PAPI++ WORKING NOTES

2

Roadmap for Refactoring classic PAPI to PAPI++

Part II: Formulation of Roadmap based on Survey Results

Heike Jagode
Anthony Danalis
Damien Genet

Innovative Computing Laboratory (ICL)

July 31, 2020

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Revision	**Notes**
07-2020	first publication

```
@techreport{2020-07-exa-papi,
  author={Jagode, Heike and Danalis, Anthony and Genet, Damien},
  title={Roadmap for Refactoring classic PAPI to {PAPI++}, {PDN} No. 2},
  institution={Innovative Computing Laboratory, University of Tennessee Knoxville},
  year={2020},
  month={July},
  number={ICL-UT-20-07},
}
```

Contents

CHAPTER 1

Introduction

The Exa-PAPI project [1] is developing a new C++ Performance API (PAPI++) software package from the ground up that offers a standard interface and methodology for using low-level performance counters in CPUs, GPUs, on/off-chip memory, interconnects, and the I/O system, including energy/power management. PAPI++ is building upon classic-PAPI functionality and strengthening its path to exascale with a more efficient and flexible software design, one that takes advantage of C++ object-oriented nature but preserves the low-overhead monitoring of performance counters and adds a vast testing suite.

To put the new PAPI++ plan into perspective, the first PAPI version that offered a standardized, easy-to-use interface for accessing hardware performance counters was released 21 years ago in 1999. The past two decades witnessed tectonic shifts in hardware technology followed by paradigm shifts in software technology. During that time, PAPI has been repeatedly"extended" with performance counter support for newly released CPUs; "redesigned" to enable hardware monitoring information that became available in other sub-systems throughout modern computer architectures (e.g., counters found in GPUs, on/off-chip memory, interconnects, I/O systems); and "upgraded" to extend PAPI's role further for monitoring and capping power consumption as well as performance events that originate from other software layers.

No viable replacement for PAPI has emerged and established itself as the de facto standard for monitoring hardware counters, power usage, software-defined events, and channeling this technological progress into a robust software package. The PAPI++ package is meant to be this replacement—with a more flexible and sustainable software design. The objective of this white paper is to describe the current design of the "classic" PAPI framework in conjunction with its sustainability challenges, as well as identify and articulate opportunities and possible solutions to implement PAPI++ to remain tenable and useful for the Exascale era—and beyond.

[1] https://icl.utk.edu/exa-papi/

CHAPTER 2

Classic PAPI Design and Limitations

The first "PAPI++ Working Notes" [1] summarized the results from a PAPI survey that the Exa-PAPI team circulated to the Exascale Computing Project (ECP) applications (AD) and software technology (ST) teams to assess their needs and requirements for hardware and software performance counter functionality. Based on these findings, the Exa-PAPI team has come up with a roadmap for refactoring "classic" PAPI functionality into a new PAPI++ software package.

The first part of this document describes the classic PAPI design and its current limitations. Modifying and extending a library with a broad user base such as PAPI requires care to preserve simplicity and backward compatibility as much as possible while providing clean and intuitive access to important new capabilities. In the second part of this document, we discuss possible implementations and C++ features for PAPI++ to provide support for the simultaneous measurement of data from multiple counter domains.

2.1 PAPI Framework

Several software design issues became apparent in extending the classic PAPI framework for multiple measurement domains. The classic PAPI library consists of two internal layers: a large portable layer optimized for platform independence; and a smaller hardware specific layer, containing platform dependent code. By compiling and statically linking the independent layer with the hardware specific layer, an instance of the PAPI library could be produced for a specific operating system and hardware architecture. At compile time the hardware specific layer provided common data structure sizes and definitions to the independent layer, and at link time it satisfied unresolved function references across the layers. Since there was a one-to-one relationship between the independent layer and the hardware specific layer, initialization and shutdown logic was straightforward, and control and query routines could be directly implemented. In migrating to a multi-component model, this one-to-one relationship was replaced with a one-to-many coupling between the independent, or framework, layer and a collection of hardware specific components, requiring that previous code dependencies and assumptions be carefully identified and modified as necessary.

When linking multiple components into a common object library, each component exposes a subset of the same functionality to the framework layer. To avoid name-space collisions in the linker, the entry points of each component are modified to hide the function names, either by giving them names unique to the component, or by declaring them as static inside the component code. Each component contains an instance of a structure, or vector, with all the necessary information about opaque structure sizes, component specific initializations and function pointers for each of the functions that had been previously statically linked across the framework/component boundary. The only symbol that a component exposes to the framework at link time is this uniquely named component vector. All accesses to the component code occur through function pointers in this vector, and vector pointers that are not explicitly set by a component default to placeholder functions that fail gracefully, allowing components to be implemented with only a subset of the complete functionality. In this way, the framework can transparently manage initialization of and access to multiple components by iterating across a list of all available component structures. This extra level of indirection introduced by calls through a function pointer adds a small but generally negligible additional overhead to the call time, even in time-critical routines such as reading counter values (`PAPI_read()`, `PAPI_stop()`).

2.2 PAPI Events

Countable events in PAPI are either preset events, defined uniformly across all CPU architectures, or native events, unique to a specific component. To date, preset events have only been defined for processor hardware counters, making all events on off-processor components native events.

2.2.1 Preset Events

Preset events can be defined as a single event native to a given CPU, or can be derived as a linear combination of native events, such as the sum or difference of two such events. More complex derived combinations of events can be expressed in reverse Polish notation and computed at run-time by PAPI. The number of unique terms in these expressions is limited by the number of counters in the hardware. For many platforms the preset event definitions are provided in a comma separated values file, `papi_events.csv`, which can be modified by developers to explore novel or alternate definitions of preset events. Because not all preset events are implemented on all platforms, a utility called `papi_avail` is provided to examine the list of preset events on the platform of interest.

Currently, only the raw native events from the PAPI CPU components can be used to derive generic PAPI preset events, such as `PAPI_FP_OPS`, `PAPI_L1_MISS`. For PAPI++, this limitation will be removed and we will redesign the internal PAPI framework so that other components, such as the Nvidia and AMD GPU components, can be extended with preset event support. Specifically, the adoption of floating-point operations (FLOP) presets for the GPU components (currently, Nvidia and AMD, later also Intel) have a high priority in the HPC community, and we will start this effort by investigating and analyzing the use of PAPI high- and low-precision FLOP counters on Nvidia GPUs in order to develop a mapping of AMD FLOP events to Cuda FLOP events. Additional effort will focus on the development of presets for a one-to-one mapping between Nvidia and AMD events related to cache and memory activities, such as misses, hits, accesses.

2.2.2 Native Events

PAPI components contain tables of native event information allowing native events to be programmed in essentially the same way as a preset event. Each native event may have a number of attributes, called unit masks, that can act as filters on exactly what gets counted. These attributes can be appended to a native event name to tell PAPI exactly what to count. An example of a native event name with unit masks from the Intel Haswell EP architecture is shown below:

```
OFFCORE_RESPONSE_0:DMND_DATA_RD:DMND_RFO:DMND_IFETCH:PF_DATA_RD:PF_RFO:PF_L3_DATA_RD:
PF_L3_RFO:NO_SUPP:L3_MISS:SNP_NONE:SNP_NOT_NEEDED:SNP_MISS:SNP_NO_FWD
```

Attributes can be appended in any order and combination, and are separated by colon characters. Some components such as LM-SENSORS may have hierarchically defined native events. An example of such a hierarchy is shown below:

```
LM_SENSORS.max1617-i2c-0-18.temp2.temp2_input
```

In this case, levels of the hierarchy are separated by period characters. Complete listings of these and other native events can be obtained from a utility analogous to `papi_avail`, called `papi_native_avail`.

2.3 PAPI Components and EventSets

An important consideration in extending a widely accepted interface such as PAPI is to make extensions in such a way as to preserve the original interface as much as possible for the sake of backward compatibility. Several entry points in the PAPI user API were augmented to support multiple components, and several new entry points were added to support new functionality. By convention, an event to be counted is added to a collection of events in an EventSet, and EventSets are started, stopped, and read to produce event count values. Each EventSet in Component PAPI is bound to a specific component and can only contain events associated with that component. Multiple EventSets can be active simultaneously, as long as only one EventSet per component is invoked.

The binding of EventSet and component has become too limiting and restrictive considering the number of PAPI components (more than 30) available today. For PAPI++, we will provide a way to combine different types of performance counters from different architectures (and hence, PAPI components) in the same EventSet.

2.3.1 The PAPI CPU Component: Not your typical “Component”

The CPU component is unique for several reasons. Historically it was the only component that existed in earlier versions of PAPI. Within Component PAPI one and only one CPU component must exist and occupy the first position in the array of components. This simplifies default behavior for legacy applications. In addition to providing access to the hardware counters on the main processor in the system, the CPU component also provides the operating system specific interface for things like interrupts and threading support, as well as high resolution time bases used by the PAPI Framework layer.

We consider the necessity for the existence of a CPU component—and only one CPU component—to be an undesirable restriction. If one considers implementations that may not need, or wish, to monitor the CPU activity, and also implementations that may contain heterogeneous CPUs, it becomes clear why enforcing the existence of one, and only one, CPU component in every PAPI installation is undesirable. Eliminating this restriction is an open research issue in PAPI++ and we are investigating various mechanisms for making the existence of CPU components optional.

2.3.2 Accessing the CPU Hardware Counters

CPU Hardware counter access is provided in a variety of ways on different systems. When PAPI was first released 21 years ago, there was significant diversity in the operating systems and hardware of the Top500 list [2]. AIX, Solaris, UNICOS and IRIX shared the list with a number of variants of Unix. Linux systems made up a mere 3.6% of the list. Most of these systems had vendor provided support for counter access either built-in to the operating system, or available as a loadable driver. The exception was Linux, which had no support for hardware counter access. This is in sharp contrast to today, where 100% of the Top500 systems run Linux variants.

Several options were available to access counters on Linux systems. One of the earliest was the perfctr patch [3] for x86 processors. Perfctr provided a low latency memory-mapped interface to virtualized 64-bit counters on a per process or per thread basis, ideal for PAPI’s “first person” counting and sampling interface. With the introduction of Linux on the Itanium processor, the perfmon [4] interface was built-in to the kernel. When it became apparent that perfctr would not be accepted into the Linux kernel, perfmon was rewritten and generalized as perfmon2 [5] to support a wide range of processors under Linux, including the IBM POWER series in addition to x86 and IA64 architectures. After a continuing effort over several years by the performance community to get perfmon2 accepted into the Linux kernel, it too was rejected and supplanted by yet another abstraction of the hardware counters, first called `perf_counters` in kernel 2.6.31 and then `perf_events` [6] in kernel 2.6.32.

Nowadays, the `perf_events` interface is mature and has the advantage of being built-in to the kernel, requiring no patching on the part of system administrators. Nonetheless, classic PAPI still has support for hardware counter access through perfctr, perfmon, and various other substrates. All these CPU-specific substrates are not only outdated, but also deeply intertwined with the CPU component and the architecture-independent part of the PAPI framework. In PAPI++, obsolete substrate support will be removed, and the `perf_events` support will be completely detached from the framework allowing users to configure PAPI without the CPU component.

2.4 Additional Limitations

This section summarizes additional issues that are under investigation for PAPI++.

Data Types: PAPI supports returned data values expressed as unsigned 64-bit integers. This is appropriate for counting events, but is not as appropriate for expressing other values. We are exploring ways to encode and specify other 64-bit data formats including: signed integer, IEEE double precision, fixed point, and integer ratios.

Dynamic Configurability: The current mechanism for adding new components is not well suited to introducing new components. Methods are needed for an automated discovery process for components, both at build time and at execution time.

Synchronization: Components can report values with widely different time scales and remote measurements may exhibit significant skew and drift in time from local measurements. Mechanisms need to be developed to accommodate these artifacts.

CHAPTER 3

Software Engineering with C++

Historically speaking, the PAPI framework has been implemented in C and also provides a Fortran API. The complexity of modern hardware and software systems for HPC, however, necessitates the use of modern programming languages to ease the development process, avoid code repetition, and keep the volume of code that's exposed to changing requirements as minimal as possible. While there is no question of the robustness of procedural programming languages such as C and modern Fortran, when developing a new library from the ground up, modern software engineering demands generic programming, data abstraction and encapsulation, inheritance, to name but a few, all of which can be easily expressed with C++. It is only natural for the development of PAPI++ to adopted C++ as implementation language to leverage its support for object-oriented programming.

3.1 Overcoming PAPI Limitations with C++ Features

As for the language specification, for PAPI++ we are targeting C++11 or newer, as it introduces many new features, such as built-in atomic support, and is completely supported by the GNU, Intel, and LLVM compilers.

3.1.1 Overloading

C++ allows multiple functions to have the same name as long as they are distinct by their signatures (i.e. argument types, number of arguments a function uses). For instance, a single `PAPI_read()` function with versions for reading counter values that are `long long int`, `double`, `string`, etc., instead of multiple type-variants, e.g., `PAPI_read_longlong()`, `PAPI_read_double()`, etc. This is crucial for templating, as all function calls must be generic.

3.1.2 Generics a.k.a. Templates

C++ templates reduce the complexity of programming by implementing a routine once for a generic type, which can then be automatically instantiated for specific types such as `long long int`, `double`, `float` or `string` counter values. Existing PAPI currently is limited to `long long int` counter values. Extending classic PAPI to go beyond the notion of a simple counter by allowing arbitrary information to be exported by performance counters would involve either hand coding various versions (`long long int`, `double`, `string`, etc.) of each `PAPI_read()`, `PAPI_stop()`, `PAPI_accum()`, `PAPI_write()` routine, or coding a search-and-replace script to crudely automate the conversion to other precisions. Templates fully automate this process and ensure type safety.

3.1.3 Exceptions

Traditional PAPI relies on returning an info parameter with an error code, which, typically, is inherited from the underlying interface (i.e. the Linux kernel, or various other interfaces such as Powercap, Lmsensors, BGPM, Emon, RAPL, PCP, CUPTI, ROCm, NVML, to name but a few of what is supported in PAPI). C++ allows throwing exceptions, which a parent context can catch. This can simplify error checking by grouping all the error checks together.

Exceptions also prevent ignored errors, as often happens with returned error codes—the exception must be caught somewhere, or it will propagate all the way up to main. The existing PAPI C code often misses error handling after every function call in the component implementations.

However, excessive use of exceptions can make the use of a library cumbersome from the point of view of an application developer. In PAPI++ we will ensure that errors that must not be ignored are propagated to the caller as exceptions, without generating exceptions for issues that are of secondary importance.

3.1.4 Value Initialization

Value initialization—which is distinct from default-initialization— is a very commonly used feature that allows, among other things, providing a default constructor for user-defined objects. This will be extremely useful for the new PAPI component, EventSet, and event objects. Specifically, the C++11 "brace syntax" is particularly valuable for function templates `template <typename T>`, where T can be of different types. The brace feature `T temp {};` allows for safe value initialization of `temp` regardless of what type T is.

3.1.5 User-defined Data Types: Event and EventSet

In classic PAPI, there is a tight correspondence between EventSets and components. In Section 2.3, we explain in-depth the consequences of this design decision. In summary, this leads to the following two extremely critical limitations for an application or performance tool using PAPI:

(1) An EventSet can only contain events that belong to a single component.

(2) Only one EventSet per component can be active at any given time.

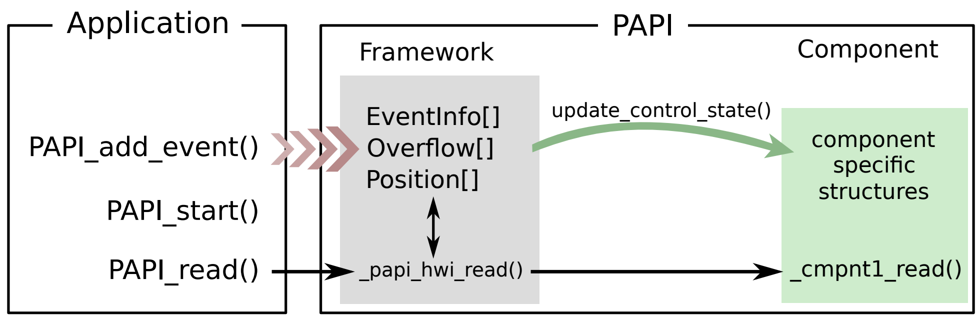

Figure 3.1: Application-Framework-Component interaction in classic PAPI

The interaction between an application, the PAPI framework, and a component with respect to events and EventSets is depicted graphically in Figure 3.1. As shown in the figure, when an application adds an event to an EventSet the framework updates several arrays that contain information about the event, details that pertain to overflowing, and indirection arrays. Then, the framework invokes a function (`update_control_state()`) of the component that is responsible for the newly added event, so that the component may update its own specific structures. Later, when the application invokes `PAPI_read()`, the framework calls the read function of the corresponding component which returns the values of all the counters in the EventSet (since they all belong to the same component).

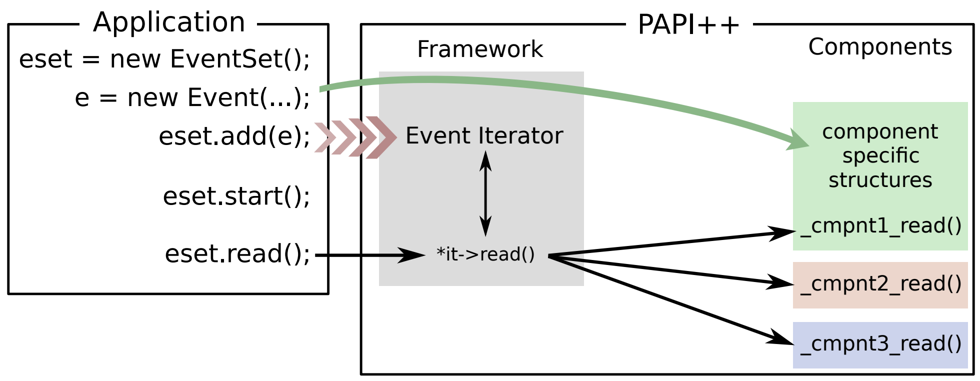

Figure 3.2: Application-Framework-Component interaction in PAPI++

In PAPI++ we aim to free users from the tight correspondence between EventSets and components, and the limitations that come with it. Figure 3.2 expresses this increased flexibility using pseudo-C++ constructs. In PAPI++, “Event” and “EventSet” will be user-defined data types (classes). In classic PAPI, functions such as `PAPI_start()`, `PAPI_read()` and `PAPI_stop()` start and stop the monitoring of an EventSet, and functions such as `PAPI_overflow()` manipulate properties of the EventSet that affect its behavior. In the PAPI++ design, these will be member functions of the new classes, and some of them will exist in both the EventSet class and the

Event class, so that decisions can be delegated to the most appropriate entity. For example, the EventSet does not need to know which component each event belongs to because the individual events themselves are aware of their component origin.

Since each event is an object of the class Event, component-specific information can be updated at event creation (in the constructor of the class). Then, adding the event to an EventSet will insert it in a data-structure with an iterator. When the application tries to read the values associated with an EventSet it calls EventSet.read() only once, and internally the content of the iterator is traversed and for each event the framework calls the member function `read()` of that event. Since each event is uniquely associated with a component, the Event.read() function will automatically call the read() function of the proper component. This design allows for the different events of an EventSet to be able to belong to arbitrary components. Also, this design allows for different events to have values of different types (`long long int`, `double`, etc.), or even values that are non-primitive types—such as strings, or arrays of data—without requiring cumbersome APIs that include the type name in the function name, or non-safe practices like pointer casting between variables of different types.

Several decisions regarding the PAPI++ API remain open and are the subject of active investigation by the Exa-PAPI team. For instance, Event.read() may or may not be called directly form the application layer. On one hand, it may allow for more fine-grained control by the application. On the other hand, however, it prevents optimizations that can be performed if events are always grouped in an EventSet. Furthermore, classes such as "Component"—which we will need to implement to avoid duplicated code between individual components—and classes implemented to handle architecture-dependent details, appear to have no reason to be publicly exposed to the application layer and will likely remain private to PAPI++.

3.1.6 Classic PAPI Interfaces and Performance Overhead

The C++ example given in the previous section seemingly breaks the existing C and Fortran interface of classic PAPI (e.g., `PAPI_add_event()`, `PAPI_read()`); yet, backward compatibility with the classic PAPI interfaces is of paramount importance to PAPI++ and will be maintained. To achieve this, we will provide wrapper C and Fortran functions that will have the same signature and behavior as the functions of classic PAPI. However, new and more advanced features that rely on C++ functionality—such as events that return results of different types—might require cumbersome C APIs, or might be omitted from future C/Fortran interfaces.

Another major concern for PAPI++ is the potential performance overhead caused by the new design and increased level of abstraction offered by C++. To avoid design decisions that would lead to excessive overhead, we will prototype several alternative designs (with varying mixtures of C and C++ code) early on, and we will investigate their performance overhead so that we can balance abstraction and performance before we commit to a final design we will adopt for PAPI++.

Bibliography

[1] Heike Jagode, Anthony Danalis, and Jack Dongarra. Formulation of requirements for new PAPI++ software package, PDN no. 1. Technical Report ICL-UT-20-02, Innovative Computing Laboratory, University of Tennessee Knoxville, January 2020.

[2] Top500. Operating system share, November 1999. https://www.top500.org/statistics/list/, 1999.

[3] Mikael Pettersson. Power. http://user.it.uu.se/~mikpe/linux/perfctr/, 2003.

[4] Perfmon2. Sourceforge project page. http://perfmon2.sourceforge.net, 2005.

[5] S Jarp, R Jurga, and A Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119(4):042017, jul 2008. doi: 10.1088/1742-6596/119/4/042017. URL https://iopscience.iop.org/article/10.1088/1742-6596/119/4/042017.

[6] Ingo Molnar. Performance Counters for Linux, v8. http://lwn.net/Articles/336542, 2009.