Proposed Features for the PAPI 6.0 Release

Vince Weaver

vincent.weaver@maine.edu

30 June 2014

Abstract

The PAPI 6.0 release will be a major overhaul of the PAPI interface. We attempt to preserve backwards compatibility as much as possible, but some code that uses PAPI may break in the transition from PAPI 5.x to PAPI 6.

This document describes the major changes proposed in PAPI 6.0 and gives information on how to properly migrate code to the new interfaces.

Contents

1 Background

The PAPI 6.0 release enhances the PAPI library while removing some of the limitations in the previous PAPI 5.x API/ABI. PAPI 5.x was a major overhaul compared to the previous 4.x release, see our previous paper [1] for changes in that release.

Many of the changes break the ABI, meaning that binaries linked against 5.x versions of PAPI will not link against the new version. That is why we bumped the major version number to 6.0.

We attempted to minimize the API breakage. This means that hopefully any code that compiled against PAPI 5.x will compile against 6.0 with no changes. See Section 1.1 for information on changes that might break the API.

1.1 Changes that may break the API

We attempt to remain as backwards compatible as possible. Most code that worked with PAPI 5.x should work with PAPI 6.0 with no code changes.

If your code does break, please report this to the PAPI mailing list and we will do our best to address the problem.

1.2 Changes that break components

We may need to make changes to the CDI to address limitations in the PAPI 5.x interface.

2 Proposed Features for PAPI6

Not all of these are realistic. This is just a list of things that might be nice to have.

2.1 System-Wide 3rd Person Measurements

On modern systems performance measurement has become less about first person (per-process) measurement and more about system-wide third person measurement. Various improvements need to be made before PAPI will support this well.

2.1.1 Improved Uncore/Northbridge Support

PAPI does not support this well.

2.1.2 Default Core for Per-Package Measurements

Currently you have to set a package with `PAPI_set_opt(PAPI_CPU_ATTACH)`, even on systems that only have one package.

We should default to the first CPU, but should document this well so users are not confused.

2.1.3 RAPL component Improvements

Optionally use the new perf_event interface instead of the rapl component that has lots of security and overflow issues.

Tricky, as the interface is the same as regular perf_event and events may show up under the perf_event component.

2.1.4 Other Energy/Power modules

AMD has "Application Power Management" which is a lot like Intel RAPL. The Linux kernel already supports this, but it appears as a hwmon (so in PAPI lmsensors component) rather than under perf_event. (AMD support seems limited to server machine, fam15h model 0-15. I have a fam15h model 19 that does not have support).

Various ARM processors also have support in a similar fashion.

Can we somehow make a unified driver? A separate APM drive?

Should we create power presets? `PAPI_PKG_POWER`? Should they have umasks? `PAPI_PKG_POWER:0`?

2.2 perf_event improvements

The perf_event code is a mess of backwards compatibility code and a lot of baggage from the perfmon2 module it is based on. It could use a fairly extensive re-write.

The primary user of PAPI these days tends to be on Linux/perf_event. This drives development, and features are made that only take advantage of perf_event support.

Should we keep concentrating on this? Should PAPI become a thin self-monitoring and name-library layer on top of libpfm4 and perf_event? Or should we continue to be cross-platform?

2.2.1 Finer-grained user/kernel support

perf-events supports per-event user/kernel support.

Currently overwritten by the PAPI event-set wide settings.

Tricky to implement this right.

2.2.2 perf event rdpmc support

Linux 3.4 adds support for using userspace `rdpmc` for low-latency perfctr-style reads. We should support this. It allows latencies almost as low as those found with perfctr. See Figure 1.

I had some code implementing this which mostly worked, need to get back to merging it.

2.2.3 Enhanced Multiplexing Support

The current perf_event multiplexing support runs each event separately.

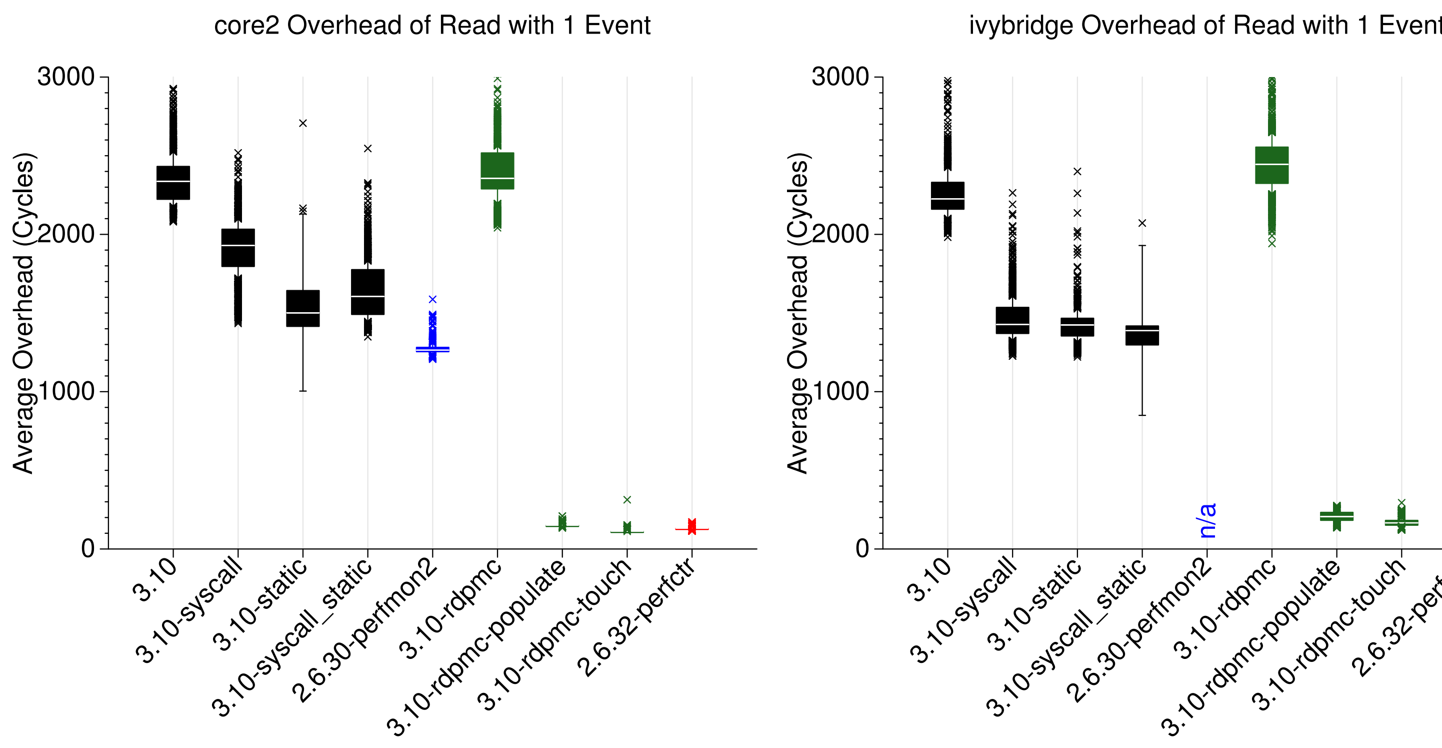

Figure 1: perf_event rdpmc overhead can approach perfctr. I have a paper in submission about this.

It might be good to add an interface that allows multiplexing events together when possible (for example, have FLOPS components together, or have IPC and CPI components together if possible).

This is made difficult by the various schedulability bugs in the Linux kernel.

2.2.4 Always enable inherit

Users seem to expect threads to accumulate for final totals, especially with OpenMP.

Some are even using the high-level PAPI interface.

Should we enable inherit by default if available?

2.2.5 Reported number of counters

Problem with the idea of "num_counters" when on perf_events that doesn't really matter, with software counters you can add more than physical without multiplexing.

Not always possible to add the number of events listed anyway due to conflicts and scheduling.

2.2.6 Split Each perf_event type to own component

Perf_event supports various different event types. With the exception of software events, the events cannot be added to the same shared event group. It might make sense to show these as separate

PAPI components?

We already do this for uncore. Maybe having power events different makes sense too?

The confusing part is the software events, because you can do things like have event groups containing an event like task-cycles as well as an arbitrary event like power.

2.2.7 cgroup support

perf_event supports cgroups. Expose this through PAPI?

Joining a cgroup is fairly straight forward with the interface.

Creating a cgroup is a lot of work, but in theory the user already can do this if they have cgroups working.

This might just involve parsing some files in /proc.

2.3 New Component Support

2.3.1 Parallel Library Components

MPI or OpenMP component. The new MPI 3.0 specification has a tools interface for reading out performance type information; we could have a component for this.

2.3.2 Watt's Up Component

I need to finish it.

2.4 New Platform Support

2.4.1 Windows Support

There has been some discussion on the mailing list about this?

2.5 Overhaul of the High Level Interface

The current interface is awful and falling apart. Need to architect a new one.

2.6 Interface Changes

2.6.1 Further cmp_info_t field removals

There are a few fields that describe the counters. Are they worth keeping? Many are not used internally by PAPI at all, but outside programs can access them directly via `PAPI_get_component_info()`.

The fields `fast_counter_read`, `fast_real_timer`, and `fast_virtual_timer` might provide useful information on the latency of various operations, and may guide someone who is instrumenting code in critical sections. The definition of "fast" can be a bit arbitrary though.

The `cntr_umasks` field specifies whether events can have umasks. These days PAPI internally just assumes all events can, so this value is meaningless.

2.6.2 get_memory_info vector

Another issue is `.get_memory_info` and how it applies to components. For example, a GPU might want to provide memory information for its onboard RAM. The best solution might be to leave a `get_memory_info` function hook for providing this kind of information and rename the OS one to `get_system_memory`.

2.6.3 Component Specific Settings

An often requested feature is some way to change component settings, similar to `ioctl()`. This would allow changing internal component options (such as sampling interval in the coretemp component). The current method of setting options, `PAPI_set_opt()`, only lets you set a certain subset of features, and is not extensible without modifying papi.h.

One suggestion is re-using the `user` function, and passing in name/value pairs in string format that the component could then parse. Since we are breaking API/ABI anyway, we could just add a new function totally. In that case we should also add a way of providing a list of which settings are possible to users.

A way of enumerating list? Bunch of strings? For example, sampling rate in coretemp?

There is a `user` function defined, which is currently unused.

This could be a good way to solve the "how do I set component-specific values" problem. Perhaps the `user` function could be used to pass in name/value string pairs that a component could act on.

If are breaking the API/ABI anyway though we might as well just add a new routine. Also a way to export the values supported.

2.6.4 PAPI_enum_components()

Should we add such a function?

```
int PAPI_enum_components(int current, int modifier);
```

With new support for distinguishing between components that are compiled-in versus enabled, we need some way of enumerating components.

2.6.5 More return values

It might be desirable to return more than just 64-bit values. Especially components that might want to return chunks of values at a time to be processed later. It is unclear the best way to handle this. One proposal is to return points to blobs of memory, though this could get messy quickly.

The `perf_event` component can return series of samples, as can some other components. How do we expose this?

Implementing the fits-in-64-bits option will be easy; designing a proper infrastructure for bigger values will be hard.

2.6.6 PAPI get OS info call

Open question: would a `PAPI_get_os_info()` call be useful for users, to get access to this new structure? The problem is if it is globally visible we get stuck supporting the various fields forever.

2.6.7 Multi-component EventSets

Allow creating eventsets with events from multiple components.

This would allow measuring interesting metrics like flops/Watt with one read.

This would complicate a lot of the PAPI code. An extra flag would be needed in the EventSet struct, and the components would have to be called iteratively.

Would this be part of software event support, or generic code?

2.7 Internal Changes

2.7.1 Official CDI interface

Should we have an official "papi_cdi.h" that is included by modules, rather than them including "papi_internal.h" and getting access to everything?

2.7.2 More Code Coverage

Code that is not compiled quickly breaks. The PAPI code is littered with `#ifdef`s and `configure` options, so much so that a lot of code quietly breaks without anyone noticing.

To fix this I've been attempting to remove as many ifdefs as possible. Have as much configuration be determined at run time as possible.

Use configure as little as possible as well, again try to determine things at runtime.

A big help will be to have configure enter component subdirs and call those subconfigures. Many people do not test some of the more obscure components because there's this barrier to entry, and it is hard to script.

2.8 Removing Obsolete Components

The ACPI component was removed when it was found to be not providing useful information.

At what point should perfctr and perfmon2 components be dropped? The Solaris OS support?

Should Windows support be dropped? It hasn't worked in ages and its `#ifdef`s make the code extremely messy.

Any-null support has been dropped.

2.9 Improving Test Infrastructure

The current test infrastructure reports many spurious errors when a counter such as PAPI_FP_OPS is unavailable. The tests should detect this early and do a "skip" rather than a fail.

We should also probably add a test early on that detects if counters are not available, and skip any tests that depend on HW counters being available. This will allow the tests to run on systems like VMs where PAPI is still usable for timing and components but not for HW counting.

We should also add infrastructure for things like perf_event specific tests. I have a number of these but they are currently outside the main PAPI tree.

2.10 Event Naming

2.10.1 PAPI-specific Umasks

It would be ideal if tools could specify features using only the name, without having to do multiple PAPI calls to set settings. For example, `PAPI_TOT_CYC:CPU=0:USER=1`.

There has been some work on this, but only for Linux/libpfm4 native events.

It would be nice if there were a generic solution.

This would involve a lot of string parsing in the front end.

2.10.2 Event Fields that take a Range

Some event times currently available in libpfm4 actually take a umask with a range, not just a plain umask value. For example, on Intel processors there's a CMASK field which can take an 8-bit parameter.

Currently there is no way to specify the availability of this so that it shows up sanely in `papi_native_avail`.

The best option for exposing this is probably adding a new enumeration type that enumerates these and returns a value like "CMASK=0-255". Currently enumeration only operates on strings; there is no way to pass back ranges. After passing back a value like shown the user will have to parse the string back into a range (and not just blindly try to use it as a umask).

This might also be useful for specifying events that fit a pattern, such as reporting "CORE0-15:TEMPERATURE" in a temperature component rather than enumerating 16 different ones explicitly.

Returning results like this will break programs that try to enumerate all possible events (although that's already a bit of a losing proposition). To not break things it might make more sense to provide yet another enumeration type where a program has to specifically ask for umasks that take a range, maybe `PAPI_NTV_ENUM_RANGES`.

It is unclear the best way to expose this via enumeration so that tools can automatically generate all possible events. PAPI enumeration tends to be done via strings, and all values returned are expected to be valid events. This makes it difficult to pass along values such as integer ranges.

2.10.3 Privileged Events

Some events can only be run by privileged users. It would be nice to expose this to users so they can know not to use them unless they have root privileges. This includes events such as ones that set the `ANY` bit on Intel Nehalem machines (on perf_event kernels) as well as Uncore events.

2.10.4 All Named Events Transition

A long-term PAPI goal is to migrate to the use of named events (strings for names) rather than externally visible eventcodes. This is driven by the lack of room in a 32-bit eventcode to encode all possible bit-fields in a modern CPU counter event, especially once components are thrown into the mix.

The libpfm4/perf_event component was a first implementation of this kind of infrastructure, and that seems to work. Here, events are created and assigned an eventcode at lookup time (once they are determined to be valid).

Currently events are allocated when they are first looked up (not at add time). This means that a program that enumerates all events could create thousands to millions of events, when only a few are used. This would waste memory and also slow lookups.

When doing a lookup by name, the simple approach would do a linear search of all events, one component at a time. This could be slow. It might make sense to have an additional layer of indirection that caches all event names and has a pointer to the actual events inside of each component.

2.10.5 Distinguish Events that can be Sampled

A way to distinguish between events that can/cannot be sample sources. Currently the only way to determine this is to check whether an event is 'derived' in the preset event_info structure.

2.11 Extended Sampling Interfaces

perf_event provides a complex sampling infrastructure, where multiple samples can be queued up and only read when a threshold is crossed. Currently we have no way of exposing this to the user.

Also, support for Intel PEBS and AMD IBS sampling eventually made it into the kernel. These provide extra values when sampling, anything from latency values to entire CPU state. We need to find a good way to export these values in a way that the user can access.

Various components, especially power ones, also return values in big buffers with multiple measurements at once. Exporting this through the traditional PAPI interfaces will be a challenge.

The current interface looks like this:

```
int   PAPI_overflow(int EventSet, int EventCode, int threshold,
                    int flags, PAPI_overflow_handler_t handler);


void (*PAPI_overflow_handler_t) (int EventSet, void *address,
                                 long long overflow_vector, void *context);
```

So you can set up one event to sample on, after a fixed amount of events, and you have to use a signal handler each time the overflow happens. You can get the processor state which will provide the instruction pointer. This is enough to provide rudimentary profiles.

Modern processors provide a lot more info and buffering, and on Linux the perf_event interface provides access to this.

Much of the related work cited (especially gooda) just uses the raw perf_event interface. There are some issues using this in PAPI.

- It is not cross platform. So if we designed an interface around the perf_event one, it would not necessarily be a good fit for non-Linux. So BlueGene, FreeBSD, Windows, etc.

  One thing we can do is just use the perf_event interface and use it as the official ABI and any other architecture would have to write a compatability layer. Easier now, but a lot of work down the road.

- The perf_event interface is obscure, varies by kernel version, not documented, and very complex.

  You can specify an arbitrary number of events to monitor, an arbitrary number of Linux specific events to monitor, gather an abritary number of events in a sample.

  This does include things like Intel and AMD's PEBS and IBS, including detailed processor state and cache latency info (assuming you have a new enough Linux kernel).

> The problem is the kernel just returns you this big binary blob of data. You either have to store this somewhere and deal with it later, or else try to decode it when you get it (with obvious performance issues).
>
> Writing a blob decoder is a huge deal. We'll have to do it I guess. Gooda does it, and actually I think they spent years getting it done right.
>
> The question is once we decoe the blob, how do we present it to the user? Predefined fields? Some sort of XML file?

A new advanced PAPI sample interface might look something like this:

```
PAPI_sample(int EventSet,
    int *EventCodes,
    int *perf_event_flags,
    int threshold,
    char *buffer,
    int buffer_size,
    PAPI_overflow_handler_t handler);
```

and what will happen is that once the buffer is 75% full (or so, user defined) you get a signal with the big binary blob of perf_event data, and then you save it or decode it.

Once we get the perf_event sample info, we might also want to include things like RAPL info, or MPI info, or Lustre info in the samples, things PAPI knows about but perf_event doesn't. This would be desirable but would involve having a PAPI sample abstraction layer above the OS one and all the inherent difficulties of interleaving the streams.

2.12 Extended User-events

Should the pre-defined event codes be absorbed by user-defined events? Can we create user-defined events using events found in the components?

2.13 Virtual/Cloud Issues

2.13.1 Enhanced Virtualized Timer Support

We are still investigating the best way to return useful time values while running inside of a VM.

2.14 Advanced Frequency Scaling Support

It is hard to determine exactly what frequency a CPU is running at.

PAPI is full of assumptions that MHz is fixed. The `PAPI_get_virt_cycles()` and `PAPI_get_real_cycles()` calls just multiply time by MHz for example.

Due to the design of the Linux frequency scaling mechanism, we can only poll for changes, we cannot just read them out. That makes adjusting for scaling or things like turbo boost very difficult.

We should still investigate enhancing PAPIs support though.

2.15 PAPI_FP_OPS

It is hard to choose proper events for floating point.

Currently Sandy Bridge `PAPI_FP_OPS` cannot hold all of x87, SSE, and AVX events at one time. If NMI watchdog is enabled and hyperthreading too, then only 3 events are available.

Should we implement multiplexed presets?

What about Haswell which might have *no* floating point events?

2.16 Avoiding High-Latency Component Initialization

Some components (most notably CUDA) can take a relatively long time to initialize. This can cause tools using PAPI to have a large startup overhead, even when the user does not plan to use the component at all.

We have attempted to address this with the disable component support described in Section **??**.

There have been some rumblings that this might still be an issue, most notably some components can take many seconds to initialize. If this is still the case we should re-investigate how to handle this.

2.17 Mitigating Long Latency Reads

As we move away from only CPU counter reads, there becomes the potential for long-latency reads. For example, reading values from a power meter connected to the system via a serial cable can take milliseconds. This can start to impact the performance of the program being measured.

One possible solution would be to have a component spawn a separate thread that handles I/O (almost like a daemon). This thread can periodically poll the hardware, and the PAPI component can return a cached (though possibly slightly old value) with low latency.

Setting this up properly can be a complicated process; it might be worthwhile to provide infrastructure that handles this.

2.18 Multiple users of itimers

How to avoid conflict in this case is an open question (especially as the RAPL component would like to do this).

2.19 New Pre-defined Events

It might be nice to have `PAPI_TOT_UOPS`.

2.20 Better Build Environment

It might be nice if configure automatically traversed down to the component directories. It's a bit difficult that configure has to be run explicitly for various of the components.

The dependencies in the Makefiles aren't complete, which means that sometimes a a `make clean` is needed after making a change. This mostly affects things in the test directories.

Make distclean sometimes leaves files hanging around (you can see this by making distclean and then running `git status`.

The `.gitignore` file could probably be improved.

2.21 Other Language Bindings

Currently we have support for C and Fortran. It might be nice to have Python, Java or Perl.

3 Conclusion

The PAPI 6.0 release will be an opportunity to improve PAPI support with input from the community.

4 References

References

[1] V.M. Weaver. New features in the PAPI 5.0 release. Technical report, University of Tennessee, 2012.