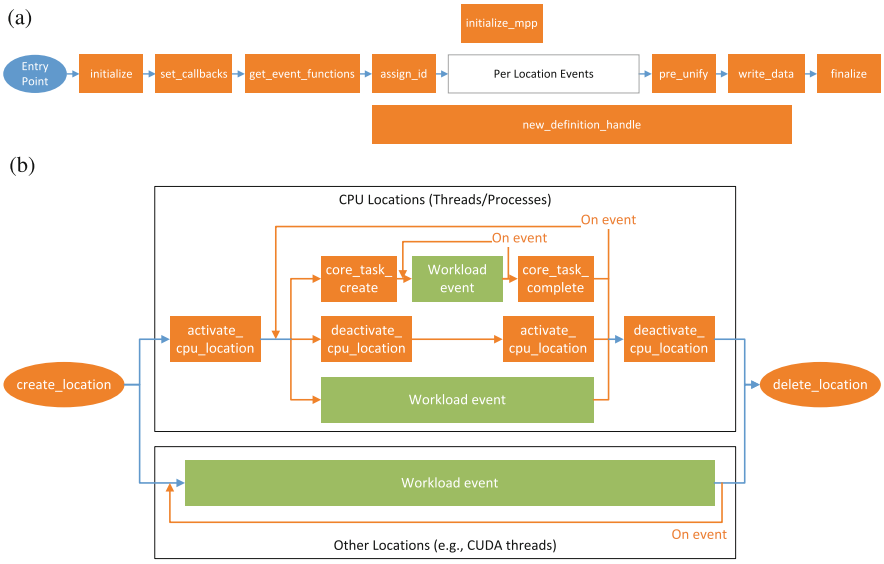

Fig. 7 Order of calls to substrate plugin management functions. All functions except for the plugin definition (entry point) are optional. Management events issued by Score-P are colored *blue* (mandatory implementation) or *orange* (optional implementation). Application events that are issued by the monitored application are colored *green*. (**a**) Per process substrate plugin calls. (**b**) Per location substrate plugin calls

to the file system (`write_data`) and when the measurement system is shut down (`finalize`).

In the measurement phase, plugins are called whenever a new location (e.g., a thread) is created (`create_location`). Locations are distinguished into CPU locations and other locations, e.g., threads that are executed on a GPGPU. CPU locations are activated after they are created (`activate_cpu_location`) and de-activated (`deactivate_cpu_location`) before they are closed. In the meantime, they can also be activated and deactivated, e.g., when a thread is suspended from providing monitoring data. If the CPU locations use task model programming (e.g., OpenMP 3 tasks), these tasks are also published to the plugin. Whenever a location is not de-activated, it can create application events. When a location is closed, the `delete_location` function of plugins is called. An overview of per-process and per-location calls is depicted in Fig. 7.

4.3 Introduced Overhead

Score-P loads the plugins in each process using the dynamic linker library functions `dlopen` and `dlsym`. This initialization is performed only once before the actual measurement and therefore introduces no perturbation and limited overhead. The retrieved function pointers for event and management functions are stored in Null-terminated lists. If plugins do not implement specific functions, the effective length of these lists is reduced. When an event or management function is called within Score-P and at least one plugin registered for this function, the measurement environment traverses the respective list and calls the registered functions. If no plugin registered for an event, the plugin infrastructure does not cause any overhead.

The overhead is analyzed in experiments designed similar to the tests presented in Sect. 3.3 using the same system and test program (Listing 1). Runtime events are recorded by Score-P's profiling substrate and the inclusive runtime of the main function is determined in combination with the `cube_stat` tool. We do not use any metrics, but a minimal substrate plugin that registers for enter and exit events as defined in Listing 3. Again, we change the number of loops that call the instrumented function `foo`, repeat the measurement of each problem size ten times and use the median result. The resulting runtimes are depicted in Fig. 8, where measured values are points and the lines represent the linear regression of these points. The difference of the slopes of the two linear fits represents the costs for a single call to the substrate, which happens to be 3 ns (12 cycles).

4.4 Use Case: Region-Based Energy Efficiency Tuning

As a first example for back-ends, we use libadapt, which has previously been used to enable energy efficiency optimizations with VampirTrace, e.g. for OpenMP

Listing 3 Minimal substrate event

```
static void enter_region( ... ){
}
static void exit_region( ... ){
}

/* Register event functions */
static uint32_t
get_event_functions(
SCOREP_Substrates_Mode          mode,
SCOREP_Substrates_Callback** returned)
{
 functions=calloc(...);
 functions[SCOREP_EVENT_ENTER_REGION] = enter_region;
 functions[SCOREP_EVENT_EXIT_REGION] = exit_region;
 *returned = functions;
 return SCOREP_SUBSTRATES_NUM_EVENTS;
}
```

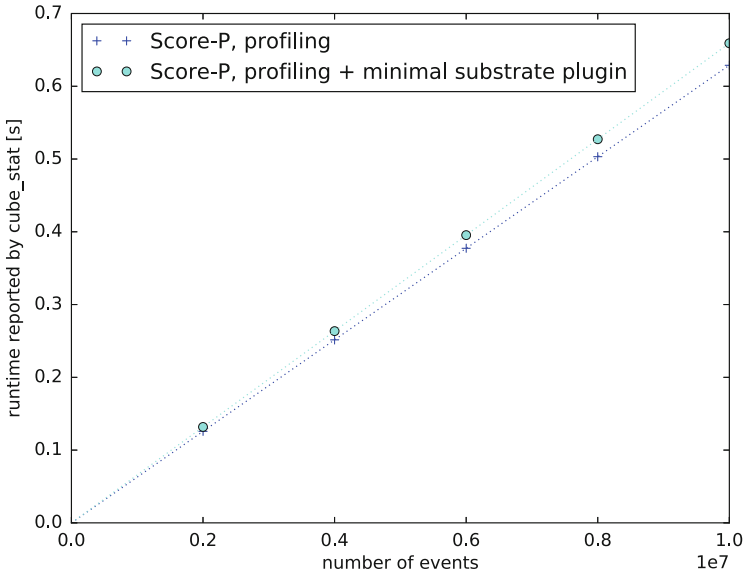

Fig. 8 Measured overhead for a minimal substrate plugin that registers for enter and exit events

parallel [19] and MPI parallel [21] programs. It provides various back-ends that support tuning of processor frequencies, idle states, and various low level optimizations at the level of code-regions.

In order to use libadapt, the plugin registers four management events (initialize, set_callbacks, get_event_functions, and new_definition_handle) and four application events (enter region, exit region, fork, and join). To be able to cope with

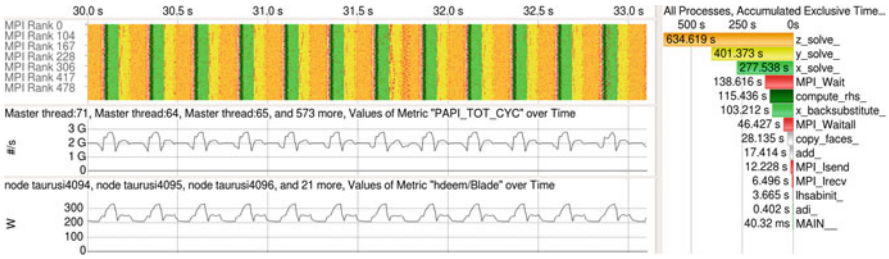

Fig. 9 MPI parallel NPB BT (576 ranks). *Left side* (*from top to bottom*): executed functions (function names on *right panel*), average frequency of involved processor cores, average power consumption of nodes

incoming region handles at enter and exit events, the plugin stores the handles when they are defined. Afterwards, the plugin calls libadapt with every enter and exit event of registered functions and adjusts the hardware/software environment according to the user's specification. Since, Score-P interrupts threads and processes, the user has to enforce the pinning of threads to cores or hardware threads. Neither the plugin nor libadapt check whether the applied tuning parameters result in an optimized execution. However, such an analysis can be done with Vampir and Scalasca. An example is depicted in Fig. 9 where we used Score-P and libadapt to change the processor core frequency of an MPI parallel benchmark depending on the executed region. The power monitoring is provided via a plugin metric for the HDEEM measurement infrastructure [9].

4.5 Use Case: Balancing-Based Energy Efficiency Tuning

Some parallel programs struggle with load imbalances that lead to a significant portion of time spent in synchronization. The overall energy efficiency of such programs can be improved by reducing the clock frequency and voltage for those threads that would enter the synchronization early at nominal speed. Examples that target different parallelization paradigms are given in Sect. 1.

The load balancing substrate plugin intercepts the start and end of a list of blocking MPI and OpenMP calls. It then optimizes the execution of a "synchronized region" r. This region consists of a computing part (which might include non-blocking communication) and a blocking communication part. The plugin assumes that the blocking communication part is fast and slows down the whole synchronized region to an extent that the computing arrives just in time for synchronization. Different synchronized regions are distinguished by using a strictly synchronous metric that provides a unique identifier based on the current call stack. The target frequencies $f_t(r)$ are adjusted in the following way: if the compute time represents

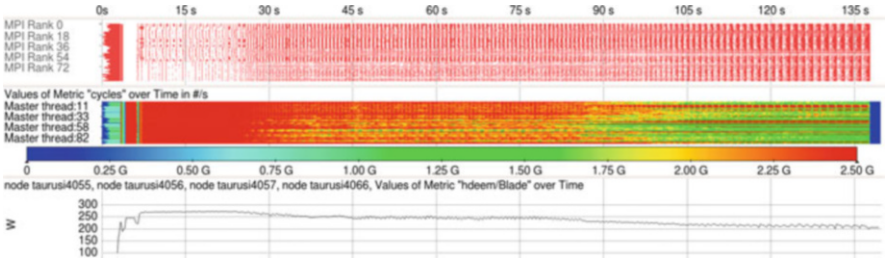

Fig. 10 Execution of weather prediction workload (COSMO SPECS FD4) on 96 MPI ranks with load balancing substrate. Displayed information *from top to bottom*: executed MPI functions (colored *red*); average frequency of involved cores; average power consumption

at least 95% of the synchronized regions, $f_t(r)$ is set to the reference frequency. If it constitutes at least 85%, $f_t(r)$ is set to the frequency that has been used recently $f_m(r)$. If it is less than 85%, f_t is computed by multiplying $f_m(r)$, with the fraction of the computation time and adding a delta frequency to still arrive too early for synchronization in future executions: $f_t(r) = \frac{t_{compute}}{t_{total}} * f_m(r) + \delta$. To avoid flickering frequencies, the maximal predicted optimal frequency of the previous four repetitions of the synchronized region is applied (Fig. 10).

4.6 Use Case: Event Flow Graphs

As a third example, we present event flow graphs comparable to [2]. Event flow graphs represent a function call sequence of a program where each node represents an instrumented region, and each edge the transition rules between the regions. In our version, each node represents a specific call stack and is labeled with the name of the lowest function of the respective stack, i.e., the instrumented functions. To distinguish call stacks, we use the same metric that is also used in the previous section. We use three different notations for edge labels. The first one is represented by a single number n, which describes that this transition is taken the nth time the previous node is traversed. The second notation comprises three numbers i, j, k. Here, i and j describe the first and last time the previous node is traversed and this transition has been taken. k describes the stride: the transition is taken when the previous node is executed the ith, $(i+k)$th, $(i+2k)$th $\ldots$, jth time. The third notation i, j, k, l, m extends this scheme with additional information on nested loops. The outer loop has stride l and is executed m times. This enables us to further reduce the number of edges when a loop that can be represented with three values is interrupted at a regular interval.

One example for event flow graphs is given in Fig. 11, which depicts the main loop of the first MPI rank of the NAS Parallel Benchmark LU. The MPI communication within this loop starts with an `MPI_Send` (top node) and ends with

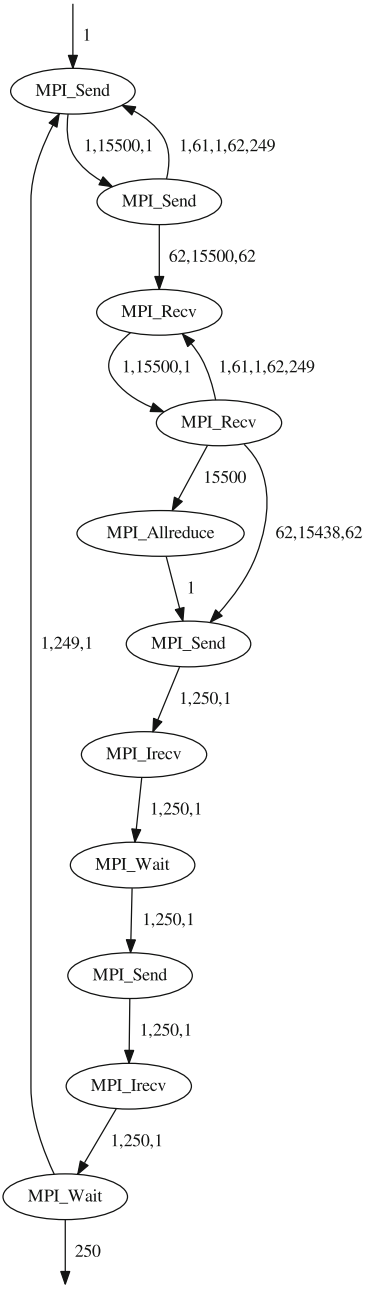

Fig. 11 Event flow graph of the MPI communication for the inner computation loop of MPI parallel NAS Parallel Benchmark LU (Class A, 4 ranks), rank 0

Listing 4 Communication in inner compute loop for first rank of MPI parallel NPB LU - Class A, 4 ranks total

```
for (i=1;i<=250;i++) {
  for (j=1;j<=61;j++) {
    MPI_Send();
    MPI_Send();
  }
  for (j=1;j<=61;j++) {
    MPI_Recv();
    MPI_Recv();
  }
  if (i == 250) {
    MPI_Allreduce();
  }
  MPI_Send();
  MPI_Irecv();
  MPI_Wait();
  MPI_Send();
  MPI_Irecv();
  MPI_Wait();
}
```

an `MPI_Wait` (bottom node). This loop is executed 250 times. The event flow graph can be used to reproduce the communication pattern for testing purposes. Listing 4 depicts such a reproduced code.

The same plugin can also be used for OpenMP parallel programs. In another example, we execute a thread parallel NPB LU with size C on 24 threads and extend the performance measurement with PAPI metrics that are provided by Score-P. To illustrate the effectiveness of the program execution, we color the nodes and edges depending on their relative stall cycles.[3] A green edge or node has no or only some stall cycles, a red node or edge indicates that most cycles are spent stalled. A general overview of the program is depicted in Fig. 12a. However, such a representation cannot depict nested calls. In the next step, we attribute a node to every enter and exit event. Now, the nodes represent single monitoring events and the edges the regions between the instrumentation points. Since monitoring events do not provide performance metrics, only the compute regions (edges) are colored. To limit the amount of events, we filter `omp flush` directives. A fragment of the resulting plot is depicted in Fig. 12b.

[3] *Relative stall cycles* $= \frac{\textit{CYCLE\_ACTIVITY:CYCLES\_NO\_EXECUTE}}{\textit{PAPI\_TOT\_CYC}}$.

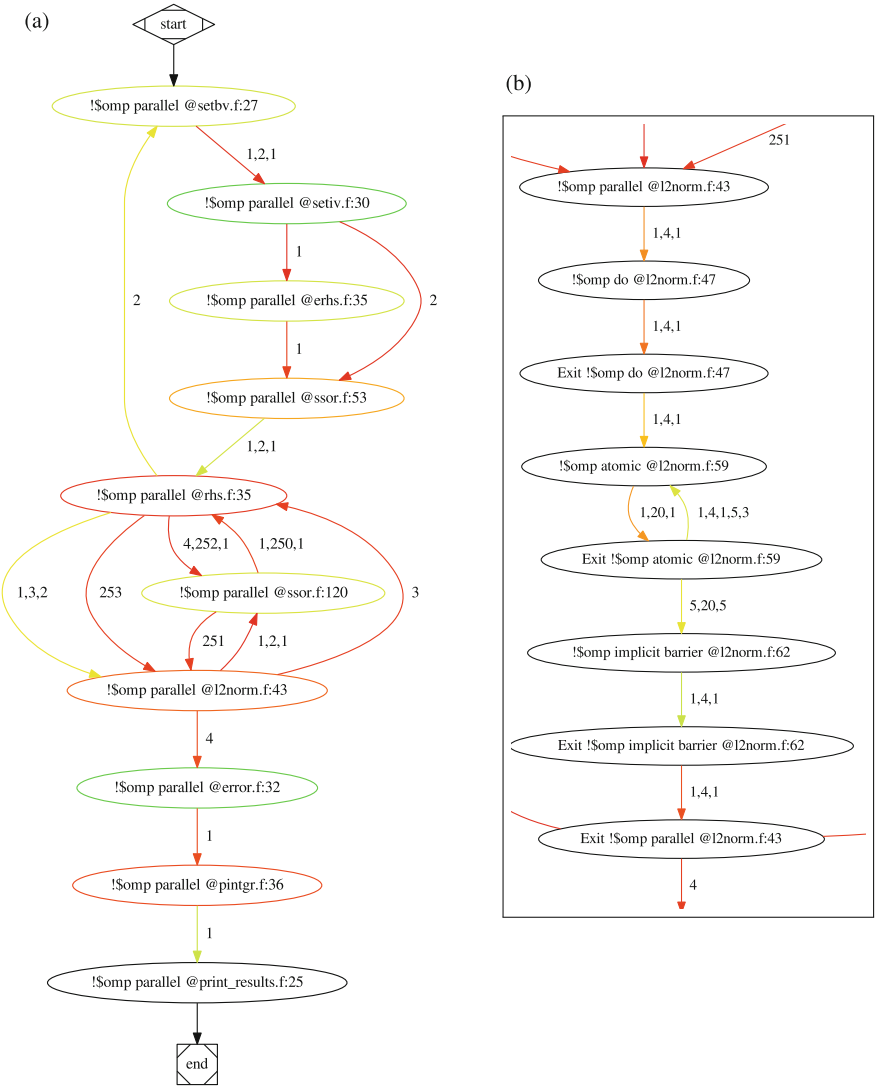

Fig. 12 Event flow graphs of parallel regions for NAS Parallel Benchmark LU (OpenMP, Size C) with colored nodes and edges. A *green color* indicates no stall cycles, *red* indicates a high amount of stall cycles. (**a**) Master Thread, Event flow graph of parallel regions. (**b**) Event flow graph of OpenMP instrumentation, zoom into parallel region `@l2norm.f:43`

5 Conclusion and Further Work

In this paper we described two interfaces that can be used to extend the functionality of Score-P. We summarized the general idea behind the interface and the calls that possible plugins do receive. Additionally, we demonstrated that the expected runtime overhead of the interfaces is adequate, compared to the overhead that is introduced by the remaining Score-P infrastructure. Furthermore, we have shown several examples for the described interfaces. We demonstrated that watchpoints can be used to monitor accesses to functions and variables. This enables analysts to investigate them without an explicit instrumentation. We also, described how performance counters can be used that can not be associated to single threads. For substrates, we demonstrated that it is possible to tune the hardware/software environment at the level of code-regions. We also demonstrated how a balancing-based energy-efficiency optimization could be implemented. Our last use case recorded event flow graphs. Such a plugin can be used to provide performance analysts with a high-level abstraction of the recorded events, since it reduces the number of displayed events significantly in comparison to traces. It can also be used to accompany profiles that do not store the order of executed regions.

Future work includes supplemental spatial scopes for metrics. For example, uncore metrics, as described in Sect. 3.4, would benefit if they could declare that they are recorded per socket. To implement such scopes, the system tree, which is gathered by Score-P must collect and store architectural information from within a compute node. Another challenge is the mapping of hardware thread events to software threads, which relies on such an extended system tree. Here, Score-P could parse the affinity of monitored software threads and store it for a post-mortem analysis. Finally, the analysis tool Vampir should be extended so that metrics of different scopes can be tallied up. For example, if the instructions are counted per thread and the last level cache accesses are counted per socket, the instructions per cache access can be calculated per node.

Acknowledgements This work has been funded by the Bundesministerium für Bildung und Forschung via the research project Score-E (BMBF 01IH13001), the German Research Foundation (DFG) in the Collaborative Research Center "Highly Adaptive Energy-Efficient Computing" (HAEC, SFB 912), and by the European Union's Horizon 2020 Programme in the READEX project under grant agreement number 671657.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exper. (2010). doi:10.1002/cpe.1553
2. Aguilar, X., Fürlinger, K., Laure, E.: MPI trace compression using event flow graphs. In: Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par) (2014). doi:10.1007/978-3-319-09873-9_1

3. Barcelona Supercomputing Center: Extra user guide manual for version 3.1.0. https://www.bsc.es/sites/default/files/public/computer_science/performance_tools/extrae-3.1.0-user-guide.pdf. Online at bsc.es; Accessed 20 Dec 2016
4. Bhalachandra, S., Porterfield, A., Prins, J.F.: Using dynamic duty cycle modulation to improve energy efficiency in high performance computing. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (2015). doi:10.1109/IPDPSW.2015.144
5. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: Ompt: an openmp tools application programming interface for performance analysis. Lect. Notes Comput. Sci (2013). doi:10.1007/978-3-642-40698-0_13
6. Forum, M.: MPI: a message-passing interface standard. version 3.1 (2015). http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Online at mpi-forum.org; Accessed 20 Dec 2016
7. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput. Pract. Exper. (2010). doi:10.1002/cpe.1556
8. Gerndt, M., César, E., Benkner, S. (eds.): Automatic Tuning of HPC Applications - The Periscope Tuning Framework (PTF). Shaker Verlag, Herzogenrath (2015)
9. Hackenberg, D., Ilsche, T., Schuchart, J., Schöne, R., Nagel, W.E., Simon, M., Georgiou, Y.: Hdeem: high definition energy efficiency monitoring. In: Energy Efficient Supercomputing Workshop (E2SC) (2014). doi:10.1109/E2SC.2014.13
10. Ilsche, T., Schuchart, J., Schöne, R., Hackenberg, D.: Combining instrumentation and sampling for trace-based application performance analysis. In: Tools for High Performance Computing (2015). doi:http://dx.doi.org/10.1007/978-3-319-16012-2_6
11. Intel: Intel xeon processor E5 and E7 v3 family uncore performance monitoring reference manual (2015). Reference number: 331051-002
12. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al.: Score-p: a joint performance measurement run-time infrastructure for periscope, Scalasca, Tau, and Vampir. In: Tools for High Performance Computing (2012). doi:10.1007/978-3-642-31476-6_7
13. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. J. Supercomput. (2002). doi:10.1023/A:1015741304337
14. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with Vampir, Vampirserver and Vampirtrace. In: Parallel Computing Conference (PARCO) (2007)
15. NVIDIA: CUPTI user's guide (2016). http://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf. Online at docs.nvidia.com; Accessed Dec 2016 20
16. Pallipadi, V., Starikovskiy, A.: The ondemand governor past, present, and future. In: Proceedings of the Ottawa Linux Symposium (OLS) (2006). https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf. Online at kernel.org
17. Pallipadi, V., Li, S., Belay, A.: cpuidle: do nothing, efficiently. In: Proceedings of the Ottawa Linux Symposium (OLS) (2007). https://www.kernel.org/doc/ols/2007/ols2007v2-pages-119-126.pdf. Online at kernel.org
18. Rountree, B., Lownenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.: Adagio: Making dvs practical for complex hpc applications. In: Proceedings of the 23rd International Conference on Supercomputing (ISC) (2009). doi:10.1145/1542275.1542340
19. Schöne, R., Molka, D.: Integrating performance analysis and energy efficiency optimizations in a unified environment. Comput. Sci. Res. Dev. (2013). doi:10.1007/s00450-013-0243-7
20. Schöne, R., Tschüter, R., Hackenberg, D., Ilsche, T.: The vampirtrace plugin counter interface: introduction and examples. In: Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par) Workshops (2011). doi:10.1007/978-3-642-21878-1_62
21. Schöne, R., Treibig, J., Dolz, M.F., Guillen, C., Navarrete, C., Knobloch, M., Rountree, B.: Tools and methods for measuring and tuning the energy efficiency of HPC systems. Sci. Program. (2014). doi:10.3233/SPR-140393

22. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|speedshop: an open source infrastructure for parallel performance analysis. Sci. Programm. (2008). doi:10.1155/2008/713705
23. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. (2006). doi:10.1177/1094342006064482
24. Spiliopoulos, V., Kaxiras, S., Keramidas, G.: Green governors: a framework for continuously adaptive DVFS. In: International Green Computing Conference and Workshops (IGCC) (2011). doi:10.1109/IGCC.2011.6008552
25. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Tools for High Performance Computing. In: Collecting Performance Data with PAPI-C (2010). doi:10.1007/978-3-642-11261-4_11
26. Treibig, J., Hager, G., Wellein, G.: Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the International Conference on Parallel Processing Workshops (ICPPW) (2010). doi:10.1109/ICPPW.2010.38
27. Wang, B., Schmidl, D., Müller, M.S.: Evaluating the energy consumption of openmp applications on Haswell processors. Lect. Notes Comput. Sci. (2015). doi:10.1007/978-3-319-24595-9_17
28. Weaver, V.M.: Linux perf_event features and overhead. In: The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath (2013)