The VampirTrace Plugin Counter Interface: Introduction and Examples

Robert Schöne, Ronny Tschüter, Thomas Ilsche, and Daniel Hackenberg

Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden – 01062 Dresden, Germany
{robert.schoene,ronny.tschueter,thomas.ilsche,
daniel.hackenberg}@tu-dresden.de

Abstract. The growing complexity of microprocessors is not only driven by the current trend towards multi-core architectures, but also by new features like instruction set extensions, additional function units or specialized processing cores. The demand for more performance and scalability also results in an increasing complexity of the software stack: operating systems, libraries, and applications all need to exploit more parallelism and new functionalities in order to meet this demand. Both aspects – hardware and software – put pressure on performance monitoring infrastructures that face two conflictive requirements. On the one hand, performance tools need to be somewhat stable without entailing significant software changes with every additional functionality. On the other hand they need to be able to monitor the influence of new hardware and software features. We therefore present a plugin interface for our performance monitoring software VampirTrace that allows users to write libraries that feed VampirTrace with data from new (platform dependent) performance counters as well as hardware features that may not be accessed by Open Source software. This paper describes the interface in detail, analyzes its strength and weaknesses, depicts examples, and provides a comparison to other plugin-like performance analysis tools.

1 Introduction

Profiling and event tracing are the two major analysis techniques to evaluate performance and pinpoint bottlenecks within parallel programs on high performance computing (HPC) systems. A well-established event tracing infrastructure is VampirTrace. Its main focus is to instrument and trace parallel programs written in Fortran or C that are parallelized with the Message Passing Interface MPI [1]. However, VampirTrace also supports OpenMP parallelized programs and hybrid MPI & OpenMP programs, Pthreads, UNIX processes and parallel Java programs [2,3]. The tool is jointly developed by the Forschungszentrum Jülich and the Technische Universität Dresden. It is shipped with OpenMPI [4] and therefore available as a software package for all major Linux distributions.

VampirTrace supports numerous performance counters to track metrics such as PAPI or I/O events. These performance counters are typically standardized

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 501–511, 2011.

and portable. New processor or operating system features are often system specific and not compatible with previous or upcoming systems. However, they influence the overall system performance and need to be tracked by performance monitoring tools. To branch a monitoring infrastructure for each different platform would enlarge the code base significantly – this is inefficient and slows down the development progress. The inclusion of all the new features within the main branch is also not feasible since it would increase both the code size and the error-proneness of the software. We therefore extend our performance monitoring software VampirTrace with the presented VampirTrace plugin counter infrastructure that effectively resolves the described issues. The new interface allows developers to write libraries that feed VampirTrace with data from new (platform dependent) performance counters as well as hardware features that may not be accessed by Open Source software. This effort is an important step towards a VampirTrace infrastructure that enables arbitrary, platform specific performance counters while remaining stable and consistent.

The paper is structured as follows: Section 2 topics design and implementation details of the VampirTrace plugin counter infrastructure. Three different plugin counter libraries that extend the functionality of VampirTrace are presented in Section 3. We discuss design limitations in Section 4 and address related work with respect to plugins for performance analysis tools in Section 5. Finally, Section 6 presents conclusions and outlines future work.

2 The Plugin Counter Interface

2.1 Design

Currently, there are four possible types of plugin counters that differ with respect to their type of synchronicity:

Synchronous plugin counters are very similar to other performance events in VampirTrace. Whenever a VampirTrace event occurs (e.g., the call of an instrumented function or a call into the MPI library), the current value of such a plugin counter is gathered and merged into the event trace.

Asynchronous Callback plugin counters usually start background threads that report data by calling a function of the VampirTrace counter interface. The event data can be gathered from a buffered local or even remote location. Such plugin counter libraries have to provide timestamps along with the counter values since they can not be matched directly to a VampirTrace event. Functions to generate timestamps in a supported format are passed on to the plugin counter library during the initialization phase of VampirTrace.

Asynchronous Post-mortem plugin counters collect tracing information during the full runtime of a program. The event data is collected by VampirTrace after the program has finished. Function calls that implicate overhead occur either prior to or after the program runtime, thus minimizing the program perturbation. If the plugin library itself gathers its data from an external source over a network, the measurement process is not influenced at all.

Asynchronous On-event plugin counters are a hybrid approach. While performance events are collected asynchronously, VampirTrace retrieves the data only when a classic event (e.g., function or MPI call) occurs. The advantage of this plugin counter type is that the event buffer size can be decreased compared to post-mortem plugin counters while still allowing a similar asynchronous collection method.

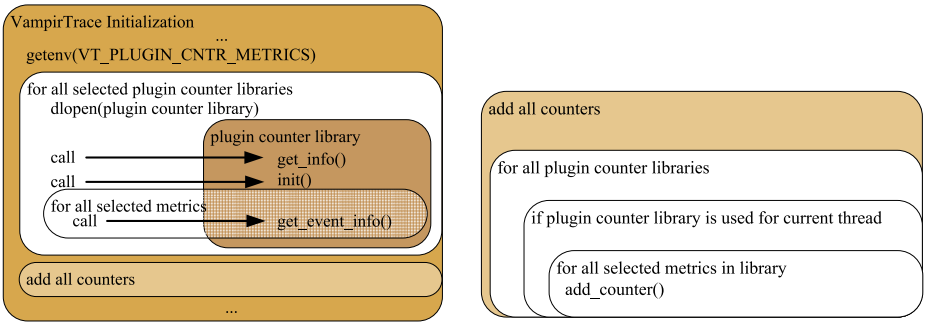
Another distinction for plugin counter libraries is the scope of their counters. For example, PAPI counters can be related to a thread while CPU-related counters are not thread specific. Thread-independent counters can be associated to a host (e.g., network interface counters) or to the whole system (e.g., usage of an NAS). The current implementation of the plugin counter interface allows measuring counters per thread, per process, on the first thread of the first process of each host ("once per host"), or only on the first thread of the first process ("once").

Plugin libraries can also define counter datatypes such as unsigned and signed integer or floating point values. The relation of values to time can be defined as "relates to the current timestamp" (e.g., temperature of components), "relates to the time frame from the last event to the current" (e.g., average power consumption for the last time frame), "relates to the time frame from the current event to the next one" (e.g., the current processor, a thread is scheduled on), "relates to the time frame from the first time stamp to the current" (e.g., PAPI reads).

2.2 Implementation

The VampirTrace plugin counter interface is developed using C. It depends only on POSIX functions and definitions declared in `dlfcn.h` and `stdint.h`. Functions from `dlfcn.h` enable dynamic loading of plugins at runtime. The interface defines functions for initialization, adding counters, enabling and disabling counters, providing results, and finalization. A subset of at least five functions has to be implemented, others are optional. A minimal useful plugin can be written in less then 50 lines of code.

Specific plugin counters can be added by the user who can define the environment variable `VT_PLUGIN_CNTR_METRICS`. This variable consists of the library name followed by the counter name. For example, setting it to `Power_watts` would define the library `libPower.so` and the counter name `watts`. Multiple plugin counters can be passed by separating them with colons. VampirTrace evaluates the specified metrics and checks for the existence of implicitly defined libraries. This is done for every process that is monitored by VampirTrace (e.g., for every MPI rank). Afterwards, the plugin counter libraries are loaded using `dlopen`. Each plugin counter library has to implement the function `get_info`, which provides VampirTrace with pointers to all needed functions. For instance, the `get_event_info` function assigns meaningful names and units to the counters. Furthermore, plugin developers can use this function to extend the passed counter name to multiple counters. The functionality of wildcards can be a possible use case. If the user sets `VT_PLUGIN_CNTR_METRICS` to `Power_*`, `get_info`



(a) Initialization per process (b) Initialization per thread

Fig. 1. Initialization procedures for plugin counters

might add one counter per available node and provide a comprehensible name (e.g., "power for node 1") and unit ("watts") for each. Finally, VampirTrace activates all defined metrics for the corresponding library. The entire process is depicted in detail in Figure 1a. Additionally, for every new thread of the program the required metrics are determined and added as pointed out in Figure 1b.

The trace buffer for the measured thread is shared by VampirTrace and all used plugins. Writing a trace entry is not an atomic operation and a plugin thread can therefore not write to this buffer directly. Mutexes could assure mutual exclusion when writing events but this would add an unacceptable overhead. Therefore, each callback plugin needs to create separate event buffers. The buffer sizes need to be chosen carefully by the plugin library developer to avoid both event loss and excessive memory usage.

2.3 Overhead Analysis

Monitoring tools typically influence the runtime behavior of the monitored application. In our case, both VampirTrace itself as well as the plugins create a certain overhead. The latter is fully in control of the plugin developer and we therefore focus on the overhead that is induced by VampirTrace and the plugin interface itself. We use a synthetic, OpenMP-parallel program that runs 4 threads on the test system. The system consists of 4 Intel Xeon 7560 processors and 128 GiB registered DDR3-1066 memory. Each processor runs at a core frequency of 2.27 GHz. The TurboBoost overclocking feature allows processor cores to increase their frequency up to 2.67 GHz. All measurements are performed with Linux kernel version 2.6.32.12. This configuration is also used for the examples presented in Section 3. The benchmark repeatedly calls an empty function (immediate return), a worst-case scenario in terms of trace overhead. We compare a minimal synchronous and a minimal asynchronous post-mortem plugin counter to the runtime with no counters. The runtime is measured at the beginning and the end of the program, ignoring the initialization and finalization phase of VampirTrace.

When no counter is added, each function call lasts about 1.65 μs. Every synchronous counter increases this time by about 600 ns, most of which is required to write the counter value to the trace buffer. Asynchronous post-mortem counters do not influence the runtime of the program. Therefore, plugin libraries that record events on an external system and gather the data in the finalization phase have no impact on the program execution at all.

2.4 Additional Software Infrastructure

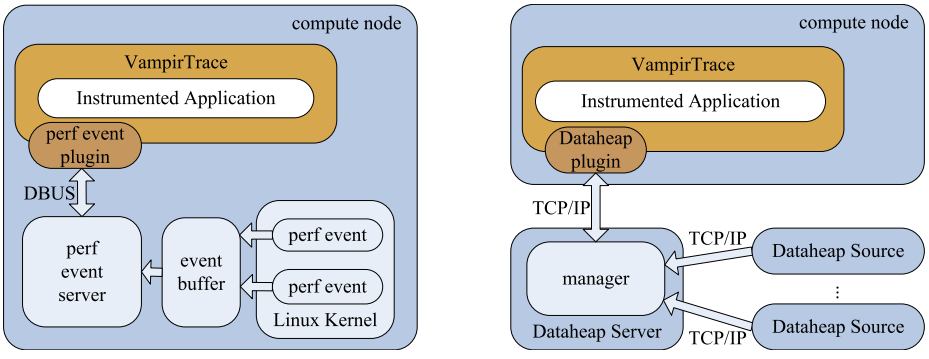
For the examples presented in Section 3, we use two additional tools. The DBUS-based *perf event server* provides enhanced access to kernel tracing events. The *Dataheap* is a distributed counter collection system.

The Linux kernel tracing infrastructure [5] enables event counting on a user level, but certain restrictions apply in a non-privileged context. Users can only read counters that are attached to their processes (*per-task-counters*). This affects some of our use cases in Section 3, for example in case of a plugin that monitors the operating system scheduler events. When the time slice of the observed process ends, the operating system scheduler selects a new task and starts its execution on the CPU. These actions are executed from the context of the observed (previous) process. Scheduling events that are created whenever the own process is stripped *from* a CPU will be reported correctly. Scheduling the observed process *to* a CPU is performed from the context of a different task and will not be reported by a *per-task-counter*. It is therefore necessary to trace scheduling events for all processes. This implies the usage of *per-CPU-counters* that trace all events of a specific type on one CPU. These counters gather information of foreign processes and therefore require privileged rights. Our DBUS-based *perf event server* allows applications to send tracing requests that include their PID, the event that shall be traced, and the desired memory to buffer the data. The server checks for appropriate user rights and available memory and starts the monitoring if both requirements are satisfied. The plugin collects the gathered data from the server after the task finishes and merges it into the trace file (see Figure 2).

The distributed counter collection system *Dataheap* uses a central management daemon that runs on a dedicated server and collects performance data from information sources. The *Dataheap* manager then distributes this data to arbitrary clients, for example monitoring tools. The default usage scenario of the *Dataheap* framework implies a distributed set-up, where sources and clients run on different nodes. We use this infrastructure for several different purposes, for example monitoring I/O activity by reading information from network attached storage servers, or for measuring the power consumption of compute nodes.

3 Examples

In this Section we demonstrate the potential of the VampirTrace plugin counter interface. Two plugin counters exploit the Linux kernel tracing infrastructure,



(a) Perf event server and plugin (b) Dataheap infrastructure and plugin

Fig. 2. Comparison of perf event and Dataheap Plugin infrastructure

which was introduced with kernel version 2.6.31. The third one utilizes the *Dataheap* infrastructure. For our tests we use a small MPI program that executes the following commands:

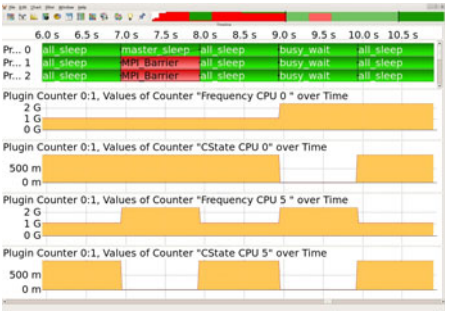
```
0: all ranks: sleep 1 second
1: rank 0: sleep 1 second
2: all ranks: sleep 1 second
3: rank 0: sleep 1 second
4: all ranks: sleep 1 second
5: all ranks: busy waiting for 1 second
6: all ranks: sleep 1 second
7: all ranks: busy waiting for 1 second
8: all ranks: sleep 1 second
```

Each of these commands is followed by an `MPI_Barrier` to synchronize all ranks. The resulting traces are presented in Figure 3, 4, and 5.

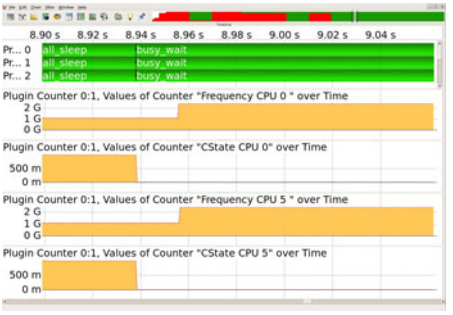
3.1 Power State Tracing

We use three different kernel events to determine the C-state and clock frequency of a CPU: *power:power_frequency*, *power:power_start*, and *power:power_end*. A *power:power_frequency* event is created every time a CPU changes its frequency. *power:power_start* and *power:power_end* correlate with a CPU entering or leaving a sleep state. The plugin is implemented as post-mortem type to reduce the overhead within VampirTrace. This means that the event buffers for the kernel events have to be large enough to hold all events.

The possibility of dynamically overclocking a processor (e.g., via Turbo Boost) is not considered, as both the current availability of overclocking and its real frequency are not passed from the operating system to userspace. This could be fixed by adding information about the registers `aperf` and `mperf` to the reported event. Moreover, switching to another C-state has to be done in kernel



(a) Test application - frequency and C-state for every CPU



(b) Frequency of a CPU is increased with a short delay after it leaves a C-state

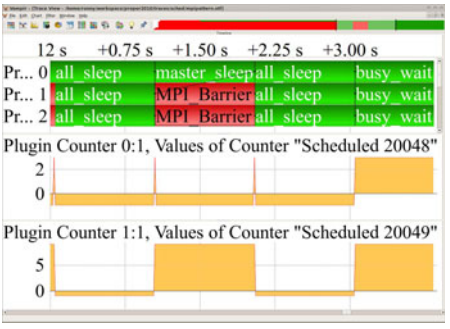
Fig. 3. Power state tracing, frequency and C-state for every CPU

mode since all routines initiating a C-state (e.g., `mwait` or `hlt`) require privileged rights. Therefore we use the *perf event server* introduced in Section 2.4. Switching to another frequency is mostly done by a governor that adapts the processors frequencies automatically based on the recent load (ondemand/conservative governor). The counters displayed in Figure 3a show that whenever a process starts sleeping, its processor immediately switches to a higher C-state. As shown in Figure 3b, the adaption of the frequency is not quite as fast, as the CPU governor bases its frequency scaling decisions on the recent load.

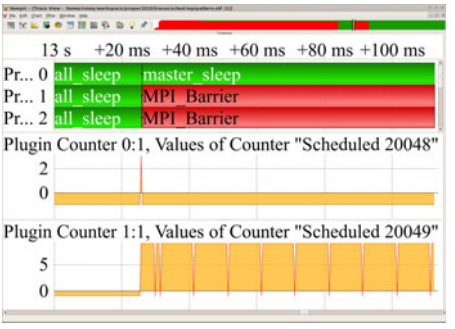
3.2 Scheduler Tracing

Processes or threads that migrate to other cores have to rebuild their register content and their cache entries. The induced overhead can be performance relevant and of interest for an optimization task. Our next plugin therefore monitors *sched:sched_switch* events that are created whenever a process is scheduled onto a CPU or stripped from it. Figure 4a depicts the trace visualization of our test application including scheduler events. Figure 4b shows that there is periodical scheduler activity during the busy waiting phase in a blocking `MPI_Barrier` routine (see process 1 with PID 20049). The counter values correspond to the CPU number that a selected process is scheduled to. For example, when process 0 (with PID 20048) is active, it is scheduled onto CPU 3. Whenever process 0 is put into a sleep state, it is unscheduled (-1).

Since a *sched:sched_switch* event is created every time the operating system scheduler is invoked, tracing this event makes excessive use of the result buffer. With the support for *sched:sched_migrate_task* events, our plugin provides an alternative to trace scheduling behavior. Such a *sched:sched_migrate_task* event is created when a process is scheduled onto a different CPU. This occurs less frequently and reduces the required buffer size significantly. The reduced buffer size comes along with the drawback that the *sched:sched_migrate_task* event can not detect phases, where the observed process is not scheduled onto a CPU.



(a) Overview of scheduler behavior for the test application. Processes are unscheduled (-1) when they enter a sleep state



(b) Counter 1:1 shows periodical operating system interrupts every 10 ms during busy waiting in a blocking MPI routine

Fig. 4. Scheduler event tracing, CPU id for every rank

3.3 Power Consumption Tracing

There are currently three distinct *Dataheap* counter plugins, each with its own advantages and disadvantages. We use the asynchronous post-mortem plugin, which only registers at the management daemon and collects the written data after the application has finished. This solution is clearly preferable in terms of trace overhead. The trace depicted in Figure 5 uses the *Dataheap* framework to provide information about the power consumption of our test system (see Section 2.3).

We use a ZES LMG 450 power meter with a 10 Hz sampling frequency that reports its measurement data to the *Dataheap* manager. The trace shows how the power state changes of the CPUs reduce the power consumption of the whole system. We can see that the OpenMPI implementation of `MPI_Barrier` uses a busy waiting algorithm that consumes considerably more power than our busy waiting loop.

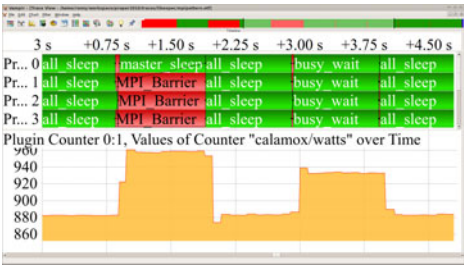


Fig. 5. Power consumption in Watts, collected from an external counter database

4 Limitations

Recording events from performance counters always increases the overall tracing overhead. The proposed VampirTrace extension allows using a post-mortem

interface that moves the overhead for the actual event generation within VampirTrace from within the program runtime to after the program has finished. Therefore, the program perturbation is small (kernel tracing) or zero (tracing of external events, see Section 3.3). However, the post-mortem runtime overhead as well as the memory overhead for the event buffers can be significant. These overheads are not introduced by the plugin counter interface, but by the plugins themselves.

Another limitation is that only one counter is internally measured at a time. This can be a drawback when using numerous synchronous events. In contrast, PAPI and the Linux kernel tracing infrastructure allow reading multiple events with only one function call, thereby avoiding some overhead.

5 Related Work

While there is a wide range of performance analysis tools and libraries, only some of them provide a plugin interface. The *Collector Plugins* of Open|SpeedShop [6] are comparable to our approach. However, while Open|SpeedShop focuses on profiling with only limited tracing support, the focus of VampirTrace clearly is event tracing.

The PAPI project specifies an application programming interface for accessing hardware performance counters. Monitoring these counters can for example give hints how software can be optimized for a specific platform. PAPI version 4 provides the possibility to implement own counters called *Components*, which can be read using PAPI. For design reasons, PAPI has to be recompiled for every new component. Moreover, its design prohibits asynchronous events. However, for synchronous events that are counted for every thread, it is a more flexible and well accepted interface.

The Vampir framework is a well established performance analysis tool chain with a strong focus on scalable high performance computing applications [2,3,7]. Recent work hast demonstrated the applicability of both the event monitor VampirTrace and the performance visualizer Vampir on state of the art systems including hardware accelerators [8,9]. While we used the kernel tracing infrastructure to trace scheduling events, Kluge et al. [10] follow a different approach. They use an additional thread, which monitors all other threads with a specific polling interval. A high polling frequency strongly influences the test system even if no rescheduling events occur. A too low frequency increases the possibility of missed events and reduces the time accuracy of scheduling events.

6 Conclusion and Future Work

This paper presents the design and implementation of the VampirTrace plugin counter interface along with some typical usage scenarios. The main goal of this work is to address the issue of the constantly increasing number of performance events sources that are typically highly platform specific. The plugin counter interface allows these events to be recorded with VampirTrace while at the same

time strongly reducing the need to modify the core source code of the tool. Its tight integration into VampirTrace and the availability of asynchronous events further increase the benefit of this extension. Our exemplary implementation of three different plugins demonstrates the potential of the newly defined plugin infrastructure. All three plugins are used in current research efforts to analyze programs and libraries with respect to performance and energy efficiency. Future work will focus on making the plugin counter interface generally available within the VampirTrace trunk. Moreover, other plugins are currently under development, e.g., a libsensors plugin to read hardware information asynchronously.

Acknowledgment

The authors would like to thank Matthias Jurenz for his continuous and valuable contributions to the VampirTrace project as well as Michael Kluge for his support and his work on the Dataheap project. The authors also thank Intel Germany for providing us with the Intel Nehalem EX evaluation platform. This work has been funded by the Bundesministerium für Bildung und Forschung via the Spitzencluster CoolSilicon (BMBF 13N10186).

References

1. The MPI Forum: MPI: A Message Passing Interface (2010), http://www.mpi-forum.org/
2. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)
3. 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: Bischof, C.H., Bücker, H.M., Gibbon, P., Joubert, G.R., Lippert, T., Mohr, B., Peters, F.J. (eds.) PARCO. Advances in Parallel Computing, vol. 15, pp. 637–644. IOS Press, Amsterdam (2007)
4. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
5. Eranian, S.: Linux new monitoring interface: Performance Counter for Linux. In: CSCADS Workshop 2009 (2009) http://cscads.rice.edu/workshops/summer09/slides/performance-tools/cscads09-eranian.pdf
6. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: An open source infrastructure for parallel performance analysis. Sci. Program. 16, 105–121 (2008)
7. Brunst, H., Hackenberg, D., Juckeland, G., Rohling, H.: Comprehensive Performance Tracking with Vampir 7. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 17–29. Springer, Heidelberg (2010)

8. Hackenberg, D., Brunst, H., Nagel, W.: Event Tracing and Visualization for Cell Broadband Engine Systems. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 172–181. Springer, Heidelberg (2008)
9. Hackenberg, D., Juckeland, G., Brunst, H.: High resolution program flow visualization of hardware accelerated hybrid multi-core applications. IEEE International Symposium on Cluster Computing and the Grid, vol. 0, pp. 786–791 (2010)
10. Kluge, M., Nagel, W.: Analysis of Linux Scheduling with VAMPIR. In: Shi, Y., van Albada, G., Dongarra, J., Sloot, P. (eds.) ICCS 2007. LNCS, vol. 4488, pp. 823–830. Springer, Heidelberg (2007), doi:10.1007/978-3-540-72586-2_116