Proposed Post-4.2 PAPI API/ABI Changes

Vince Weaver

vweaver1@eecs.utk.edu

13 March 2012

Abstract

While expanding the number of components available on PAPI we are running into some limitations that were missed during the 3.x to 4.0 PAPI-C transition. In order to move forward, we will need to break the PAPI ABI, API, and component interface. Most of the changes will hopefully be small and not noticed by the average PAPI user. We are preparing this document in advance in the hope that we can get feedback from those affected by these changes.

Contents

1 Background

We are currently extending PAPI in new directions and we are encountering various limitations of the current PAPI 4.2 API/ABI as well as the PAPI-C component interface (known as the Component Development Interface, or CDI).

What follows are discussions of new functionality that will require extending the ABI in non-backwards compatible ways. Since such extensions will require a major version number bump, we are trying to have as many changes as possible happen at the same time to avoid needing a similar change at a later time.

ABI changes, while annoying for backward compatibility, usually just require programs to be recompiled. A few of the changes we are making may also require API changes, which would involve users having to make changes to their code. We will do our best to avoid API changes as it is inconvenient for everyone involved.

We are preparing this document in advance so that the various PAPI users can provide feedback, both to let us know if we have missed anything, as well as to protest changes that might require too much code-churn.

2 Changes to the Component Development Interface (CDI)

The PAPI-C component interface (also known as the Component Development Interface or CDI) was originally based on the hooks needed by the perfctr substrate. Because of this, the CDI contains many fields and function pointers not needed by most modules. Some of these extra fields may cause confusion, some really are Operating System (not module) specific, and pretty much none of it is documented well.

2.1 Changes to `.cmp_info`

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 `.cmp_info` structure is directly exposed as ABI to the user via the `PAPI_get_component_info()`, so any changes made here will break both the API and the ABI (although users are only affected if they access some of the more obscure fields directly, something that is hopefully unlikely).

2.1.1 Splitting Operating System Specific fields out of `.cmp_info`

Currently the 0th component (usually the hardware performance counter substrate: either perf_event, perfctr, or perfmon2 on Linux) has a special meaning. Its component fields hold various OS specific information, such as kernel version numbers, POSIX timer values, etc. Other components can set these values in their own `.cmp_info` structure, but these are ignored.

We propose splitting these fields out into their own structure. This allows the OS specific values, of which there only need to be one copy, to stand alone. It also 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). Previously this could be accomplished with the "any-null" substrate, but for various reasons this was a sub-optimal solution. 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.

Most of the fields being moved 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] are referenced (although the PAPI code is 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.

To implement the OS split, the following needs to be 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`

We should pad the structure with some extra members so that we can further extend things without breaking the ABI.

A `get_os_info()` type helper function might be nice to have, so that components can access this info without having to declare the main structure as globally visible.

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: yes: bbd7871f4

2.1.2 Other Changes to the `.cmp_info` structure

Currently the `name` and `version` fields are automatically set by CVS. With the change to git we'll have to change how these are set. Currently code caring about the name field tends to do code of the sort:

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

Because of this, we should standardize on the `name` field being the name of the component source file.

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: 6f0c1230f29b5f

2.1.3 Processor Specific fields in `.cmp_info`

Currently we carry along some processor-specific flags. Should these be removed? This information is only really relevant to the specific perfcounter components, and should be revealed as part of enumeration.

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

The fields of interest are Itanium: `data_address_range`, `instr_address_range`, `cntr_IEAR_events`, `cntr_DEAR_events`, `cntr_OPCM_events`, `opcode_match_width`, `profile_ear`, Intel: `edge_detect`, `invert`, Power5: `cntr_groups`.

ABI breakage: yes

CDI breakage: no, as components shouldn't care about these values

API breakage: yes; these values are exposed in cmp_info

Implemented: partial; edge_detect, invert: 401f37bc59

2.1.4 Counter Attributes

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.

ABI breakage: yes

CDI breakage: maybe

API breakage: yes; these values are exposed in cmp_info

Implemented: N/A

2.1.5 Proposed Additions to the `.cmp_info` structure

We should add a `short_name` field that components can set, and this can be prepended to event names. Also a `description` field to describe what a component does.

Should we add a `type` field that notes whether a component is a CPU, I/O, etc? The `papi_xml_event_info` tries to provide this info. Should it be a string, an enum, or something else?

We should also add 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: partial; short_name and description: 9f3e634a6b4

2.2 Changes to `papi_vector_t` Function Pointers

The CDI provides a large number of function pointer hooks, many of which are a legacy of its perfctr substrate heritage and not needed by most components.

2.2.1 Splitting Operating System Specific fields out of `papi_vector_t`

As with `.cmp_info`, we propose creating a separate `papi_os_vector_t` structure and moving OS-specific functions there.

To implement the OS split, the following needs to be done (we have implemented these changes already and they seem to work):

- 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`.
- The new `int _papi_init_os(void)` function initializes the new `PAPI_os_vector_t` structure.

- All components that reference the OS specific fields should have them removed (none should be using them anyway).

In the past, some substrates may have used CPU counters when calculating the result of `.get_real_cycles`. We would not be able to support that with these new changes.

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`.

ABI breakage: no

CDI breakage: yes

API breakage: no

Implemented: yes: 40bc4c57f8f9

2.2.2 Removing Bipartite Map Functions

There are various function pointers that provide 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. Currently only perfctr uses it, as libpfm3 and perf_event kernels in theory handle the scheduling for you.

Should we make this code non-generic and just fold it in with the perfctr code?

ABI breakage: no

CDI breakage: yes

API breakage: no

Implemented: yes: e69815d7429b2

2.2.3 Repurposing the User Function

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.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: N/A

2.2.4 Native Event Conversion Routines

Currently we have 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` is currently only implemented by perfctr. This causes problems, as `PAPI_get_event_info` does `.ntv_code_to_bits` followed by `.ntv_bits_to_info`. Because of this, the `event_info_t` structure is only available when using perfctr.

Ideally we should add a new `ntv_code_to_info` function. We discuss this a bit further in Section 2.2.6. Once we have that, `ntv_bits_to_info` will be removed.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: yes: 9c54840e5

2.2.5 Removing Obsolete Functions

The `.add_prog_event` function is only used by `PAPI_add_pevent()` PAPI-3 compatibility code and not actually implemented by any of the substrates. It should probably be removed.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: yes: 8da3622291a

2.2.6 Proposed Additions to `papi_vector_t`

`ntv_code_to_info()`

As described earlier, it currently is not possible to get access to the `event_info_t` information if `ntv_code_to_bits` is not available (and it's not for many components). By adding a `ntv_code_to_info()`, we can allow direct access to the event info, which is necessary to allow access to enhanced event information.

`get_real_nsec`, `get_virt_nsec`.

Currently PAPI has support for PAPI_get_real_nsec() and PAPI_get_virt_nsec() which are just extrapolated from the cycle count based on MHz. This is suboptimal on platforms that can actually return nanosecond values. We should make an OS-vector for this and let OSes that support proper nanosecond reporting handle things themselves.

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.

ABI breakage: no
CDI breakage: no
API breakage: no
Implemented: N/A

3 New/Modified PAPI interface functions

The PAPI interface has many functions; here we propose a few more.

3.1 PAPI_add_named_event()

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

Currently adding an event by name is a two stage process: first you run `PAPI_event_name_to_code()` (checking for errors) and then pass the result to `PAPI_add_event()`. With the move to all named events this gets a bit cumbersome; this is a very common operation. We can easily implement a function that does both for us.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: yes; 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: yes; 1c87d89c4a7

3.3 PAPI_which_component()

```
int PAPI_which_component(int event);
```

This is helpful when using multiple components, especially when we drop 16-event limit.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: no

3.4 PAPI_enum_component_event()

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

`PAPI_enum_event()` does not take a component ID; it gets the component number from the bitfield. This needs to be changed when we move to break the 16-component limit.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: no

3.5 PAPI_enum_components()

```
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.

ABI breakage: no
CDI breakage: no
API breakage: no
Implemented: no

4 Extensions to PAPI_event_info_t

In Kluge et al.'s "Collecting Distributed Performance Data with Dataheap" paper they describe other useful event description fields that we might want to provide to users. PAPI's diverse set of components can now report values that are more than just unitless 64-bit unsigned values. We need to provide units, datatypes, and other such information to the user.

The best way to provide this information is to extend the `PAPI_event_info_t` structure and export this information to the PAPI user. This is in theory possible with `PAPI_get_event_info()`, but currently this breaks on most substrates/components as they do not implement the `ntv_code_to_bits` function, as described in Section 2.2.6.

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: yes; c4579559d7

4.1 Reducing the Size of `PAPI_event_info_t`

The current `PAPI_event_info_t` looks like this:

```
typedef struct event_info {
  unsigned int event_code;
  unsigned int event_type;  /* preset only */
  unsigned int count;       /* number of terms */
  char symbol[PAPI_HUGE_STR_LEN]; /* name of the event */
  char short_descr[PAPI_MIN_STR_LEN];    /* not used */
  char long_descr[PAPI_HUGE_STR_LEN];
  char derived[PAPI_MIN_STR_LEN];        /* presets */
  char postfix[PAPI_MIN_STR_LEN];        /* postfix presets */
  unsigned int code[PAPI_MAX_INFO_TERMS];
  char name[PAPI_MAX_INFO_TERMS][PAPI_2MAX_STR_LEN];
  char note[PAPI_HUGE_STR_LEN];          /* developer note */
} PAPI_event_info_t;
```

Many of these fields are predefined-event only, and take up a lot of space with all those static character arrays.

It might make sense to change the static arrays to dynamically allocated ones to reduce footprint, though this raises the issue of having to free these strings later.

In the end, making the visible structure smaller broke too many things. Leaving it as is but changing the behind-the-scenes structs made more sense. Users rarely allocate more than one `PAPI_event_info_t` anyway.

4.2 Specifying non-64bit uint Values

Currently results returned by PAPI are always unsigned 64-bit integers. There are other values that can be cast to fit in the same return value: signed 64-bit, 64-bit floating point values, two 32-bit values (a ratio?). A field can be added to `PAPI_event_info_t` that says what the value contains, and the possibilities are probably small enough to be assigned #defines.

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.

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

4.3 Specifying Units

Currently we assume the result is a unitless "count". Some of our components already need to specify units (Watts, Joules, Kelvin, etc.).

A field can be added to `PAPI_event_info_t`. Probably the best idea is having this be a string, that way it is easily extensible without having to have a huge list of unit #defines in papi.h.

4.4 Specifying Component

We should have a field in `PAPI_event_info_t` that quickly allows following back to the parent component.

4.5 Kluge et al.'s other Extensions

In Kluge et al.'s "Collecting Distributed Performance Data with Dataheap" paper they describe other useful fields that we might want to add to the `PAPI_event_info_t` structure. 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.

- Location (Local, Uncore, CPU)
- Unit and Data Type (Mentioned previously)
- Value Type (monotonic, sum, instantaneous)
- Time Scope (from start or since last)
- Read Mode and Frequency

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 ways of sub-sorting).

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.

Recently we have encountered event types that cannot be enumerated with the current PAPI setup.

ABI breakage: no
CDI breakage: no
API breakage: no
Implemented: N/A

5.1 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`.

5.2 Software Events

On perf_events there is a distinction between hardware events and software events provided by the kernel. Even if HW counting is not working (due to lack of support, etc.) the SW events are still supported. It would be good to be able to enumerate these and use them even if the HW counters are not available. This is also useful in virtualized systems where access to the HW counters might not be available, but a perf_event kernel is still installed and can provide SW events.

5.3 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.

5.4 Missing Invert and Edge Umasks

While libpfm4/perf_event supports the "invert" and "edge" qualifiers on Intel processors, PAPI does not report these during enumeration. This entry is to remind me to add support for these; they are considered extended attributes but to add support we have to make sure not to include the "user" or "kernel" attributes.

6 Removing the 16-component limit

Currently the component is specified by a 4-bit field in the eventcode. This limits the number of compiled-in components to 16. We are going to remove this limitation.

This is a multi-step process:

- Remove calls to `PAPI_COMPONENT_INDEX()`, `PAPI_COMPONENT_MASK()` and `PAPI_COMPONENT_AND_MASK()`
- Replace `PAPI_COMPONENT_INDEX(eventcode)` calls with an as-yet not written `papi_which_component(eve` call. This call will be slower than the simple shift/mask of the previous implementation. The new implementation will probably have to do a `_papi_ntv_code_to_info()` and requires a lot of the proposed change to all-string events to be completed.

ABI breakage: no
CDI breakage: maybe, if components are making assumptions based on event bits
API breakage: no
Implemented: N/A

7 PAPI 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 substrate 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).

The transition to all named events is a multi-step process:

- First, utilities (such as `papi_native_events` and `papi_xml_event_info`) should stop reporting event values (done in PAPI 4.2.1). Since event codes are internal only and generated on the fly, there's no guarantee that the event codes reported in one run of a tool will be the same during the next run.
- Second, code assumptions based on event bit-fields need to be removed. This is primary the assumption that the component number can be found from the 4 almost upper 4-bits of the event code.

- Third, some functions will have to be changed to have a component field. This work was done incompletely when component support was added. Most notably, `PAPI_enum_event()` needs to be changed.

There are some other things that need to be worked out. 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.

ABI breakage: no
CDI breakage: no
API breakage: no
Implemented: N/A

7.1 Component Name Prepending

We've been discussing stealing the libpfm4 naming convention and having an optional component name pre-pended to events. This will make it more clear which events are which, and handle cases where multiple components have the same name. This could lead to events like `cpu::INSTRUCTIONS_RETIRED:ALL` and `gpu0::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: N/A

8 Error Propagation

There have been separate discussions by the PAPI developers on this topic. Here's a quick summary of the two issues involved:

- PAPI's error functions (`PAPI_perror()` and `PAPI_strerror()`) should be closer to their POSIX equivalents, and

- 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

ABI breakage: yes
CDI breakage: yes
API breakage: yes
Implemented: N/A

9 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.

10 Extended User-events

Should we always enable User-Defined 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?

11 Virtualized Events (PAPI-V)

There are many issues to be resolved before PAPI-V (PAPI for virtualized systems) can be declared finished. Here's a brief sampling.

It is possible to detect (usually) if we're running inside of a VM by checking a flag in `/proc/cpuinfo` on Linux. We should set a value in the hardware structure. In theory there's a special cpuinfo interface for getting VM name and version. We should report that too.

In perf_events there is the notion of Software Events that can be safely used inside of a VM even if there is no HW-event virtualization. Should we add a way to tell if an event is a SW event at enumeration time? This also can be used in the workaround to avoid kernel crashes on older Linux kernels where overflow on SW event would crash things.

Should we make attempts to report real wall-clock time to users, in addition to the standard real and virtual time? Are our virtual time routines confusingly named now that Virtual Machine virtualization has taken over the term virtual?

As various VMs are adding support for virtualized and para-virtualized performance counters accessible from the guest, we should test and make sure PAPI runs under them.

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: reporting we are in VM: d7496311119

12 Dynamic Frequency-Scaling; Removing use of MHz in PAPI code

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

Some of the PAPI code in fallback mode uses the MHz to estimate cycles if a RDTSC-style counter is not available. We should remove all uses of MHz like this unless the platform it is running on guarantees this will stay constant.

The various PAPI tests also like to make assumptions based on the MHz value which should be removed or otherwise worked around.

We should add a `minimum_mhz` and `maximum_mhz` value to the `hw_info_t` structure. These can be set based on the files in `/sys`. This will take into account DVFS, as well as stranger things such as heterogeneous processors such as the ARM BIG.little. The old MHz values can probably be kept to preserve backwards compatibility, but its use should be deprecated.

ABI breakage: yes

CDI breakage: no

API breakage: yes

Implemented: N/A

13 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.

To address this it might be useful to have lazy initialization of the component infrastructure. The component should only be initialized if a user tries to create an eventset using that component.

Currently PAPI initializes all components at `PAPI_library_init()` time, even if they are never used. Ideally events could be enumerated even if a component was not initialized. This is not possible for components that gather the event names using an external library (like CUDA). The slowdown could also be mitigated if the component was always specified in the event name (using the :: syntax) but for backward-compatibility reasons we do not require this.

A short-term solution might be to have initialization only happen at enumeration time or event add time, but this would impact users who might assume event-add is a quick process.

Another suggested solution is that a low-level interface could be added that changes init into a two-stage process. First the library is initialized enough to see which components are available. Then a second init function is called that either enables all components (the way things currently work) or else just a selected few.

ABI breakage: maybe

CDI breakage: no

API breakage: maybe

Implemented: N/A

14 Properly Detecting Component Availability

Currently if a component is compiled in, it is assumed that the hardware the component accesses is available and working. This is not always the case, especially on distributions that enable as many components as possible and run on various machines.

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.

A proper solution is to just remove the component from the `_papi_hwd[]` structure if `init_substrate()` fails. The code to do this is very small, as shown in a patch sent to us by Bull.

ABI breakage: yes

CDI breakage: yes

API breakage: yes

Implemented: yes 6b18415879c16f

15 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.

16 Other Issues for Discussion

16.1 More Code Coverage

Code that is not compiled quickly breaks. The PAPI code is littered with #ifdefs 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.

16.2 Removing Obsolete Components/Substrates

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

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

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

Any-null support has been dropped.

16.3 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.

16.4 Locking Changes

The locking code is a bit of a mess. A big chunk (including the papi_lock.h header) can go away with the any-null removal.

The actual declaration of the locks (for Linux at least) should be moved out of the libpfm4 code and into the Linux generic code.

Fallbacks for using POSIX pthread mutexes should be added. This is very useful when using Valgrind to debug the code. This will need a configure switch to enable. (It is not a huge change to the code, as currently PAPI always links with pthreads on Linux).

ABI breakage: no

CDI breakage: no

API breakage: no

Implemented: yes: 92689f626b

16.5 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?

16.6 Outside Comments

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.