Redesigning PAPI's High-Level API

Frank Winkler

Innovative Computing Laboratory
The University of Tennessee
Knoxville, Tennessee, USA
frank.winkler@icl.utk.edu

Abstract. PAPI (Performance Application Programming Interface) provides a portable and efficient API to access the hardware performance counters found on modern microprocessors. With the introduction of Component PAPI or PAPI-C in early 2010 PAPI has extended its reach beyond the CPU and can now monitor system information across a range of components from CPUs to network cards, graphics accelerator cards, parallel file systems and more. To collect performance events, PAPI provides two APIs, the low-level and high-level API. The legacy high-level API was designed for simplicity, but could only handle preset CPU events. To access events from all installed components, the programmer had to use the low-level API. This paper introduces a new high-level API that enables the measurement of both preset and native events. It is intended for programmers who want to perform simple event measurements with minimal code instrumentation.

Keywords: Code Instrumentation, Performance Counter

1 Introduction

PAPI [3] [7] provides a simple interface, called the high-level (HL) API, for application developers who wish to perform direct instrumentation of their source code with minimal effort. The HL API that has existed in PAPI up to version 5.7, was implemented in response to demand from the community for something simpler than the low-level (LL) API. However, this original interface had a number of significant limitations. These limitations include:

- No support for native events
- Lack of support for measurements within threads
- No metadata support for the identification of measured sections
- Missing support for reporting measurement results

The new HL API overcomes those limitations and provides application developers the ability to record performance events of instrumented code sections, called regions, of serial, multi-processing (MPI, SHMEM) and thread (OpenMP, Pthreads) parallel applications. It is intended for programmers who want to perform simple event measurements in a very convenient way as they only have

to mark code sections. The remainder of this document is organized as follows: Section 2 compares the legacy HL API with the new HL API. In addition, scenarios are shown in which the use of the new HL API is more suitable than the LL API. Section 4 discusses overhead measurements performed with both the legacy and new HL API. Finally, this paper concludes with a summarization in Section 5 and gives a small outlook for future work in Section 6.

2 Comparison of the legacy and new high-level API

2.1 Legacy high-level API

The legacy HL API provided eight functions as shown in Table 1. A programmer could start, stop, and read counters for only CPU preset events which was a major limitation in comparison with the LL API. Figure 1 demonstrates the functions used to start, read, and stop counters. `PAPI_read_counters` and `PAPI_stop_counters` returned the current counts. Note that when the program called the function `PAPI_read_counters` the counters would be reset after being read. The determination of events to be recorded was part of the instrumentation procedure. If a programmer wanted to add or change events, the entire application had to be recompiled. The legacy HL API also provided a function for summing up events. However, event summation did not work for events from different threads. For the programmer's convenience the legacy HL API also offered functions for calculating derived metrics like *IPC*, *MFlops/s*, and *MFlips/s* as well as real and processor time without determining the required events. However, if a programmer wanted to record all derived metrics, he had to call all three functions causing additional overhead. It should be noted that these functions for derived metrics are now part of the LL API. Another major limitation of the legacy HL API is that the programmer had to implement the output of the measured results.

Table 1. Functions of the legacy high-level API

Function name	**Description**
`PAPI_start_counters`	start counting hardware events
`PAPI_read_counters`	copy current counts to array and reset counters
`PAPI_stop_counters`	stop counters and return current counts
`PAPI_accum_counters`	add current counts to array and reset counters
`PAPI_ipc`	gets instructions per cycle, real and processor time
`PAPI_flops`	simplified call to get Mflops/s (floating point operation rate), real and processor time
`PAPI_flips`	simplified call to get Mflips/s (floating point instruction rate), real and processor time
`PAPI_num_counters`	get the number of hardware counters available on the system

Fig. 1. Code snippet of the legacy high-level API in C

```c
#include <papi.h>
#define NUM_EVENTS 2

main()
{
  int Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};
  long_long values[NUM_EVENTS];
  int retval;

  retval = PAPI_start_counters(Events, NUM_EVENTS);
  if ( retval != PAPI_OK )
    handle_error(1);
  /* Do some computation here */
  retval = PAPI_read_counters(values, NUM_EVENTS);
  if ( retval != PAPI_OK )
    handle_error(1);
  /* Print results of values and continue computation */
  retval = PAPI_stop_counters(values, NUM_EVENTS);
  if ( retval != PAPI_OK )
    handle_error(1);
  /* Print results of values here */
}
```

Fig. 2. Code snippet of the new high-level API in C

```c
#include <papi.h>

int main()
{
  int retval;

  retval = PAPI_hl_region_begin("computation");
  if ( retval != PAPI_OK )
    handle_error(1);
  /* Do some computation here */
  retval = PAPI_hl_read("computation");
  if ( retval != PAPI_OK )
    handle_error(1);
  /* Continue computation here */
  retval = PAPI_hl_region_end("computation");
  if ( retval != PAPI_OK )
    handle_error(1);
}
```

2.2 New high-level API

The main goal of the new HL API is to make it easier to use. This means fewer functions but more functionality than the legacy HL API. The new HL API contains only four functions, as shown in Table 2. Using those functions a programmer can mark a code section as a "measurement region" and assign a name to this region. PAPI will automatically make measurements during the execution of that code section, and the measurements will be associated with the name given to the region. This is supported for both C and Fortran.

Table 2. Functions of the new high-level API

Function name	**Description**
`PAPI_hl_region_begin(const char*)`	read performance events at the beginning of a region (the first call also starts counting the events)
`PAPI_hl_read(const char*)`	read performance events inside of a region and store the difference to the corresponding beginning of the region
`PAPI_hl_region_end(const char*)`	read performance events at the end of a region and store the difference to the corresponding beginning of the region
`PAPI_hl_stop()`	stop a running high-level event set

A code section is identified by a unique region name and starts with `PAPI_hl_region_begin` and ends with `PAPI_hl_region_end`, as seen in Figure 2. The first call to `PAPI_hl_region_begin` starts counting the performance events. Those are read at the beginning and end of a code section, the latter stores the difference between the end and begin values. To get more detailed measurements, a programmer can insert several `PAPI_hl_read` calls inside a code section. Each read call stores the difference from the corresponding begin call. It should be noted that, unlike the legacy HL API, the `PAPI_hl_read` call of the new API does **not** reset the counters. Furthermore, `PAPI_hl_region_end` does not stop counting the performance events. Counting continues until the application terminates. Therefore, the programmer can also create nested regions if required. It is also possible to use the same region name for different code sections. In this case, the individual measurement results are added up. An example would be the use of a code section within a loop.

Contrary to the legacy HL API, events are not specified explicitly in the source code. Users specify the events to be counted (as a comma separated list) via an environment variable, as seen in Figure 3. PAPI automatically detects the corresponding components and also checks the availability of events and combinations. This is done in the first call of `PAPI_hl_region_begin`. If there are

events that are not supported due to hardware restrictions or typos they are ignored and PAPI only performs the measurements with the working events. If programmers want to measure instantaneous events like temperature or power consumption, they can specify this with the instant flag as seen in Figure 3. Instantaneous values are stored at `PAPI_hl_read` and `PAPI_hl_region_end` without computing the difference from the beginning of the region. Another benefit of the environment variable is that events can be changed without recompiling the entire application.

Fig. 3. Setting events via environment variable: PAPI_TOT_INS is specified as a delta event meaning that the difference between PAPI_hl_region_begin and PAPI_hl_[read|region_end] is stored. coretemp:::hwmon=instant is specified as an instantaneous event meaning that the current counter value is stored.

```
export PAPI_EVENTS="PAPI_TOT_INS,coretemp:::hwmon=instant"
```

Table 3. Default events of the new high-level API

Performance event
`perf::TASK-CLOCK`
`PAPI_TOT_INS`
`PAPI_TOT_CYC`
`PAPI_FP_INS or PAPI_VEC_SP or PAPI_VEC_DP`
`PAPI_FP_OPS or PAPI_SP_OPS or PAPI_DP_OPS`

If no events are specified by the programmer, PAPI will use a set of default events as seen in Table 3. Default events that are not available on the current machine, e.g. `PAPI_FP_OPS`, are automatically skipped. In the latter case PAPI tries to use `PAPI_SP_OPS` or `PAPI_DP_OPS`. If `PAPI_EVENTS` is set, default events are not recorded unless they are added to `PAPI_EVENTS`. Another special feature of the new HL API is the automatic library initialization. The first region call of `PAPI_hl_region_begin` takes care of the library initialization, even if it is a thread-parallel application. An output of the measured events is created automatically after the application exits. In the case of a serial, or a thread-parallel application there is only one output file. The output is generated in the current directory by default. However, it is recommended to specify an output directory for larger measurements, especially for MPI applications via the environment variable `PAPI_OUTPUT_DIRECTORY`. The output example in Figure 4 was generated with default events and shows performance events for the region "computation" (see code snipped in Figure 2) in JSON format. As it is a serial application there is only one thread containing performance events. In case of a thread-parallel application there would be JSON objects for each thread. MPI applications would be saved in multiple files, one per MPI rank. In the case

where measurements are performed, while there are old measurements in the same directory, the HL library will not overwrite or delete the old measurement directories. Instead, timestamps are added to the old directories. For more convenience, the output can also be printed to stdout by setting `PAPI_REPORT=1`. This is not recommended for MPI applications as each MPI rank tries to print the output concurrently.

Another pratical feature of the new hl API is the support of multiplexing. Multiplexing allows a user to count more events than total physical counters by time sharing the existing counters at some loss in precision. This occurs only for CPU core events and can be enabled via the environment variable `PAPI_MULTIPLEX`. However, the programmer can only activate multiplexing for the entire application run and not only for a specific code section. Beyond that, it is not possible to select specific event groupings for multiplexing. Figure 4 shows an overview of environment variables that are used by the new HL API.

Table 4. Environment variables of the new high-level API

Environment variable	**Description**
`PAPI_EVENTS`	Performance events to measure
`PAPI_MULTIPLEX`	Enable Multiplexing
`PAPI_REPORT`	Print performance report to stdout
`PAPI_OUTPUT_DIRECTORY`	Path of the measurement directory
`PAPI_HL_VERBOSE`	Enables warnings and info
`PAPI_DEBUG=HIGHLEVEL`	Enable debugging of high-level routines

Similar to the legacy HL API, the new HL API also provides measurement results for derived metrics like *IPC*, *MFlops/s*, and *MFlips/s* as well as real and processor time when using default events or the required events for a derived metric. In order to obtain those derived results, the python script `papi_hl_output_writer.py` has to be used with the generated output (see Figure 4). The legacy HL API provided a function `PAPI_accum_counters` that accumulated the current value to the last stored value. Using this function, a user were able to sum up all events recorded within the same thread. The new HL API can summarize performance events over all threads and MPI ranks when using the option "accumulate" for the python script `papi_hl_output_writer.py` as seen in Figure 5.

The new HL API is not only characterized by its very simple usability, but also by its robustness. Several strategies have been considered to ensure that the application is disturbed as little as possible when measuring performance data. No faults should occur in an application due to improper use of the HL functions by the programmer. If a region is not completely marked, e.g. the beginning or the end of a region is missing, the application will continue. The programmer gets a warning and PAPI cleans up all internal data structures in order not to

disturb the remaining runtime of the application. The latter can happen if the programmer marks a region where the begin part is inside and the end part is outside of a parallel region. It is therefore the responsibility of the programmer to make sure that a matching region is in the same thread.

In order to mix the HL and LL API, the programmer must call `PAPI_hl_stop` when LL calls are used after a marked region. Also note that the `PAPI_hl_stop` call must be in the same thread as the marked region. In case the LL and HL API is using an event set where one event is the same, the programmer must stop the LL running event set before continuing with the HL API.

Fig. 4. JSON output of the new high-level API for a serial application

```
{
  "cpu␣in␣mhz":"1995",
  "threads":[
    {
      "id":"0",
      "regions":[
        {
          "computation":{
            "region_count":"1",
            "cycles":"2080863768",
            "perf::TASK-CLOCK":"1042308865",
            "PAPI_TOT_INS":"2917520595",
            "PAPI_TOT_CYC":"2064112930",
            "PAPI_FP_INS":"375785927",
            "PAPI_FP_OPS":"375787554"
       ...
}
```

Fig. 5. Accumulated JSON output of the new high-level API for a serial application with derived metrics

```
python papi_hl_output_writer.py --type=accumulate
{
  "computation": {
    "Region␣count": 1,
    "Real␣time␣in␣s": 0.97,
    "CPU␣time␣in␣s": 0.98,
    "IPC": 1.41,
    "MFLIPS/s": 386.28,
    "MFLOPS/s": 386.28,
    "Number␣of␣ranks": 1,
    "Number␣of␣threads": 1,
    "Number␣of␣processes": 1
  }
}
```

2.3 Additional experimental functions of the new high-level API

The new HL API also offers optional functions (see Figure 5) that allow the programmer to control initialization and finalization, determine events in the source code, and trigger the performance report. It is useful for very complex applications where performance measurements are only required for a small code section. However, the new HL API was designed with simplicity in mind. For this reason, all advanced functions are only available in a feature branch. It should be noted that the feature branch is in an experimental state. Using the optional functions may cause unwanted side effects. `PAPI_hl_init` initializes the PAPI library and some HL specific features. As mentioned in section 2.2 the first call of `PAPI_hl_region_begin` automatically calls `PAPI_hl_init` if not already called. `PAPI_hl_set_events` offers the programmer the possibility to determine events in the source code as an alternative to the environment variable `PAPI_EVENTS`. The content of `PAPI_EVENTS` is ignored if `PAPI_hl_set_events` was successfully executed. It should also be noted that `PAPI_hl_set_events` has to be called before the first `PAPI_hl_region_begin` call. It is also possible to disable the performance measurement of an instrumented application by setting `PAPI_EVENTS=NONE`. This feature is very useful when the measurement overhead is to be determined. `PAPI_hl_finalize` stops all running event sets, destroys them, and cleans up internal data structures. All subsequent HL function calls are ignored after `PAPI_hl_finalize` has been called. However, this only works for serial applications (or SMP applications that do not use threads), since the thread that calls `PAPI_hl_finalize` does not have access to other threads that still have running event sets. One solution is the function `PAPI_hl_cleanup_thread` that has to be called at the end of each thread. But this only works for Pthreads or parallel OpenMP regions without the "`parallel for`" pragma. In the latter case, `PAPI_hl_cleanup_thread` can be called at the end of the last parallel region. When using "`parallel for`" pragmas, the clean up call only works if the number of threads equals the number of loop iterations.

Table 5. Experimental functions of the new high-level API

Function name	**Description**
`PAPI_hl_init()`	initialize the high-level interface
`PAPI_hl_cleanup_thread()`	stop running events of a thread and clean up all local data structures
`PAPI_hl_finalize()`	stop running events of the master thread, clean up all global data structures and shut down the high-level interface
`PAPI_hl_set_events(const char* events)`	set specific events to be recorded
`PAPI_hl_print_output()`	generate performance report

3 Low-level versus high-level API

The LL API manages performance events in user-defined groups, called event sets. It is aimed at experienced application programmers and tool developers who require fine-grained measurement and control of the PAPI interface. It provides access to both PAPI presets and native events and supports all installed components. Several performance tools like Score-P [5], Vampir [2], TAU [6], and HPCToolkit [1] support PAPI and in fact use the LL API. In order to support all installed components, each performance tool has to create the required event sets for each component.

Figure 6 shows a small example of measuring events from different components. First the PAPI library must be initialized. In this example two events from different components are to be measured. Each component requires a separate event set, so two event sets must be created. After that, events have to be added to the corresponding event set. The first event `PAPI_TOT_INS` is preset and can be added to the event set easily using the function `PAPI_add_event`. To save space, the return check is omitted for all subsequent PAPI functions. The second event `appio:::READ_BYTES` is provided by the component `appio`. Here, the event name must first be converted into an event code. The event code can then be added to the second event set. Similar to the legacy HL API, the measurement can now be started with `PAPI_start`. However, each event set has to be started separately. This also applies to reading and stopping the event sets. The example from Figure 2 can produce the same results with the new HL API as event sets are generated automatically. Using the new HL API, the events `PAPI_TOT_INS` and `appio:::READ_BYTES` can both be added to the environment variable `PAPI_EVENTS`, although they belong to different PAPI components. Handling multiple components and multiple event sets is taken care implicitly by PAPI. This shows the added value of the new HL API over the low-level API, as it greatly simplifies performance measurements with different components.

Performance tools that provide a simple tools interface for code sections instrumentation can easily use the new HL API. Those tools do not have to worry about the actual performance measurement and performance report as everything is done by PAPI's HL functions. The programming model Kokkos [4] provides a callback based tools interface[1] which is very well suited for the new HL API. A PAPI connector for Kokkos has already been implemented and can be found in the official Kokkos-tools repository[2].

It should also be noted that the new HL API can be used in conjunction with the LL API as long as they do not use the same event sets or components at the same time. One possible scenario could be a complex parallel application with offloading using CUDA. The HL API could be used to measure performance events of host functions using perf events. For a more in-depth analysis such as starting and stopping events to identify a problematic section of a CUDA

[1] `https://github.com/kokkos/kokkos-tools/wiki/Profiling-Hooks`
[2] `https://github.com/kokkos/kokkos-tools`

function, the low-level API together with the CUDA component would be the better choice.

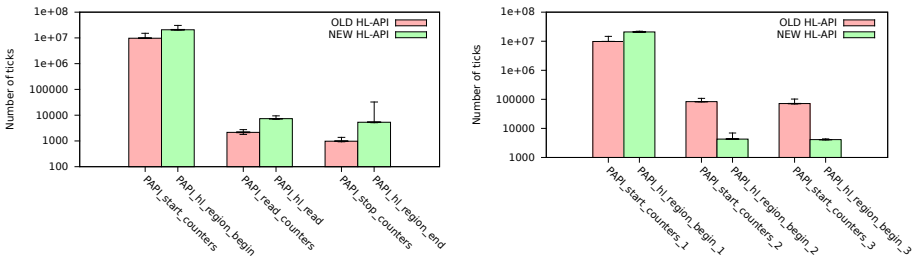
Fig. 6. Code snippet of the low-level API in C: This example is measuring the events PAPI_TOT_INS and appio:::READ_BYTES from two components (perf, appio). For readability, the return check for the most PAPI functions and the output implementation for the measurement results are omitted. It should be noted that the new high-level example from Figure 2 can produce the same measurement results.

```c
#include <papi.h>
#define NUM_EVENTS 2

main()
{
  int i;
  int retval;
  unsigned int native = 0x0;
  int EventSet[NUM_EVENTS];
  long_long values[NUM_EVENTS][1];

  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT)
    handle_error (1);

  for (i = 0; i < NUM_EVENTS; i++) {
    EventSet[i] = PAPI_NULL;
    PAPI_create_eventset(&EventSet[i]);
  }

  PAPI_add_event(EventSet[0], PAPI_TOT_INS);
  PAPI_event_name_to_code("appio:::READ_BYTES", &native);
  PAPI_add_event(EventSet[1], native);

  for (i = 0; i < NUM_EVENTS; i++)
    PAPI_start(EventSet[i]);

  /* Do some computation here */

  for (i = 0; i < NUM_EVENTS; i++)
    PAPI_read(EventSet[i], values[i][0]);

  /* Print results of values and continue computation */

  for (i = 0; i < NUM_EVENTS; i++)
    PAPI_stop(EventSet[i], values[i][0]);

  /* Print results of values here */

}
```

4 Overhead measurements

To compare the overhead of the new HL API with the legacy one, start/begin, read, and stop/end routines were measured, as seen in Figure 7. The measurements were performed with one hardware event on a Xeon(R) CPU[3]. Since the overhead of the routines is extremely low, CPU cycles were counted. With a CPU speed of 2 GHz, one can consider about $2 * 10^9$ ticks per second. The first call of the start/begin routine uses the most cycles, $9.7 * 10^6$ ticks (4,85 ms) for `PAPI_start_counters` and $2*10^7$ ticks (10 ms) for `PAPI_hl_region_begin`. Besides initializing the HL interface, creating and starting the event sets and reading the counters, the first call of `PAPI_hl_region_begin` also reads and tests all events to be recorded from the environment variable, creates event sets for the corresponding components and stores the read values in the internal data structure. This explains the overhead of $5ms$ compared to `PAPI_start_counters`. Figure 7 also clearly shows that recurring calls of `PAPI_hl_region_begin` are slightly faster (about 80000 cycles) than recurring calls of `PAPI_start_counters`. In contrast to `PAPI_start_counters` that must restart the event set on each call, the second call of `PAPI_hl_region_begin` only reads the counters and stores them. The read and end routines of the new HL API also consume more cycles than the read and stop routines of the legacy HL API as they need to store the difference to the corresponding begin values.

Fig. 7. Overhead comparison of legacy and new high-level API: The graphic on the left compares the start/begin, read, and stop/end routines for the legacy and new high-level API. The new API has a slightly higher overhead (about $5ms$), which is mainly due to the automatic storage of the counters. The graphic on the right shows successive calls of the start/begin routines. While the first PAPI_hl_region_begin call takes much more time due to creation and starting of the event sets, the following calls are significantly faster as they only read counters and store them.



[3] Xeon(R) CPU E5-2650 v3

5 Conclusions

PAPI's new HL API lets users record performance events of instrumented regions of serial, multi-processing and thread-parallel applications. It is designed for simplicity, while offering some flexibility. Events to be recorded are determined via an environment variable that lists both preset and native events separated by commas. This enables users to measure different events in successive runs of their application without recompiling. In addition, users do not need to take care of printing performance events since an output is generated at the end of each measurement. Some of the benefits of using the HL API rather than the LL API are that it is easier to use and requires less setup. For instance, the dynamic setting of performance events via the environment variable and the automatic detection of components makes the use of the HL API extremely simple. It should also be noted that the new HL API can be used in conjunction with the LL API and, in fact, does call the LL API internally.

6 Future Work

The new HL API enables a very simple code instrumentation for thread-parallel applications. However, all measurements must be trigged from each thread. If a programmer instruments a code section around a parallel region, PAPI will only collect performance events from the master thread. The current HL API requires code instrumentation inside a parallel region to record performance events from all threads. This can be an important limitation for some users. To avoid this limitation, mechanisms for automatic thread detection must be implemented. This could also ensure a clean finalization of PAPI for thread-parallel applications via the function call `PAPI_hl_finalize`. There are different concepts to detect thread creation and destruction, for example using library wrapping for Pthreads or OMPT[4] for OpenMP. These concepts will be investigated and possibly used in PAPI++, the successor of PAPI. Furthermore, the new HL API will be reimplemented for PAPI++ and will then provide more user control using C++ features. For example, default function arguments could be used for the region instrumentation to address a specific event set or all event sets per default.

List of Acronyms

HL high-level
LL low-level
API Application Programming Interface
PAPI Performance Application Programming Interface

[4] OMPT: Tools interface in the OpenMP spec

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. Concurrency and Computation: Practice and Experience (2010)
2. Andreas Knüpfer and others: The Vampir Performance Analysis Tool-Set. In: Proc. of the 2nd Int. Workshop on Parallel Tools for High Performance Computing (2008)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications 14(3), 189–204 (2000), https://doi.org/10.1177/109434200001400303
4. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing 74(12), 3202 – 3216 (2014), http://www.sciencedirect.com/science/article/pii/S0743731514001257, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
5. Mey, D.a., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Competence in High Performance Computing (2012)
6. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (May 2006), http://dx.doi.org/10.1177/1094342006064482
7. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with papi-c. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009. pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)