New Features in the PAPI 5.0 Release

Vince Weaver

vweaver1@eecs.utk.edu

24 August 2012

Abstract

The PAPI 5.0 release adds new functionality and cleans up old interfaces. We have attempted to preserve backwards compatibility as much as possible, but some code that uses PAPI may break in the transition from PAPI 4.x to PAPI 5.

This document describes the major changes introduced in PAPI 5.0 and gives information on how to properly migrate code to the new interfaces.

Contents

1 Background

The PAPI 5.0 release enhances the PAPI library while removing some of the limitations in the previous PAPI 4.x API/ABI. Many changes were also made in the PAPI-C [2] component interface (also known as the Component Development Interface, or CDI).

Many of the changes break the ABI, meaning that binaries linked against 4.x versions of PAPI will not link against the new version. That is why we bumped the major version number to 5.0.

We attempted to minimize the API breakage. This means that hopefully any code that compiled against PAPI 4.x will compile against 5.0 with no changes. See Section 1.1 for information on changes that might break the API.

We encountered many limitations with the PAPI-C CDI interface. We have majorly overhauled the interfaces used by components; unfortunately this means that any component maintained outside of the PAPI source code tree will likely need changes to work with PAPI 5.0. See Section 1.2 for a summary of these changes.

1.1 Changes that may break the API

We attempt to remain as backwards compatible as possible. Most code that worked with PAPI 4.x should work with PAPI 5.0 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.

`PAPI_COMPONENT_AND_MASK` and `PAPI_COMPONENT_MASK` have been removed as they are no-longer necessary due to the removal of the 16-component limit.

Various fields in the `.cmp_info` structure have been removed. Since we return this internal structure via the `PAPI_get_component_info()` call, any code that accessed obscure values in this structure may break.

`PAPI_perror()` has been changed to match the C-library style API. Previously `PAPI_perror()` took 3 arguments and wrote the results to a string, which then had to be printed. The new interface only takes one argument, a string, which is directly passed to stderr.

1.2 Changes that break components

We have made many changes to the CDI to address limitations in the PAPI 4.x interface. If you maintain a component you will probably need to make changes to your code. See Section 2 for a list of all changes that were made.

2 Changes to the Component Development Interface (CDI)

The PAPI-C Component Development Interface (CDI) provides an infrastructure for creating modularized "components" that add functionality to PAPI.

The CDI interface was originally just the hooks needed by the Linux/perfctr substrate. Therefore the fields and function pointers exported did not necessarily line up with the needs of other modules. Specifically, many operating-system specific interfaces were enabled which no module should really need.

In PAPI 5.0 we cleaned up the interface, removed many function pointers that no one was using (or that no one should have been using), and added a lot of extra functionality.

The CDI consists of two parts. The first is the `.cmp_info` structure that contains info on a component. The second is the function vector that contains the interfaces used by the component.

The following changes should be made to update a component from 4.x to 5.0.

Rename the function points .init to .init_thread .shutdown to .shutdown_thread .init_substrate to .init_component .shutdown_substrate to .shutdown_component

Modify your .name field to match your directory name.

Rename all PAPI_ESBSTR to PAPI_ECMP

Add a .description field.

Make sure the .allocate_registers function returns PAPI_OK for success (previously it returned 1 for success / 0 for failure).

Make sure to include papi_vector.h in your component, as papi_internal.h does not include it for you anymore.

Remove any use of the `PAPI_MPX_DEF_DEG` define. If you are setting the `.num_mpx_cntrs` field, and your component does not support multiplexing, then just set it to the same value as `.num_cntrs`

2.1 Changes to `.cmp_info`

The `.cmp_info` structure is directly exposed as ABI to the user via the `PAPI_get_component_info()`, so any changes made here will break the ABI and change the API (although the API changes can be made in a way that preserves backward compatibility).

2.1.1 Removing Operating System Specific fields from `.cmp_info`

In PAPI 4.x the 0th component (usually the hardware performance counter substrate: either perf_event, perfctr, or perfmon2 on Linux) had a special meaning. Its component fields held various OS specific information, such as kernel version numbers, POSIX timer values, etc. Other components could set these values in their own `.cmp_info` structure, but these were ignored.

With PAPI 5.0 these OS-specific fields are removed from the `.cmp_info` structure and placed in their own structure, `_papi_os_info` of type `PAPI_os_info_t`. This allows the OS specific values, of which there only need to be one copy, to stand alone.

In the OS split, the following was done:

- The following fields are removed from `.cmp_info` and put into a new `PAPI_os_info_t` structure: `itimer_sig`, `itimer_num`, `itimer_ns`, `itimer_res_ns`, `clock_ticks`, `os_version`.
- The following new fields are included in `PAPI_os_info_t`: `name`, `version`.
- A new `int _papi_init_os(void)` function is added for each supported Operating System that initializes the new `PAPI_os_info_t` structure.
- All uses of `_papi_hwd[0]->cmp_info` are turned into `_papi_os_info`

In addition, some padding was added to the end of `_papi_os_info` for future expansion.

Most of the moved fields involve the POSIX itimers. You only get three per process; currently values set in the component itimer fields were likely ignored as usually only the ones in component[0] were referenced (although the PAPI 4.x code was not consistent in this area). If components want to do software multiplexing or overflow they need to be able to set these values. How to avoid conflict in this case is an open question (especially as the RAPL component would like to do this).

The creation of one singular `_papi_os_info` allows building PAPI without a HW perfcounter component, something that is useful on systems without perfcounter support (such as pre-2.6.31 versions of Linux, or even Mac OSX or Windows). In 4.x a separate "any-null" substrate was required, but now that is no longer necessary.

ABI breakage: yes, the size of the cmp_info structure
CDI breakage: yes, components that access obscure cmp_info fields
API breakage: yes, anything that gets a cmp_info struct and acts on the more obscure fields
Implemented: git commit `bbd7871f4`

2.1.2 Component Names in `.cmp_info`

The start of 5.0 development began with a transition of the PAPI sourcecode from a CVS repository to git. Prior to the git transition the `name` and `version` fields in `.cmp_info` were automatically set by CVS, looking something like:

```
$Id: linux-net.c,v 1.11 2012/02/13 17:09:51 terpstra Exp $
```

With PAPI 5.0 the `name` field can be whatever the developer feels is appropriate. By default we have changed all in-tree modules to have the name set to the name of the primary source code file, with the extension dropped off. So for example, the name for "linux-rapl.c" would be "linux-rapl".

This allows existing code that used the following to still work:

```
if strstr(cmp_info.name,"example") ...
```

There can be some confusion because the name set with the `--with-components=` configure flag is based on the component directory and not related to the actual component name specified in `.cmp_info`.

ABI breakage: yes, the names of the components
CDI breakage: yes, the names of the components
API breakage: yes, the names of the components
Implemented: git commit `6f0c1230f29b5f`

2.1.3 Processor Specific fields in `.cmp_info`

In PAPI 4.4 the `.cmp_info` structure included many CPU-specific values, most of them relating to itanium processor support. These fields do not need to be global and should be limited to a specific CPU component. If necessary, the settings of these values can be detected at enumeration time.

In PAPI 5.0 we removed the following fields: Itanium: `data_address_range`, `instr_address_range`, `cntr_IEAR_events`, `cntr_DEAR_events`, `cntr_OPCM_events`, `opcode_match_width`, x86: `edge_detect`, `invert`, `profile_ear`, Power5: `cntr_groups`.

Unfortunately many programs (including some of the PAPI ctests) accessed these values directly from `.cmp_info`.

ABI breakage: yes
CDI breakage: yes, if components tried to set these fields
API breakage: yes; these values are exposed in cmp_info
Implemented: edge_detect, invert: git commit `401f37bc59`, the rest: git commits `5961c03d9`, `a245b50228`, `8f3aef4a9`, `047af62904`, `3f1f9e100`, `962c642abb`, `5aa7eac18393`, `1bf68d5d3dd`.

2.1.4 Other Additions to `.cmp_info`

We added a `short_name` field that components can set; this will eventually be prepended to event names (something like `short_name::retired_instructions`).

A `description` field has been added, for describing what the component does. This is printed (for example) by `papi_component_avail`.

A `disabled` and a `disabled_reason` field are added to allow disabling components, and for holding information on why a component is disabled.

A `component_type` field was added, so we can specify whether a component is CPU, GPU, I/O, etc. The `papi_xml_event_info` tries to provide this info. We have not defined the enum for this yet.

We add `reserved[]` padding so we can make ABI changes in the future.

ABI breakage: yes

CDI breakage: yes

API breakage: yes; these values are exposed in cmp_info

Implemented: short_name and description: `9f3e634a6b4`, disabled and disabled_reason: `0f2c25930ae94`

2.2 Changes to `papi_vector_t` Function Pointers

The CDI provides a number of function pointer hooks. These are how PAPI interacts with a component. In PAPI 4.x the available hooks included many which were not useful for most components, but were just a vestige of the hooks used by perfctr. For PAPI 5.0 we update the available hooks to better suit modern component development.

2.2.1 Removing Operating System hooks from `papi_vector_t`

As with `.cmp_info`, we create a separate `papi_os_vector_t` structure and move various operating system-specific functions there.

- The following pointers are removed from `papi_vector_t` and put into a new `papi_os_vector_t` structure: `get_real_cycles`, `get_real_usec`, `get_virt_cycles`, `get_virt_usec`, `update_shlib_info`, `get_system_info`, `get_memory_info`, and `get_dmem_info`.
- The following new fields added to `papi_os_vector_t`: `get_real_nsec`, `get_virt_nsec`
- A new `int _papi_init_os(void)` function initializes the new `PAPI_os_vector_t` structure, and each OS support file should have such a function.
- All components that reference the OS specific fields have had them removed (none should have been using them anyway).

PAPI 5.0 enhances support for `PAPI_get_real_nsec()` and `PAPI_get_virt_nsec()`. In PAPI 4.x these were just extrapolated from the cycle count based on MHz. This was suboptimal on platforms that can actually return nanosecond values. PAPI 5.0 adds support for `get_real_nsec`

and get_virt_nsec calls so that operating systems that support nanosecond resolution can properly provide it.

In the distant past, some components may have used CPU counters when calculating .get_real_cycles. This is not easily supported with the new layout.

ABI breakage: no

CDI breakage: yes

API breakage: no

Implemented: git commit 40bc4c57f8f9

2.2.2 Removing Bipartite Map Functions

In PAPI 4.x and before there were various function pointers that provided support for bipartite map event scheduling (bpt_map_avail, bpt_map_set, bpt_map_exclusive, bpt_map_shared, bpt_map_preempt, and bpt_map_update). This was developed for the complicated POWER event group dependencies, but was made generic so other code could use it. Only perfctr used this support, as libpfm3 and/or perf_event kernels handle the scheduling for you.

In PAPI 5.0 we remove the functions from the vector table; the code has been moved to papi_bipartite.h and can be included by any component that specifically needs this support.

ABI breakage: no

CDI breakage: yes

API breakage: no

Implemented: git commit e69815d7429b2

2.2.3 Native Event Conversion Routines

PAPI 4.x had the following: ntv_name_to_code, ntv_code_to_name, ntv_code_to_descr, ntv_code_to_bits, and ntv_bits_to_info.

ntv_code_to_bits was currently only implemented by perfctr. This causes problems, as PAPI_get_event_info() was implemented via .ntv_code_to_bits followed by .ntv_bits_to_info. Because of this, the event_info_t structure is only available when using perfctr.

In PAPI 5.0 we add a new ntv_code_to_info function and remove ntv_bits_to_info. This allows direct access to the event info, which is necessary to access the new enhanced event information described in Section 4.2.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `9c54840e5`

2.3 Substrate to Component Rename

PAPI 4.x used the name "substrate" and "component" interchangably (although usually substrate referred to the main CPU component).

In PAPI 5.0 we rename all uses to use "component".

This involved fixing all components, renaming `.init_substrate` to `.init_component` and `.shutdown_substrate` to `.shutdown_component`.

Also, to avoid confusion (this was a common source of bugs) we renamed `.init` to `.init_thread` and `.shutdown` to `.shutdown_thread`.

In addition, `PAPI_ECMP` was added to mirror `PAPI_ESBSTR` (the latter was left for compatability reasons) and various other uses of SUBSTRATE were renamed COMPONENT.

ABI breakage: no

CDI breakage: yes

API breakage: no

Implemented: git commit: `79b01a47ce12f4f`

2.3.1 Removing Obsolete Functions

The `.add_prog_event` function was only used by `PAPI_add_pevent()` PAPI-3 compatibility code and not actually implemented by any of the substrates. It was removed in PAPI 5.0.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `8da3622291a`

3 New/Modified PAPI interface functions

PAPI 5.0 has added some new user-visible interface functions.

3.1 PAPI_add_named_event()

```
int PAPI_add_named_event(int EventSet, char *EventName);
```

In PAPI 4.x adding an event by name was a two stage process: first you ran `PAPI_event_name_to_code()` (checking for errors) and then passed the result to `PAPI_add_event()`. PAPI 5.0 adds `PAPI_add_named_event()` that reduces this to one step.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `1c87d89c4a7`

3.2 PAPI_remove_named_event(), PAPI_query_named_event()

```
int PAPI_remove_named_event(int EventSet, char *EventName);
int PAPI_query_named_event(char *EventName);
```

These are similar in idea to `PAPI_add_named_event()`.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `1c87d89c4a7`

3.3 PAPI_get_event_component()

```
int PAPI_get_event_component(int EventCode);
```

This is meant to replace the `PAPI_COMPONENT_INDEX()` macro now that we have removed the 16-component limit and the component can no longer be determined by shifting/masking the event number.

A `PAPI_COMPONENT_INDEX()` compatibility macro is provided.

This was added as part of the 16-component limit removal described in Section 6.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `d1ed12b79807e`

3.4 PAPI_enum_cmp_event()

```
int PAPI_enum_cmp_event(int *EventCode, int cid, int modifier);
```

The existing `PAPI_enum_event()` does not take a component ID; it gets the component number from the event passed in. This new `PAPI_enum_cmp_event()` makes the component ID explicit.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `deac54cc8184`

3.5 PAPI_get_component_index()

```
int PAPI_get_component_index(char *name);
```

`PAPI_get_component_index()` returns the component index of the specified (by name) component. This is useful for finding out if a specific component exists.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `420c3d11c2822`

3.6 PAPI_disable_component()

```
int PAPI_disable_component_by_by_name( char *name );
int PAPI_disable_component( int cidx );
```

These function lets you disable components before running `PAPI_library_init()`. There are various reasons you may want to do this, usually to avoid overhead of components you are not interested in.

Here is some source code that shows how to disable the "example" component:

```
int cidx, result;
```

```
cidx = PAPI_get_component_index("example");

if (cidx>=0) {
   result = PAPI_disable_component(cidx);
   if (result==PAPI_OK)
      printf("The␣example␣component␣is␣disabled\n");
}
/* ... */
PAPI_library_init();
```

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commits `420c3d11c2822`, `ac2eac56dcd7ec`

4 Extensions to PAPI_event_info_t

In PAPI 5.0 we have added additional information to events to allow tools to make better use of the data returned.

4.1 PAPI_event_info_t Layout

The `PAPI_event_info_t` structure was re-arranged to better separate out the predefined event specific fields from the other fields. At one point we considered slimming down some of these fields (as some of them can possibly be many kilobytes in size). Upon reviewing usage of the code it turns out most users only ever have one `PAPI_event_info_t` around at a time; they call `PAPI_get_event_info()` and re-use the structure. Thus optimizing some of the fields to be dynamically allocated would not help overhead much and would add additional memory allocation/freeing complications (as well as breaking backward compatibility).

In the end the behind-the-scene preset structures *were* changed to minimize memory consumption but this was not a user-visible change.

4.2 Enhanced Event Info

In Kluge et al.'s "Collecting Distributed Performance Data with Dataheap" [1] paper they describe extended event information that can improve the usefulness of results provided by PAPI. Many of these are useful when trying to correlate system-wide events, and also when correlating events collected from different components with non-synchronized timestamps.

In PAPI 5.0 we extend the `PAPI_event_info_t` structure to provide this additional event information. The user can access this info via `PAPI_get_event_info()`, which more components should support due to the addition of the `ntv_code_to_info` function hook described in Section 2.2.3.

Ideally all of these fields will be set to proper values, but most components still leave these set to the defaults.

ABI breakage: yes, we change external visible `PAPI_event_info_t`
CDI breakage: components will have to handle the new fields
API breakage: yes, we change external visible `PAPI_event_info_t`
Implemented: git commit `c4579559d7`

4.2.1 Specifying Extended Value Types

In PAPI 4.x and before, results returned were always unsigned 64-bit integers.

PAPI 5.0 adds a new `data_type` field which allows for more interesting values, although they still all must fit in 64-bits.

Currently the following types can be specified:

```
PAPI_DATATYPE_UINT64 = 0,          /**< Data is a unsigned 64-bit int */
PAPI_DATATYPE_INT64,               /**< Data is a signed 64-bit int   */
PAPI_DATATYPE_FP64,                /**< Data is 64-bit floating point  */
```

Other data types can be imagined, such as two 32-bit values, or a ratio.

4.2.2 Units

PAPI 4.x assumed all results were a unitless count. This made reporting values difficult for some components which had results with units (Watts, Joules, Kelvin, etc.).

PAPI 5.0 adds the `units` field to `PAPI_event_info_t`. It is a string to provide maximum flexibility when specifying units (and to avoid having to have a constantly updated list of list of unit `#defines` in papi.h).

4.2.3 Component Index

The `component_index` field was added that allows mapping event info back to the parent component.

4.2.4 Location

It is useful to know where on the system that an event is running. The `location` field indicates this.

The current available values are:

```
PAPI_LOCATION_CORE = 0,             /**< Measures local to core      */
PAPI_LOCATION_CPU,                  /**< Measures local to CPU (HT?) */
PAPI_LOCATION_PACKAGE,              /**< Measures local to package   */
PAPI_LOCATION_UNCORE,               /**< Measures uncore             */
```

4.2.5 Value Type

Traditionally PAPI measures value as a running sum, starting from a `PAPI_start()` call. This does not make sense for certain types of measurements, such as temperature where you really only want an instantaneous value.

The `value_type` field lets you specify different types of values. Currently supported are:

```
PAPI_VALUETYPE_RUNNING_SUM = 0,     /**< Data is running sum from start */
PAPI_VALUETYPE_ABSOLUTE,            /**< Data is from last read */
```

4.2.6 Timescope

The `timescope` fields lets you know the scope of the current measurement; whether it starts from the beginning or if it is only since the last read.

```
PAPI_TIMESCOPE_SINCE_START = 0,     /**< Data is cumulative from start */
PAPI_TIMESCOPE_SINCE_LAST,          /**< Data is from last read */
PAPI_TIMESCOPE_UNTIL_NEXT,          /**< Data is until next read */
PAPI_TIMESCOPE_POINT,               /**< Data is an instantaneous value */
```

4.2.7 Update Type and Frequency

Not all events are continuously updated. The `update_type` and `update_freq` fields let your code know how and how often the counts are updated.

```
PAPI_UPDATETYPE_ARBITRARY = 0,      /**< Data is cumulative from start */
PAPI_UPDATETYPE_PUSH,               /**< Data is pushed */
PAPI_UPDATETYPE_PULL,               /**< Data is pulled */
PAPI_UPDATETYPE_FIXEDFREQ,          /**< Data is read periodically */
```

5 Event Enumeration Issues

One of the more difficult tasks that PAPI does is enumerate a list of all possible events, such as is done with the `papi_native_avail` utility.

Currently, you can enumerate in various ways. PAPI itself handles enumerating the predefined events (including many variations of sub-typing).

Enumeration of native events is handled by each component. Each must support `PAPI_ENUM_FIRST`, `PAPI_ENUM_EVENTS` and `PAPI_NTV_ENUM_UMASKS`. This allows finding the first event, finding each subsequent event, and finding all umasks (options) for each event. (Some of the HW counter substrates support more advanced enumeration, such as for example `PAPI_NTV_ENUM_GROUPS` on POWER).

With the move to the `libpfm4` library during the PAPI 4.x series, additional umasks have been available that take either boolean or integer parameters. PAPI supported using these umasks, but did not show them when enumerating.

With PAPI 5.0 these values are now shown. Currently only the first value is enumerated. For example, for the boolean umask "u" (meaning, use user-only events) it is enumerated as `u=0`. The other sate, `u=1` is not shown. The umask "c" (meaning CMASK) is of integer type and can take values 0 through 255. Currently only `c=0` is enumerated.

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. For a longer discussion of this issue, see Section 13.8.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit `36e864b37467b`

6 Removing the 16-component limit

In PAPI 4.0 the "native events" were specified at the bit level (see Figure 1). The event, umask, and component could be determined by shifting and masking the event.

With the advent of libpfm4 and perf_events, as well as modern hardware, this bit layout of native events became unworkable. There were more distinct events and umasks than could fit in a 32-bit value.

With PAPI 4.1 and libpfm4 the native events were changed to be dynamically allocated, so that the bit limitations could be avoided. The 16-component limit was left in.

In PAPI 5.0 we remove the 16-component limit.

As Figure 2 shows, we did this by adding a new table, which is indexed by the native event (with the NATIVE_MASK removed). This table holds the component number for the event, and the internal event. These two values are then passed on to the component as necessary. In effect,

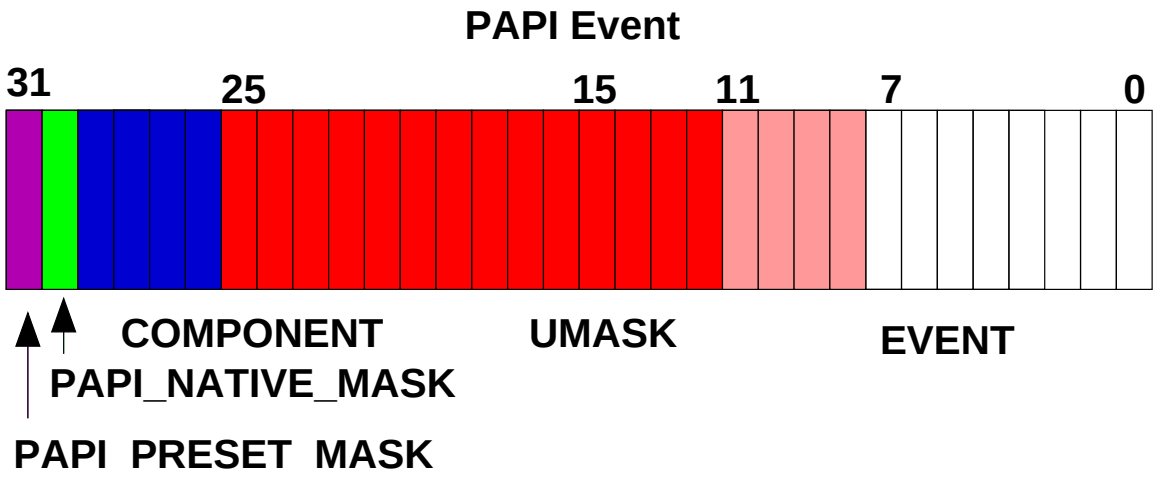

Figure 1: PAPI 4.0 native event bit layout. We ran out of room for expansion, and were limited to 16 components.

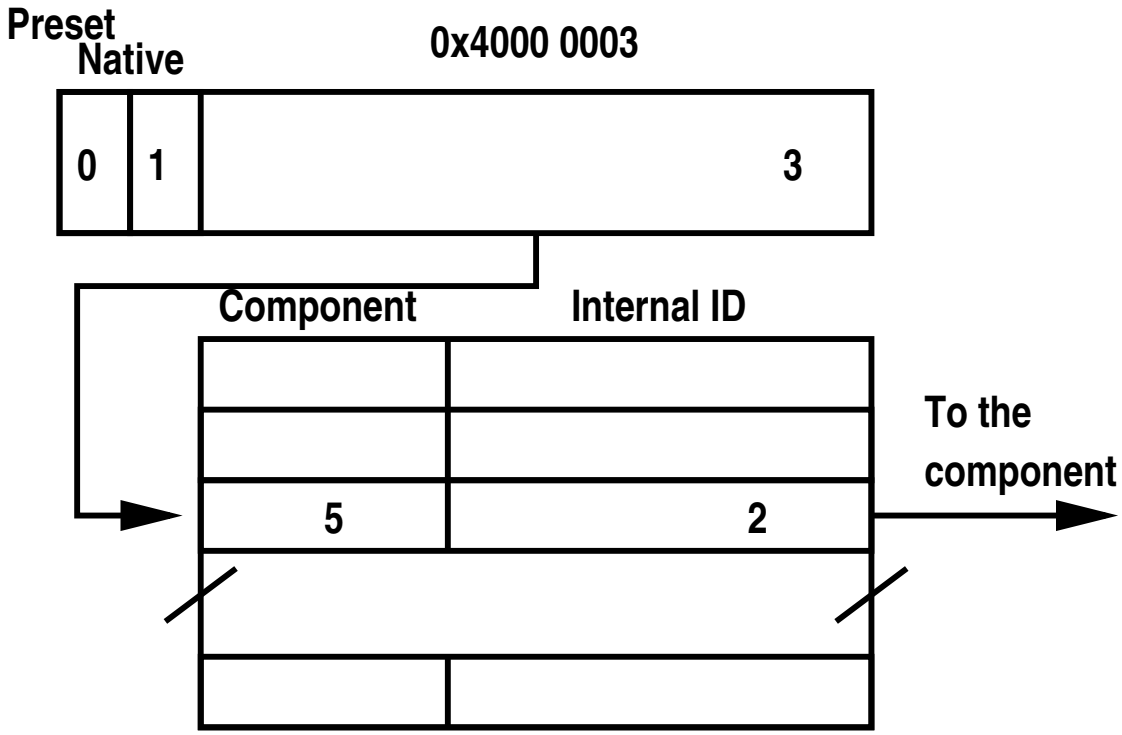

Figure 2: PAPI 5.0 native event lookup. Component lookup is now done by an array lookup rather than a shift-and-mask.

a shift/mask operation has been replaced with a table lookup. Performance measurements show that the change in overhead is minimal.

The `PAPI_COMPONENT_MASK` and `PAPI_COMPONENT_AND_MASK` values were removed. Current code or components that used them can just remove them.

In theory components can now have a full 32-bit internal event value for their own use.

For backwards compatibility the `PAPI_COMPONENT_INDEX()` call has been left around and re-implemented using the new `PAPI_get_event_component()` function.

ABI breakage: no

CDI breakage: maybe, if components are making assumptions based on event bits

API breakage: possibly if the code made assumptions about event layout or used `PAPI_COMPONENT_MASK` or `PAPI_COMPONENT_AND_MASK`

Implemented: git commit `9a26b43d29f9a`

7 Error Propagation

PAPI is a library, and thus it has limited means for passing error reports back up to the user. In PAPI 5.0 we tried to enhance this.

It is now possible to pass component initialization failure messages. The component just has to set the `.disabled_reason` field, which the user can access via `cmp_info_t`.

ABI breakage: yes

CDI breakage: yes

API breakage: yes

Implemented: git commit `0f2c25930ae94`

8 Virtualized Events (PAPI-V)

PAPI 5.0 has many enhancements for running in virtual systems.

We detect if we are running inside of a VM by accessing leaf 0x40000000 via cpuid. We return whether we are inside a VM and the VM name if available in the new `virtualized` and `virtual_vendor_string` fields in `hw_info_t`. This has been tested to work in both KVM and VMware environments.

We have a new "vmware" component that provides information from VMware pseudo-performance counters as well as the vmGuestLib library.

We have a new "stealtime" component that provides information on how long the VM we are running in has been scheduled out. This currently only works on KVM.

Recent versions of VM software provide virtualized performance counters. PAPI runs natively inside of these without changes. This includes the newest versions of VMware workstation (requires a special configuration option), the next version of VMware ESX, and KVM running on 3.3 or newer kernels with a new enough KVM/qemu.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: reporting we are in VM: `d7496311119`

9 Better CPU Frequency Support

Currently our tests can fail if frequency scaling (either for power-save or for turbo-boost) happens.

PAPI 4.x read the CPU frequency from `/proc/cpuinfo` at startup and use this value always. This value is guaranteed to be wrong on systems doing DVFS because the system is likely idle when PAPI starts but soon after the frequency is ramped up as the load increases.

Linux supports reading the current cpu frequency (and all possible frequencies) from files under /sys. The key missing feature is that Linux *will not* notify you in any way if the frequency changes, so there's no way for PAPI to know short of periodically checking.

PAPI 5.0 adds a `minimum_mhz` and `maximum_mhz` value to the `hw_info_t` structure. These are set based on the files in `/sys/devices/system/cpu/cpu0/cpufreq/` This takes into account DVFS, although we still do not handle turbo-boost as well as stranger things such as heterogeneous processors such as the ARM BIG.little.

Previous MHz values returned by PAPI are set to the `maximum_mhz` value.

The `/proc/cpuinfo` value of MHz is returned in cases where we cannot determine MHz values from /sys.

ABI breakage: yes

CDI breakage: no

API breakage: yes

Implemented: git commit `d9a58148447b`

10 Component Detection

Until PAPI 4.2.1 if a component returned an error in its `init_substrate()` routine, then PAPI as a whole would fail. To get around this, components worked around this by indicating success but setting `num_native_events` to be 0. As of PAPI 4.2.1 there is a workaround that will set this workaround by default if a PAPI error is returned.

PAPI 5.0 implements a better solution: if component initialization fails then it is marked as `disabled` with the `disabled_reason` field set in cmp_info.

ABI breakage: yes

CDI breakage: yes

API breakage: yes

Implemented: yes 6b18415879c16f

11 Locking Changes

PAPI 5.0 includes a cleanup of the locking code.

PAPI still includes hand-tuned assembly mutexes. PAPI 5.0 provides the new configure option of `- -with-pthread-mutexes` which enable using POSIX pthread mutexes. This provides support for platforms not having hand-coded locks, as well as allowing the various Valgrind race-condition detection code to work with PAPI.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit: `92689f626b`

12 Other Additions/Removals/Changes

In addition to API/ABI changes, there has been a lot of general development of the PAPI infrastructure.

12.1 New Components

The following existed in PAPI 4.2.1 but have seen major overhauls:

- coretemp – temperatures, voltages, fan readings on Linux
- cuda – NVIDIA GPU events
- example – simple test component
- infiniband – gives infiniband stats
- lmsensors – gives temperature/fan readings
- lustre – info from the lustre filesystem
- mx – myrinet stats
- net – generic Linux network statistics

The following are new for PAPI 5.0:

- appio – I/O statistics
- bgpm – various BG/Q modules
- nvml – power readings for NVIDIA cards
- rapl – power/energy estimates for Intel SandyBridge chips
- stealtime – stealtime from inside of KVM
- vmware – VM stats as well as pseudo-performance counter values from inside of VMware

12.2 Removed Components

The ACPI component was removed, as it was broken and has been replaced in functionality by the coretemp component.

12.3 Removed Windows Support

Microsoft Windows support has not really been possible since the PAPI 3.7 release. In PAPI 5.0 we removed the code. (git commit `3148cba5ce16b4`)

12.4 Removed Java Support

PAPI 3.x (or before) had some initial support for access from Java. This was unmaintained and removed in PAPI 5.0 (git commit `666249a803b10`).

12.5 New Operating Systems

No new ones have been added, although the FreeBSD support was majorly overhauled.

OSX support has been temporarily removed but can be added back.

12.6 New Processor Support

BlueGene/Q support was added in 4.4 but is still relatively new.

Intel SandyBridge has been a work in progress.

Intel Ivy Bridge support is new, as is Cedar View Atoms.

AMD Interlagos support is still new.

12.7 New Pre-defined Events

Dan added `PAPI_REF_CYC` which can give you unscaled cycles on newer Intel processors.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit: `8b9b6bef94030ee`

12.8 Component Name Prepending

We now pre-pend the component to the front of event names. This lets you specifically ask for a component. The separator is three colons. This lets you have event names such as `PE:::INSTRUCTIONS_RETIRED:ALL` and `CUDA:::TEXTURE_MISSES`. For backward compatibility all components will be searched if an event does not provide a pre-pended component name.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: git commit: `57aeb9d478cb04e6d`

12.9 Improved Error Handling

- PAPI's error functions (`PAPI_perror()` and `PAPI_strerror()`) should be closer to their POSIX equivalents (git commit `a366adf7da579eb652`).
- It would be nice to have some method of passing more detailed error information (including possibly strings) up through the PAPI stack from the components

13 Items not making it into PAPI 5

13.1 Uncore Support

`perf_event` uncore support will finally arrive in the 3.5 kernel release. We have not had time implement PAPI support.

13.2 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.

13.3 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?

13.4 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`.

13.5 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.

13.6 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.

13.7 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.

13.8 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.

13.9 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.

13.10 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 will eventually make 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.

13.11 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?

13.12 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.

13.13 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.

13.14 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.

13.15 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.

13.16 Enhanced Virtualized Timer Support

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

13.17 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.

13.18 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.

13.19 perf event rdpmc support

Linux 3.4 adds support for using userspace `rdpmc` for low-latency perfctr-style reads. We should support this.

13.20 Watt's Up Component

I need to finish it.

13.21 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 Ivy Bridge which might have *no* floating point events?

13.22 Enhanced Multiplexing Support

The current perf_event multiplexing support runs each event separately.

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.

13.23 Component Presets

It's currently not possible to set up presets in components. This should be fixable, but what kinds would we want? Would something like `PAPI_CPU_POWER` make sense?

13.24 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 3.6.

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.

13.25 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.

13.26 Multiple perfcounter implementations

With this OS separation it would become possible to build multiple perfcounter implementations into PAPI and choose the proper one at runtime (enabling a dual perfctr/perf_event PAPI) although the usefulness of that has probably dwindled now that perf_events is widely available.

13.27 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.

13.28 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).

13.29 Fixes for RAPL component

It doesn't handle overflow well.

13.30 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?

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.

13.31 New Pre-defined Events

It might be nice to have `PAPI_TOT_UOPS`.

13.32 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.

13.33 Other Language Bindings

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

13.34 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.

14 Conclusion

The PAPI 5.0 release showcases many months of work by the PAPI team. We hope you find it to be a satisfactory successor to the venerable PAPI 4.x releases.

15 References

References

[1] M. Kluge, D. Hackenberg, and W.E. Nagel. Collecting distributed performance data with dataheap: Generating and exploiting a holistic system view. In *Proc. of the 2012 International Conference on Computational Science*, pages 1969–1978, June 2012.

[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. In *3rd Parallel Tools Workshop*, pages 157–173, 2009.