Journal Title
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

PAPI Software-Defined Events for in-depth Performance Analysis

Heike Jagode[1], Anthony Danalis[1], Hartwig Anzt[1,2] and Jack Dongarra[1,3,4]

Abstract

The methodology and standardization layer provided by the PAPI performance monitoring library has played a vital role in application profiling for over a decade. It has enabled sophisticated performance analysis tool designers and performance-conscious scientists to gain insights into their applications by simply instrumenting their code using a handful of PAPI functions that "just work" across different hardware components.

In the past, PAPI development had focused primarily on hardware-specific performance metrics. However, the rapidly increasing complexity of software infrastructure poses new measurement and analysis challenges for the developers of large-scale applications. In particular, acquiring information regarding the behavior of libraries and runtimes—used by scientific applications—requires low-level binary instrumentation, or APIs specific to each library and runtime. No uniform API for monitoring events that originate from inside the software stack has emerged.

In this paper, we present our efforts to extend PAPI's role so that it becomes the de facto standard for exposing performance-critical events, which we refer to as Software-Defined Events (SDEs), from different software layers. Upgrading PAPI with SDEs enables monitoring of both types of performance events—hardware- and software-related events—in a uniform way, through the same consistent PAPI interface. The goal of this paper is threefold. First, we motivate the need for SDEs and describe our design decisions regarding the functionality we offer through PAPI's new SDE interface. Second, we illustrate how SDEs can be utilized by different software packages, specifically, by showcasing their use in the numerical linear algebra library MAGMA-Sparse, the tensor algebra library TAMM that is part of the NWChem suite, and the compiler-based performance analysis tool Byfl. Third, we provide a performance analysis of the overhead that results from monitoring SDEs and discuss the trade-offs between overhead and functionality.

Keywords

PAPI, SDE, Software-defined Events, libraries, instrumentation, performance, NWChem, MAGMA, Byfl

Introduction

Developing applications using some form of a modular, or layered, design—where different logical operations are performed by different, smaller units of a large, complex application—is not only a good software engineering principle, but is also common practice across diverse fields. Focusing on the field of High Performance Computing (HPC), we see that the community has moved away from the large monolithic FORTRAN codes that dominated the field a few decades ago, and has adopted more structured designs, which foster code reuse as well as closer collaboration among different academic groups. Besides adhering to good software engineering principles, this transition was necessitated by the increasing complexity of hardware platforms, which transitioned from single node machines to distributed-memory heterogeneous supercomputers. As a result, many modern HPC applications are not only internally organized in smaller units, but also use external libraries for functions such as communication and synchronization (with the Message Passing Interface [MPI] being the leading choice), runtimes for on-node parallelism (such as OpenMP), and a plethora of external libraries for functions such as math, or access to accelerators. In the rest of this paper, we will discuss performance aspects of such applications, and we will use the term "module" for any code entity, such as library, runtime, class, etc., which can be used as a building block of a larger application.

In HPC, where application performance is critical, there is a drawback to adopting a design that is non-monolithic. Specifically, the developers of one module lack information regarding the internal behavior of modules they deploy but did not develop themselves. For example, when using a communication library, such as MPI, the application developer does not know if the actual data transfer of a non-blocking call (such as `MPI_Isend()`) took place when that function was called, or was postponed until the matching `MPI_Wait()` was called. Similarly, when using a numerical linear algebra library, the domain scientist does not know the algorithm-specific characteristics, such as the residual or the number of sparse matrix-vector multiplications performed during the execution of an

[1]University of Tennessee, Knoxville, USA
[2]Karlsruhe Institute of Technology, Germany
[3]Oak Ridge National Laboratory, USA
[4]University of Manchester, UK

Corresponding author:
Heike Jagode, Innovative Computing Laboratory, University of Tennessee, Suite 203 Claxton, 1122 Volunteer Blvd, Knoxville TN 37996, USA.
Email: jagode@icl.utk.edu

application. In summary, when complex applications are properly structured in multiple modules, the lack of information exchange between different modules can lead to sub-optimal interactions between different modules, which, ultimately, leads to loss of performance.

Some projects with wide adoption, such as MPI and OpenMP, have been developing a "tools' interface" (MPI_T Islam et al. (2016) and OMPT Eichenberger et al. (2013) respectively). This is an effort to create custom hooks inside the libraries that implement these standards, such that external performance analysis tools can use these hooks to extract information about the internal behavior of aforesaid libraries. Efforts like these offer a solution to the problem of exchanging information between modules without breaking the modularity of complex applications. However, developing library-specific APIs is not a scalable approach. It is not feasible for every group of library developers to establish their own API and expect scientific application developers or performance analysis toolkits to adopt them all.

As a solution to this problem, we have developed an API for exporting Software-Defined Events (SDEs) through the Performance Application Programming Interface (PAPI) Terpstra et al. (2009). Being the de facto standard middleware layer for hardware performance events, with wide adoption by toolkits and application codes, and wide availability on most software stacks, PAPI is the perfect vehicle for supporting and delivering generic SDEs.

In the rest of this paper, we discuss the design decisions we have made regarding the functionality of the new PAPI SDE interface, we illustrate the concept and usefulness of SDEs in modern libraries, and we evaluate the performance effect of adding PAPI SDEs in HPC software layers.

Design and Functionality

The main functionality of PAPI Software-Defined Events is to expose performance-critical properties and the internal behavior of black-box modules in the software stack to the top-level applications, or performance analysis toolkits. PAPI has a near-ubiquitous presence on modern HPC systems, and wide support by performance analysis tools. Consequently, if library developers choose to take advantage of PAPI SDEs to export internal, performance-relevant characteristics to the outside world, they can expect this information to be used for more sophisticated performance analysis of applications that use their library. Furthermore, developers of scientific applications with heavy use of multiple external libraries, can take advantage of library-specific SDEs in order to track and fix performance issues resulting from poor coordination between different libraries.

The developers of a given software module are experts in both the semantics of their module and its implementation. For this reason, we designed the SDE functionality so the experts can export whatever information they deem important, and wherever they think it should be exported from. We do not attempt to dictate the type of information that should be exported as an SDE. Instead, we offer a generic mechanism and middleware layer that allows any kind of information to be exported.

As an illustration, iterative solvers from a math library could export the number of iterations it took until convergence was attained. Furthermore, algorithm-specific properties, such as the residual computed during each iteration, could be exported and potentially used to extrapolate when the solver is expected to reach its convergence criterion. As another example, a climate code could export a performance metric of "emulated years per second". In other words, PAPI SDEs were not designed for performance experts to annotate a third-party code so they can measure how long it took for a code segment to execute. There are other tools for this type of annotation. Rather, PAPI SDEs are meant to be used by the developers who write a software module so that the internal behavior of their module can be understood better by those who use it.

Design Decisions

Since the early design and development stages of PAPI SDEs, we interacted heavily with members of the performance analysis toolkit community, as well as developers of libraries and runtimes which would make natural targets for early adoption. These interactions revealed two principal concerns.

- Performance analysis toolkit communities strongly emphasized the importance of **preserving the existing API** that is currently exported by PAPI for measuring hardware events.
- Library and runtime communities were mostly concerned with the **performance overhead** caused by the introduction of SDEs in their codes.

Since PAPI is positioned as a *middleware layer*, success depends on adoption by other software modules and toolkits. As a result, we used the concerns raised by the community to guide our design decisions.

To address the requests made by developers of toolkits, we implemented SDE support in PAPI as a new component, which provides an API for registering SDEs into the existing PAPI framework. After SDEs have been registered, they can be accessed using the same API that has always been used for accessing hardware events through PAPI. In other words, we created a new API for library writers to register their software events with PAPI, but we maintained the existing API (e.g., `PAPI_start()`, `PAPI_read()`, `PAPI_stop()`) for users and toolkits to monitor these events.

Satisfying the performance concerns of library developers was more challenging, since there is often a tradeoff between overhead and functionality. Our approach to handling the tradeoff is multi-faceted and enables the library developers to make the choices that best fit the requirements of their code. Specifically, we provide several types of SDEs that strike a different balance between overhead and functionality. We discuss these SDE types in the following sections.

PAPI SDE Counter Types

1. **Registered counters** offer developers the ability to register an existing internal variable of their library as a PAPI counter for an SDE. The registration of such a counter happens only once, presumably during the initialization of the library, so the performance

overhead due to the registration is constant and negligible. Through this SDE type, PAPI enables entities outside a library to read a variable that already existed in the library and was updated by library code when needed. Therefore, no additional work needs to be performed by the library code in order to support this type of SDE.

2. **Registered function pointers** offer library developers the ability to register a special-purpose function: one that is internal to the library and invoked by PAPI. When called, the function acts as an accessor to deliver the counter value. This type of SDE is useful when a library has no pre-existing variable that acts as a counter for an event. It is also useful in case the value of the event counter must be derived from a complex internal state of the library rather than a single variable. The registration of a function pointer happens only once, so the performance overhead of the registration is constant and negligible. For this type of SDE, the registered function is only used to read a value, not to update a counter, and therefore does not have to be called by the library. Instead, it is only called by PAPI when the application using the library makes a call to `PAPI_read()`. Therefore, this type of SDE does not add overhead to the *fast path* of the library. In other words, when the library is involved in a "maximum performance run," where PAPI is not used to monitor its behavior, the registered function does not get called.
3. **Created counters** offer library developers the flexibility of creating a counter inside PAPI, instead of having a counter inside the library. The creation of such a counter happens only once, but updating the value of a created counter requires a call to a PAPI SDE function. This type of SDE has two main benefits over registered counters. First, created counters are always thread safe. As a result, using a created counter relieves the library developers of the need to use explicit thread safe code every time they update the counter. Second, since PAPI is aware of every update of the counter value, this type of SDE lends itself to more accurate overflow support, or other types of notification of performance analysis toolkits. The drawback of this type of SDE is that it requires a call to a PAPI SDE function inside the library code every time the event counter needs to be updated, and therefore has higher overhead than registered counters.
4. **Recorders** offer library developers the ability to record a series of values associated with an event. Similarly to created counters, the memory associated with a recorder is managed internally by PAPI, and the creation of a recorder happens only once, but a PAPI SDE function needs to be called for every new value that is being recorded. PAPI makes no assumptions about the type of the variable being recorded. Instead, the API requires only a pointer to the variable and the size of the variable. This way, library developers are free to record any type of data from simple integers to complex structures—or even strings, or arrays of values.
   When a recorder is created, PAPI automatically creates a few additional auxiliary counters. The first has the same name as the recorder with the additional suffix "`:CNT`". This counter holds the count of elements that have been recorded at any given time. In addition, there are five more auxiliary counters automatically created for each recorder. When read, these counters return the quantiles of the recorded distribution, and in particular the minimum and maximum values recorded, as well as the three quartiles (i.e., 25%, median, 75%). The names of these counters are formed by adding one of the prefixes {`:MIN`, `:Q1`, `:MED`, `:Q3`, `:MAX`} to the name of the recorder. In contrast with the auxiliary counter "`:CNT`", these statistical counters are optional and depend on the ability of PAPI to compare the recorded values. Since the recorded values can be of arbitrary type, when a library creates a recorder it is expected to provide a pointer to a function that is able to compare two values of the type that is recorded. If the function pointer is `NULL`, then the statistical counters are not created.
5. **Groups** offer library developers the possibility of aggregating the values of multiple counters into a single entity. Groups are implemented as first-class citizens and can be added into larger groups recursively. When a library creates a group it must specify whether the value that is reported when this group counter is read consists of the minimum, the maximum, or the sum of the values of the counters which belong to the group. Both registered and created counters can be added to groups, but recorders can not be added to groups. However, the auxiliary counters associated with a recorder could be added to groups. In terms of performance overhead, groups do not require any additional code to be inserted in the fast path of a library. The value of a group is assembled when a user application calls `PAPI_read()` by reading the values of all the counters that belong to the group.

PAPI SDE Application Programming Interface

While the previous sections focus on the concepts of the SDEs and their usefulness and usability by libraries, this section covers details about the actual API. The PAPI SDE API calls are only meant to be used inside libraries to export software-defined events from within those libraries. As of today, all API functions are thread-safe, and available in `C` and `FORTRAN-2008`.

From the domain scientists' perspective, questions like (a) what constitutes a *software counter*, and (b) where, in a scientific application, is the right place to log a counter, are hard to answer. It is important to note that it is a design decision of the SDE API to *not* answer these questions, but instead, to offer a generic API that can be used in diverse ways by different libraries and applications alike. As discussed in Section "Design Decisions", our design gives full control to the library developers.

On the other hand, the API for *reading* SDEs remains the same as the standard API for reading hardware events, i.e., `PAPI_start()` and `PAPI_stop()`. In that sense, nothing changes for applications and tools that already have PAPI hooks in their software. Instead, they will automatically inherit the SDE functionality.

```
void *papi_sde_init(const char *lib_name);
```

The first function that must be called by a library is `papi_sde_init()` and it has the specification shown above. This function is called only once to initialize internal data structures, and returns an opaque handle that must be passed to all subsequent calls to PAPI SDE functions.

- `lib_name` is a string containing the name of the library.

```
int papi_sde_register_counter(
    void *handle,
    const char *event_name,
    int mode,
    int type,
    void *counter);
```

For every program variable that the library wishes to register as an event counter, the function `papi_sde_register_counter()` must be called.

- `handle` is the opaque handle returned by `papi_sde_init()`.
- `event_name` is a string containing the name of the event being registered.
- `mode` is an integer declaring whether a counter is read-only or read-write. This is a way to give a software layer access to internal variables of a different software layer, and it is a feature that can be particularly useful to auto-tuning efforts. Additionally, it specifies whether the count mode is "instantaneous" or "delta".
- `type` is an enumeration of the type of the event.
- `counter` is a pointer to the actual variable that serves as the counter for this event. The type is "`void *`" to enable support for user-defined types.

```
typedef long long (*papi_sde_fptr_t)(void*);
int papi_sde_register_fp_counter(
    void *handle,
    const char *event_name,
    int mode,
    int type,
    papi_sde_fptr_t fp_counter,
    void *param);
```

One can imagine the case where a library wishes to export an event whose value does not map to the value of a single variable of the library. For example, suppose we have the case where different threads of a library are counting an event independently, but the exported event is the total count across all threads. In this case the actual count needs to be "assembled" when the user requests to read it. For such cases we provide the function `papi_sde_register_fp_counter()` for registering a function pointer to an *accessor* function provided by the library.

- `fp_counter` is a pointer to the accessor function. The accessor function has the return type "`long long int`" to conform to the existing API of PAPI.
- `param` is an opaque object that the library passes to PAPI, and PAPI passes it as a parameter to the accessor function every time it is called. This opaque parameter gives library developers the flexibility to pass custom data structures to their accessor functions when a counter is read.

```
int papi_sde_unregister_counter(
    void *handle,
    const char *event_name);
```

Can be called to unregister an event counter. Useful for implementing transient events.

```
int papi_sde_add_counter_to_group(
    void *handle,
    const char *event_name,
    const char *group_name,
    uint32_t group_flags);
```

Adds a counter to a group so that logical groups can be formed out of multiple related event counters. Groups are first-class citizens and can be recursively added to other groups. A group is automatically created the first time a counter is added to it.

- `group_name` is a string containing the name of the group.
- `group_flags` specifies whether the group should report the sum, the min, or the max of the counters it contains.

```
int papi_sde_create_counter(
    void *handle,
    const char *event_name,
    int type,
    void *counter_handle);
```

Creates a counter whose memory is managed by PAPI (instead of the library).

- `counter_handle` is an opaque handle that can be used to access the created counter.

```
int papi_sde_inc_counter(
    void *counter_handle,
    long long increment);
```

Increments the value of a created counter.

- `counter_handle` is the opaque handle returned by `papi_sde_create_counter()`.
- `increment` is the value to be added to the counter.

```
int papi_sde_reset_counter(
    void *counter_handle);
```

Resets the value of a created counter. After this function is called, the value of the counter is zero.

- `counter_handle` is the opaque handle returned by `papi_sde_create_counter()`.

```
int papi_sde_create_recorder(
    void *handle,
    const char *event_name,
    size_t typesize,
    void *record_handle);
```

Creates a multi-value SDE (recorder) that can record (log) a series of values. The memory of the recorder is handled internally by PAPI.

- `typesize` is the size of each element (to be recorded) in bytes.
- `record_handle` is an opaque handle that can be used to access the created recorder.

```
int papi_sde_record(
    void *record_handle,
    size_t typesize,
    void *value);
```

Records an element into a recorder that was created via `papi_sde_create_recorder()`.

- `record_handle` is the opaque handle returned by `papi_sde_create_recorder()`.
- `typesize` is the size of the new element in bytes.
- `value` is a pointer to the new element.

```
int papi_sde_reset_recorder(
    void *record_handle);
```

Resets the recorder by setting the number of recorded entries to zero. This function neither frees the allocated space nor zeros it. It only allows future invocations of `papi_sde_record()` to reuse the memory allocated for the recorder, and overwrite any previously recorded elements.

- `record_handle` is the opaque handle returned by `papi_sde_create_recorder()`.

```
void *papi_sde_get_counter_handle(
    void *handle,
    const char *event_name);
```

Given the opaque handle returned by `papi_sde_init()` and the name of a created counter (or a recorder), it returns the `counter_handle` of the created counter (or the `record_handle` of the recorder).

- `event_name` is a string containing the name of the event associated with a created counter, or recorder.

```
int papi_sde_describe_counter(
    void *handle,
    const char *event_name,
    const char *event_description);
```

For every SDE registered by a library, the library has the option to pass an additional string that contains a more elaborate description of the event. This string will be printed when the PAPI utility `papi_native_avail` is called. Associating a description with a registered SDE happens through the function `papi_sde_describe_counter()`.

- `event_description` is a string containing the description of the event.

```
void * papi_sde_hook_list_events(
    papi_sde_fptr_struct_t *fptr_struct);
```

The function `papi_sde_hook_list_events()` is not an API function provided by PAPI. Instead, it is an optional function that libraries should implement as a hook for the tool `papi_native_avail` to be able to list all SDEs in a library. This function is supposed to call the API functions described above to register all the SDEs the library wishes to register.

- `fptr_struct` is a structure containing pointers to all SDE functions.

Overhead-Functionality Tradeoff

The design we have adopted gives full control of the overhead-functionality tradeoff to the library developers. Each group can choose if the functionality provided by a feature justifies the amount of performance overhead this feature will add to their library, or if they want to limit the SDE types they will utilize to those with zero overhead.

Libraries that already count internal quantities and events, but do not have a standardized way to export this information to the outside world, will benefit from PAPI SDEs while facing zero performance overhead. Libraries with no event counting functionality can benefit by adding internal counters, and accessor functions can still be done with negligible or zero performance overhead. Libraries with events that do not occur frequently enough for performance overhead to be of primary concern can use created counters to communicate these events to users, or to analysis toolkits that are "listening." And finally, libraries with events whose evolution over time is important can record long series of custom event values for advanced analysis by performance-conscious users and sophisticated toolkits.

Users of PAPI-SDEs

Through significant interaction with members of the performance analysis community, as well as developers of scientific libraries and applications, the early adoption of SDEs has found its way into a number of different software layers. This section illustrates how SDEs can be utilized by different software packages, specifically, by showcasing their use in the compiler-based performance analysis tool Byfl, the sparse linear algebra library MAGMA-Sparse, and the Tensor Algebra for Many-body Methods (TAMM) library that is part of the NWChem suite.

Case I: MAGMA-Sparse

For domain scientists working with complex simulation codes, it is a burden to track down performance bottlenecks or numeric properties in the numeric software backends. In particular, if simulation codes utilize different numerical linear algebra (NLA) libraries in a recursive fashion, expert knowledge is necessary to extract algorithm-specific characteristics such as Sparse Matrix-Vector Product (SpMV) count, the residual or iteration count of a backend solver. At the same time, this information can be very useful to identify critical sections, optimize the code stack, or assess whether or not swapping a backend NLA library promises performance benefits. The lack of a standard that specifies how this information can be accessed in iterative sparse linear algebra libraries is frustrating to say the least. The alternative for domain scientists is to browse the library-specific documentation and/or the code to identify the entry points for gathering solver-specific details.

We have been working on improving this situation by augmenting MAGMA-Sparse Anzt et al. (2017)—a collection of solvers for sparse linear systems—with SDEs that are exposed though the PAPI interface. Table 1 provides the list of MAGMA solvers that now benefit from SDEs, and Table 2 lists the registered events and their descriptions as they are available through PAPI. This enables domain scientists to monitor the behavior of low-level linear algebra algorithms without needing expert knowledge about the full software stack of the simulation code.

BiCG	Biconjugate Gradient Method
PBiCG	Preconditioned Biconjugate Gradient Method
BiCGStab	Stabilized Biconjugate Gradient Method
PBiCGStab	Precond. Stabilized Biconjugate Gradient Method
CG	Conjugate Gradient Method
PCG	Preconditioned Conjugate Gradient Method
CGS	Conjugate Gradient Squares Method
PCGS	Precond. Conjugate Gradient Squares Method
GMRES	Generalized Minimal Residuals
PGMRES	Preconditioned Generalized Minimal Residuals
IDR	Induced Dimension Reduction Solver
PIDR	Precond. Induced Dimension Reduction Solver
QMR	Quasi-Minimal Residual
PQMR	Preconditioned Quasi-Minimal Residual
TFQMR	Transpose-free Quasi-Minimal Residual
PTFQMR	Precond. Transpose-free Quasi-Min. Residual
IterRef	Iterative Refinement method
LOBPCG	LOcally Optimal Block Preconditioned CG (iterative eigensolver)

Table 1. MAGMA-sparse solvers that ship with SDE support.

As discussed in the "PAPI SDE API" Section, the current PAPI-SDE component supports different SDE "types" and "modes" when registering an SDE counter. These types can be 32-bit integer values (int), 64-bit integer values (long long), 32-bit fractional values (float), or 64-bit fractional values (double). Similarly, currently supported count modes are "instantaneous" or "delta".

Our chosen naming scheme for SDEs (as seen in Table 2) makes it easy to identify the "type" of the counter, as well as its category (single- vs. multi-value). Two of the MAGMA counters are registered as 64-bit integer (long long), single-value SDEs, while the three residual counters and the runtime counter are 64-bit fractional (double), single-value SDEs. Furthermore, the `RCRD` string in the event name helps to identify multi-value SDEs (recorders), which, ultimately, need to be handled differently. As for the counting "mode," all MAGMA SDEs are registered as instantaneous counters.

Usage Examples: Using SDEs allows for easy monitoring of the characteristics and behavior of the NLA backend algorithms for a specific problem handled by the top-level application. The examples in Figures 2–4 illustrate how the convergence of Krylov solvers can be visualized with the help of PAPI SDEs. Each graph in these figures shows the convergence of eight solvers, and each figure uses a different matrix from the SuiteSparse Matrix Collection Davis and Hu (2011). Figures 1–3 contain two graphs each. The one on the left shows the case where preconditioned solvers were used (using incomplete LU factorization [ILU] as the preconditioner), and the one on the right shows the case where the solvers were not preconditioned.

At a quick glance, one can observe from all graphs that the iterative residual from the GMRES solver (yellow line) periodically spikes to the value of the initial residual. This, however, is expected since the GMRES algorithm periodically restarts its operation.

Looking at Figure 2 specifically, we can observe that QMR and TFQMR converge at the same rate as most others for about 400–500 iterations, but then suddenly stop converging further. This behavior is indeed unexpected and may indicate a problem with those particular solvers. Furthermore, in what pertains to this paper, this insight could not be gathered before by merely examining the final residual, but now it can be gained in a simple and systematic way through PAPI without the need to modify the code of these solvers.

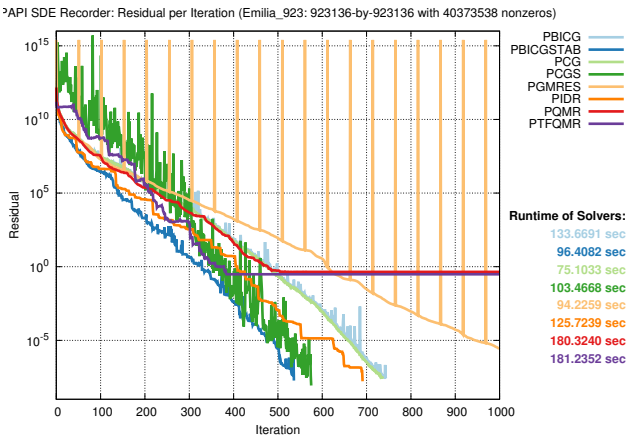
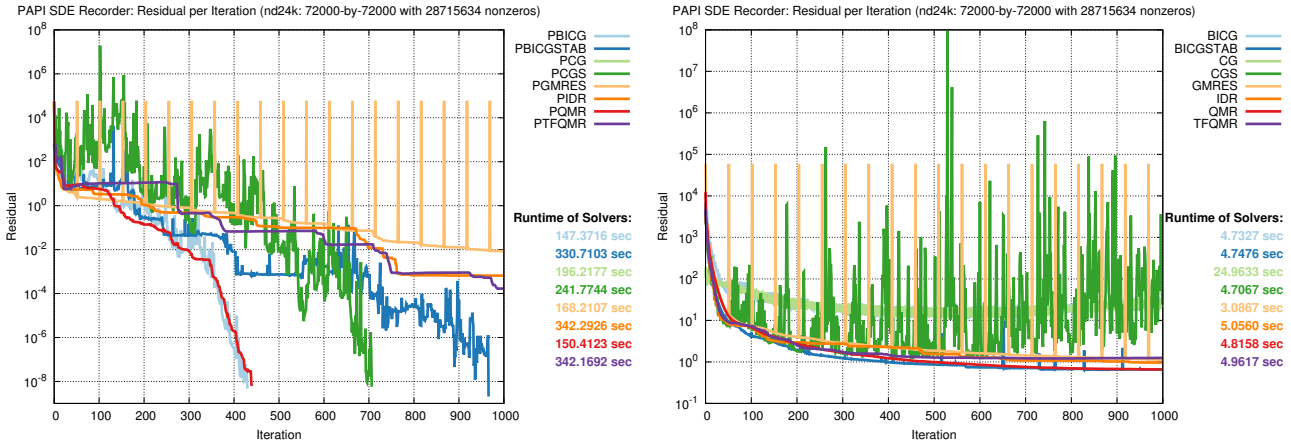


Figure 2. PAPI SDE-Recorder logging convergence of different ILU-preconditioned MAGMA solvers for a Structural Problem. (`https://sparse.tamu.edu/Janna/Emilia_923`)

SDE Name (prefixed with `sde:::MAGMA::`)	SDE Description
`numiter_I`	Number of iterations until convergence attained (I=integer)
`SpmvCount_I`	Number of sparse matrix-vector multiplications (SpMV) (I=integer)
`InitialResidual_D`	Initial residual (D=double)
`FinalResidual_D`	Final residual (D=double)
`IterativeResidual_D`	Iterative residual (D=double)
`SolverRuntime_D`	Total run-time of the solver (D=double)
`IterativeResidual_RCRD_D`	Array of all residuals until convergence (RCRD=recorder) (D=double)

Table 2. Registered SDEs in MAGMA to enable users to gather solver-specific details though PAPI.



(a) ILU-preconditioned **(b)** Unpreconditioned

Figure 1. PAPI SDE-Recorder logging convergence of different MAGMA-sparse solvers for a 2D/3D Problem. (`https://sparse.tamu.edu/ND/nd24k`)

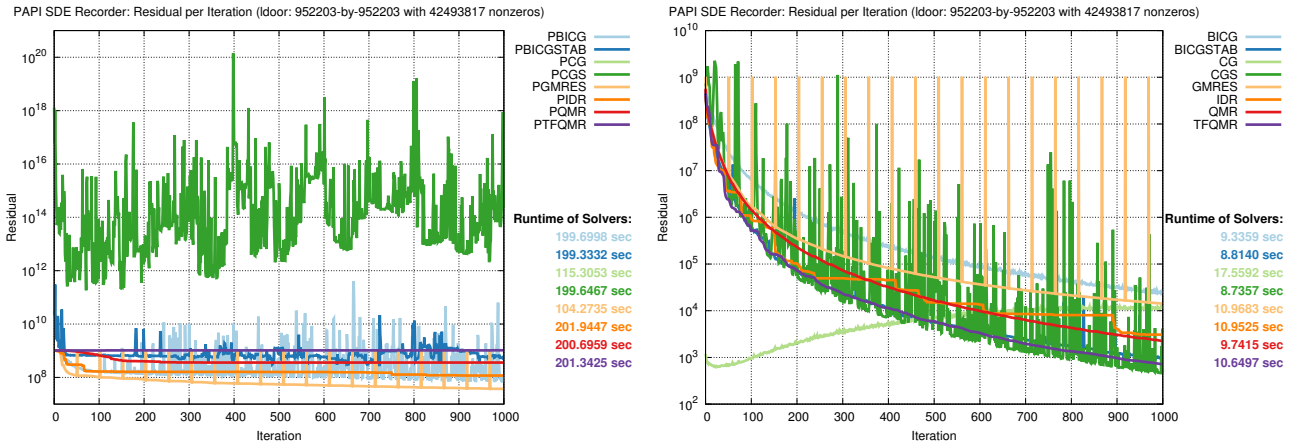
MAGMA Solver	numiter_I	SpmvCount_I	InitialResidual_D	FinalResidual_D	SolverRuntime_D (sec)
PBICG	431	862	5.9253e+04	5.0977e-09	1.4737e+02
PBICGSTAB	967	1934	5.9253e+04	8.1089e-08	3.3071e+02
PCG	1000	1000	5.9253e+04	-nan	1.9622e+02
PCGS	707	1414	5.9253e+04	1.2268e-05	2.4177e+02
PGMRES	1000	1000	5.9253e+04	8.4847e-03	1.6821e+02
PIDR	1000	2000	5.9253e+04	6.5693e-04	3.4229e+02
PQMR	440	880	5.9253e+04	6.5494e-09	1.5041e+02
PTFQMR	1000	2000	5.9253e+04	8.2972e-03	3.4217e+02
BICG	1000	2000	5.9253e+04	3.1236e+01	4.7327e+00
BICGSTAB	1000	2000	5.9253e+04	6.4980e-01	4.7476e+00
CG	1000	1000	7.7422e+01	2.5707e+01	2.4963e+01
CGS	1000	2000	5.9253e+04	2.4101e+01	4.7067e+00
GMRES	1000	1000	5.9253e+04	1.1280e+00	3.0867e+00
IDR	1000	2000	5.9253e+04	9.6088e-01	5.0560e+00
QMR	1000	2000	5.9253e+04	6.5533e-01	4.8158e+00
TFQMR	1000	2000	5.9253e+04	1.2484e+00	4.9617e+00

Table 3. PAPI single-value SDEs of different MAGMA-sparse solvers for a 2D/3D Problem. (`https://sparse.tamu.edu/ND/nd24k`)

Figure 1 considers a different matrix and plots the results from the MAGMA multi-value SDE that logs the iterative residual for each solver. For the sake of completion, Table 3 summarizes the results of all MAGMA single-value SDEs for this particular problem. The left graph in Figure 1 depicts nicely how differently these solvers behave in terms of convergence. For example, when using PCGS (dark green line) the residual first grows, but then the solver manages to converge within about 700 iterations, whereas other solvers, such as PIDR (orange line), start better but fail to converge within 1,000 iterations.

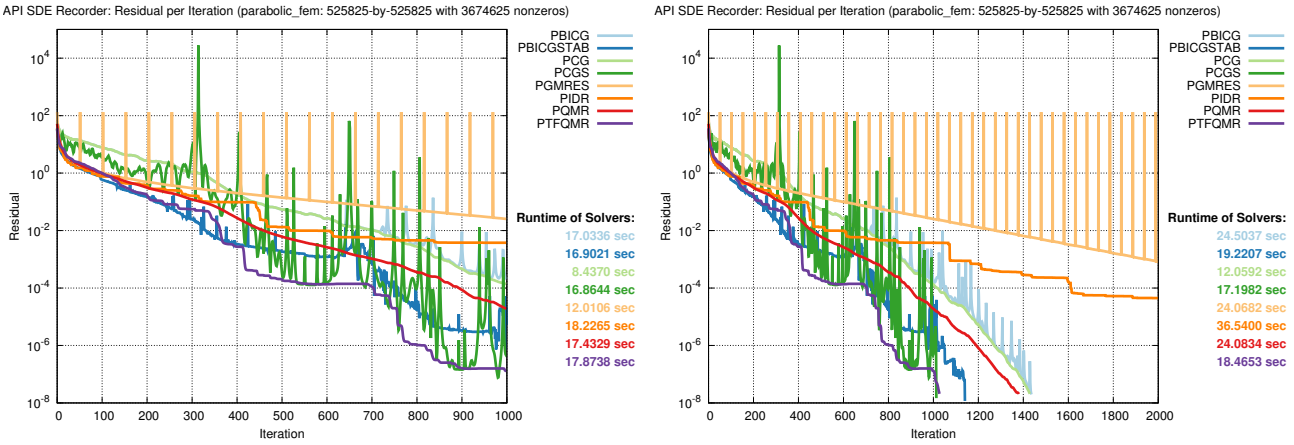
By examining the graph on the right side of Figure 1, we can easily see the benefits of preconditioning, since none of the unpreconditioned solvers manages to converge for this matrix. Notice that the Y-axes of the two graphs are not to scale, and after 1,000 iterations the unpreconditioned solvers have stagnated at a level the preconditioned solvers had reached in the first few hundred iterations.

However, the benefit of any given preconditioner is not universal. As can be seen by examining Figure 3, the use of the ILU preconditioner for this particular matrix (shown on the left) led to significantly higher residual values than



(a) ILU-preconditioned **(b)** Unpreconditioned

Figure 3. PAPI SDE-Recorder logging convergence of different MAGMA-sparse solvers for a Structural Problem. (https://sparse.tamu.edu/GHS_psdef/ldoor)



(a) ILU-preconditioned: 1,000 iterations **(b)** ILU-preconditioned: 2,000 iterations

Figure 4. PAPI SDE-Recorder logging convergence of different MAGMA-sparse solvers for a Computational Fluid Dynamics Problem. (https://sparse.tamu.edu/Wissgott/parabolic_fem)

the case of the unpreconditioned solvers (shown on the right). Anomalous behaviors like these create a unique opportunity for users of libraries that contain PAPI SDEs. In particular, a user application could initiate one of these ILU-preconditioned solvers and start monitoring the residual using PAPI. Upon realizing that the residual has jumped to values as high as 10^{14} it could stop the solver instead of waiting for the default 1,000 iterations to finish. Furthermore, since the quality of preconditioners depends on the specifics of the matrix being solved, domain scientists with knowledge about the nature of their matrix could implement their own preconditioner and study its behavior using the PAPI SDEs found inside MAGMA, instead of having to modify MAGMA code.

By using PAPI SDEs, application scientists can examine multiple intermediate values of the residual instead of only the final value. This way, one can assess the trend of the residual's evolution. In the left graph of Figure 4 we see that none of the solvers converges within the default 1,000 iterations. However, being able to see the trend, we could reasonably extrapolate from the solvers' slope that at least a few of them, such as PTFQMR and PBICGSTAB, will converge if given a few more iterations. Indeed, in the right graph of Figure 4 we see, when given more iterations, all the solvers predicted to converge reach convergence shortly after the 1,000 iteration mark.

This graph also allows for a different kind of optimization. Namely, if we examine the behavior of individual solvers, for instance PTFQMR and PCGS, we can observe that, in many cases, the residual drops in sudden steps rather than smoothly. This means that if the nature of the application allows the user to relax their convergence criterion to a value higher than 10^{-8}, for example 10^{-6}, then these solvers would converge at about 800 iterations.

Finally, there is one unpredictable observation that can be extracted from the data in Figure 4. As can be seen in the "Runtime of Solvers" information, which we provide next to each graph, the fastest solver for this problem was PCG (light green), although it does not converge in the least amount of iterations. This fact, on its own, is a well known artifact of iterative solvers and it is not surprising. However, consider the case when a user is not using PAPI SDEs to perform this in-depth analysis, but rather relies on the single value of the "final residual", which is commonly reported by libraries like MAGMA. Now, suppose a user runs the default 1,000 iterations and only sees the final residuals, as it can be seen

on the last iteration of the left graph. In this limited view, PCG would appear to be a poorly performing choice of a solver, since its residual is higher than 10^{-4} when at least two others are around 10^{-7}. However, as we saw after allowing the solvers to run for more iterations, PCG does result in the best choice for this problem (when considering the execution time until completion).

Case II: NWChem

The concept of *software-defined events*, as well as the SDE API introduced in this paper, are not limited to libraries and performance tools. The direct use of SDEs in real-world applications can be highly beneficial for in-depth analysis or for quickly identifying performance and scalability bottlenecks. Focusing on one such real-world application, this section targets the field of electronic structure theory—a significant example worth exploring because most computational chemistry methods are already unable to take full advantage of current computer resources at leadership-class computing facilities and are bound to fall behind even further on future post-petascale systems.

To ease this condition, we focused on the implementation of SDEs that allow us to characterize the performance and the level of parallelism of the NWChem quantum chemistry application Valiev et al. (2010). Specifically, we worked with the Coupled Cluster Single Double (CCSD) methods Kowalski et al. (2011) as they are currently implemented in the TAMM library, which is expected to be part of the new C++ version of NWChemEx.

The findings of these performance metrics for several CCSD kernels are discussed below in the "Usage Examples" section. Ultimately, the objective is to have computational chemistry experts, who are aware of the scientific function of different code segments, add SDEs that correspond to physically meaningful quantities that could reveal information, such as "computed electron potentials per second".

Table 4 lists the SDE-registered NWChem events and their descriptions as they are available through PAPI. All four of the NWChem single-value SDEs are registered as 64-bit integer counters. As for the counting mode, DGEMM- and FlopCount are registered as delta counters, while the `Contraction_ID` and the `MaxTaskLength` return instantaneous values.

In addition to these single-value SDEs, we added support for two multi-values SDEs. For instance, instead

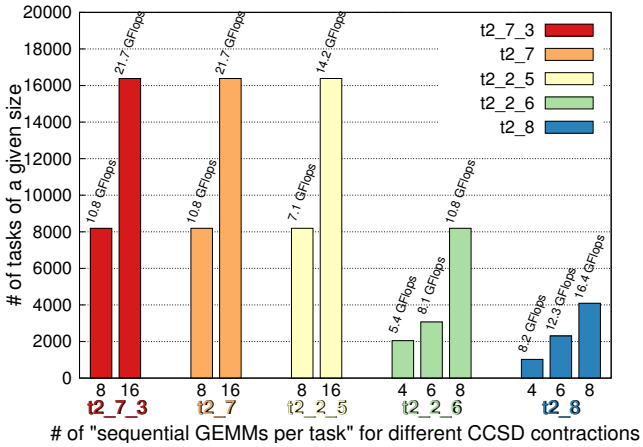


Figure 5. PAPI SDE-Recorder logging task lengths and FLOPs count for NWChem CCSD.

of just monitoring how big the biggest CCSD task in the NWChem code is, which is accomplished via `MaxTaskLength` for each tensor contraction, with the addition of `LengthPerTask_RCRD`, we can easily record the length of all tasks per contraction and study the distribution of task lengths. Similarly, the `FlopsPerTask_RCRD` recorder produces important details about the number of floating-point operations (FLOPs)computed for each sequential task.

Usage Examples: The input data is the beta-carotene molecule ($C_{40}H_{56}$) in the 6-31G basis set, composed of 472 basis set functions. In these tests, all core electrons are kept frozen, and 296 electrons are correlated.

Figure 5 shows the distribution of task lengths and FLOPs count for five contractions with the highest workload in the case of beta-carotene. The different colors in Figure 5 represent the five different contractions, labelled t2_7_3, t2_7, t2_2_5, t2_6, and t2_8. To calculate this load, we used the SDE counters that monitors the total number of floating-point operations for each CCSD contraction (e.g., `FlopCount_I`). For instance, the first contraction (t2_7_3) computes 327,680 DGEMMs (`DgemmCount`) and 32,557,729,032 FLOPs (32.5 GFLOPs) (`FlopCount`). The x-axis shows the various length of the tasks, meaning the number of sequential DGEMMs per tasks for each contraction. In the case of contraction (t2_7_3), there are two task lengths only, with 16 being the maximum number of sequential DGEMMs per task.

SDE Name (prefixed with `sde:::TAMM::`)	SDE Description
`ContractionID_I`	ID of CCSD contraction (I=integer)
`DgemmCount_I::ContrID=46`	Total number of DGEMMs for CCSD ContrID=46 (I=integer)
`FlopCount_I::ContrID=46`	Total number of floating-point operations for CCSD ContrID=46 (I=integer)
`MaxTaskLength_I::ContrID=46`	Maximum number of sequential DGEMMs per task for CCSD ContrID=46 (I=integer)
`LengthPerTask_RCRD_I::ContrID=46`	Array of the number of sequential DGEMMs per task for CCSD ContrID=46 (RCRD=recorder) (I=integer)
`FlopsPerTask_RCRD_I::ContrID=46`	Array of the number of FLOPs per task for CCSD ContrID=46 (RCRD=recorder) (I=integer)

Table 4. Registered SDEs in TAMM / NWChem to enable users to gather CCSD-specific details though PAPI.

Data from the `LengthPerTask_RCRD` recorder is plotted by the bars, and the data from the `FlopsPerTask_RCRD` is shown by the labels on top of each bar. For instance, contraction t2_7_3, which happens to also be the most compute intensive portion of the code, has more than 24,000 tasks and approx. 2/3 of these tasks compute 16 sequential DGEMMs, while the other 8,000 compute eight sequential DGEMMs. Each of these 16,000 tasks performs over 20 GFLOPs and the remaining 8,000 perform about 10.

In summary, these tasks are very expensive, which ultimately limits the scalability of CCSD. It would be beneficial to break down the computation of the CCSD methods into more fine-grained tasks, so that the serialization imposed by the traditional, linear algorithms can be transformed into parallelism, allowing the overall computation to scale to larger computational resources.

Case III: Byfl

Byfl Pakin and McCormick (2014) is a compiler-based performance analysis tool, written in C++, that relies on LLVM The LLVM Project (2018) and Clang The Clang Project (2018). Byfl works by instrumenting the application code instead of relying on sampling. It instruments an application at compile time, and then gathers and reports performance data at run time.

The motivation behind this PAPI-Byfl integration effort is very much in line with our PAPI-SDE effort. Byfl allows applications to measure—at the software level—events that are currently missing from hardware implementations, namely, bytes transferred to and from memory, and floating-point operations performed. To bridge this gap, we have augmented the Byfl software with SDEs in order to exposing internal Byfl counters though the PAPI interface. This enables monitoring of both types of performance events—PAPI-related (hardware and software) events and Byfl-related (software) events—in a uniform way, through one consistent interface.

Table 5 lists the SDE-registered Byfl events and their descriptions as they are available through PAPI. While for the Byfl case, all counters are registered as "64-bit integer" and "delta" single-value SDEs; we have seen the benefits of different types, modes, and single- versus multi-value SDEs in the previous case studies.

The code snippet in Figure 6 serves as a simple example illustrating how the new SDE API (described earlier in this paper) can be used from within Byfl, or another library that wishes to register SDEs.

Analysis of SDE Performance Overhead

In this section, we provide an experimental evaluation of the performance overhead associated with SDE recorders. We focus on `papi_sde_record()` because it is currently the most expensive function of the newly added SDE API. In order to log values with an SDE recorder, memory is dynamically allocated via `memcpy()`, and when it runs out of space, the size of the memory object is changed via `realloc()`.

The benchmark code invokes the PAPI SDE function `papi_sde_create_recorder()` to first create a recorder and then uses the SDE API call `papi_sde_record()` to record 16,384 values of type `double` (64-bit fractional). Each of the PAPI SDE functions feature thread safety. Every time our benchmark called the function to log another value with the recorder, it also measured the time it took to execute this function by reading the CPU *time-stamp counter* using the x86 instruction `rdtsc`.

The experiments, mentioned in this section, were performed on three different architectures:

- Haswell E5-2650 v3 with a frequency of 2.30 GHz
- Westmere-EP E5606 with a frequency of 2.13 GHz
- Gainestown E5520 with a frequency of 2.27 GHz

The benchmarks were written in C (as is PAPI), and compiled with `gcc` 4.8.5 using optimization level "`-O3`".

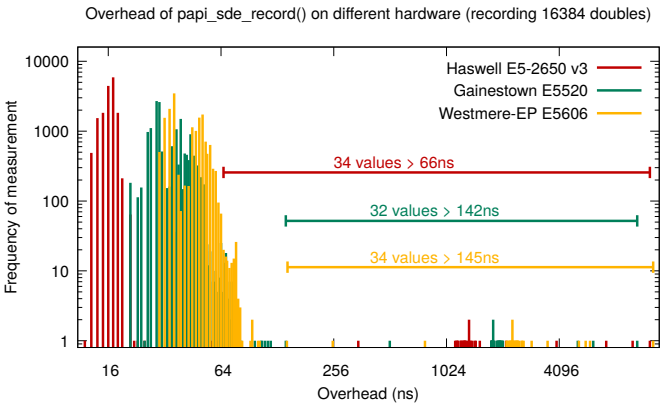


Figure 7. Performance Overhead of PAPI SDE Recorders.

SDE Name (prefixed with `sde:::BYFL::`)	SDE Description
`load_count_I`	Number of bytes loaded (I=integer)
`store_count_I`	Number of bytes stored (I=integer)
`load_ins_count_I`	Number of load instructions performed (I=integer)
`store_ins_count_I`	Number of store instructions performed (I=integer)
`call_ins_count_I`	Number of function-call instructions performed (I=integer)
`flop_count_I`	Number of FP operations performed (I=integer)
`fp_bits_count_I`	Number of bits used by all FP operations (I=integer)
`op_count _I`	Number of operations performed (I=integer)
`op_bits_count_I`	Number of bits used by all operations except loads/stores (I=integer)

Table 5. Registered SDEs in Byfl to enable users to gather internal Byfl counters though PAPI.

```
#define BYFL_MAX_COUNTERS 9

extern uint64_t  bf_load_count;
extern uint64_t  bf_store_count;
...

/* This function is called at BYFL init time */
void initialize_papi_sde(void)
{
    papi_sde_fptr_struct_t fptr_struct;

    POPULATE_SDE_FPTR_STRUCT(fptr_struct);
    papi_sde_hook_list_events(&fptr_struct);
}

/** This function registers and (optionally) describes
 *  events available from BYFL for listing in
 *  papi_native_avail.
 *  @param[in] papi_sde_fptr_struct_t fptr_struct
 *   fptr_struct->init -- function ptr to papi_sde_init
 *   fptr_struct->register_counter -- name of event
 *   fptr_struct->describe_counter -- event description
 **/
papi_handle_t papi_sde_hook_list_events(
    papi_sde_fptr_struct_t *fptr_struct)
{
    int i;
    void* sde_handle = nullptr;

    const char* byfl_counter_name[] = {
        "load_count_I",
        "byfl::store_count_I",
        ...
    };

    const char* byfl_counter_description[] = {
        "Total number of bytes loaded.",
        "Total number of bytes stored.",
        ...
    };

    uint64_t* byfl_counter_count[] = {
        &bf_load_count,
        &bf_store_count,
        ...
    };

    /* papi_sde_init() */
    sde_handle = fptr_struct->init("BYFL");

    /* papi_sde_register_counter() */
    for (i=0; i<BYFL_MAX_COUNTERS; i++) {
        fptr_struct->register_counter( sde_handle,
            byfl_counter_name[i],
            PAPI_SDE_RO|PAPI_SDE_DELTA,
            PAPI_SDE_long_long,
            byfl_counter_count[i] );
    }

    /* papi_sde_describe_counter() */
    for (i=0; i<BYFL_MAX_COUNTERS; i++) {
        fptr_struct->describe_counter( sde_handle,
            byfl_counter_name[i],
            byfl_counter_description[i] );
    }

    return sde_handle;
}
```

Figure 6. Code snippet of SDE API and its usage in Byfl.

The results of this overhead study are shown in Figure 7, which uses logarithmic axes. On the Haswell architecture, it takes approx. 16 nanoseconds to complete one recording via `papi_sde_record()`. The distribution displays that this is the case for most of the 16 thousand values that are recorded with this benchmark. On the two slower systems—Westmere-EP and Gainestown—the overhead per SDE recording measures between 30 and 40 nanoseconds.

Furthermore, the graphs for each of the three test systems display about 32 measurements with significantly higher timings (~2 microseconds) than the bulk of the distribution. These measurements (although they are rare enough not to affect the statistical properties of the distribution) are not due to noise, but rather they are an implementation artifact. Specifically, we have implemented recorders using contiguous memory that is allocated in 4KiB increments, to avoid having a large memory overhead in libraries where recording will not be heavily used. As a result of this policy, for every 512 values of type `double` that are recorded, there will be a call to `realloc()` in order to increase the available space. The parameter responsible for defining the size of the increment of memory allocation can be tuned to reduce the occurrence of reallocating (and copying) memory, and the overhead associated with it. However, as we demonstrate in this paper, the overhead is already insignificant, and thus, fine tuning the increment size should not be a concern for most users.

Related Work

The need for software developers to acquire knowledge of the internal behavior of libraries has been recognized by some of the communities that develop performance critical libraries. Particularly, the de facto standard for developing distributed-memory applications, MPI MPI Forum (2015), and one of the leading efforts for delivering multi-threaded shared memory applications, OpenMP OpenMP Architecture Review Board, provide both instrumentation and profiling mechanisms as part of their standard. The two distinct efforts, MPI_T Islam et al. (2016) and OMPT Eichenberger et al. (2013); OpenMP Tools Working Group respectively, make it clear once more that experts in performance critical libraries recognize the need for exporting internal library information to their users through instrumentation and profiling interfaces.

- MPI_T is an interface for tools introduced in the 3.0 version of MPI. It allows tools to understand and manipulate internal MPI variables in order to provide a more efficient and application-adapted execution environment. Similar to the PAPI interface, the MPI Tool Interface allows the implementation to specify internal control and performance variables, enabling tools to iterate over all possible variables to query their properties, retrieve descriptions about their meaning, and access and (if appropriate) alter their values.
- The OpenMP standard includes OMPT, a first-party interface for performance tools. It offers functions to query OpenMP states and callback functionality for relevant OpenMP events. This allows tools to explore details of an OpenMP implementation, examine runtime states associated with an OpenMP thread,

identify parallel regions and tasks, and to collect call stacks.

While these efforts provide a useful view of the execution of a parallel application, the granularity of the analysis interval is too coarse grain (mostly at the level of entry and exit point of MPI functions, OpenMP regions or tasks). More importantly, unlike the approach described in this paper, these solutions are specific to MPI and OpenMP, and so, they do not fit easily or naturally into the performance tool ecosystem. To incorporate them, developers of performance critical applications or higher-level profiling tools would have to implement profiling code customized for the communication layer of their parallel application. This paper addresses these challenges. The new SDE support in PAPI is not limited to a specific library, but enables *any* library developers to expose internal information about their libraries in a *consistent and standardized* way. Additionally, the PAPI SDE extension enables performance toolkits and application developers to capture and utilize such information across all the software layers used in an application.

TAU Shende and Malony (2006) is a profiling and tracing toolkit aimed at the performance evaluation of parallel programs, providing useful performance visualization analyses and displays. Like many performance analysis and auto-tuning tools, TAU relies on PAPI for retrieving performance counter measurements. TAU also offers the functionality to profile so-called *user-defined events*. The meaning of these events is entirely determined by the user. Unlike PAPI's SDE effort, however, TAU's user-defined events are limited to single-value events and are specific to TAU only.

Another related project is Caliper Lawrence Livermore National Laboratory, which offers a source-code annotation API for program instrumentation and performance measurement. Caliper is primarily a tool for performance experts to bake performance analysis capabilities directly into the applications they are trying to study. Among other performance values, such as timers, Caliper reads PAPI counters, so it can work synergistically with PAPI SDE by enabling performance experts to query library-specific SDEs through Caliper.

Conclusions

PAPI has provided a unification layer for hardware-based events, and enabled application developers and performance toolkits to access these events in a uniform and consistent way for more than 15 years. This paper presents our latest SDE developments that allow PAPI to perform the same role for software-based events. The addition of SDE in PAPI enables developers of libraries, application components, and runtime systems to expose internal, performance-critical information about their software in a consistent and standardized way.

The SDE integrations discussed in this paper highlight the importance of the different types of SDEs and their versatility for a wide variety of software layers such as the numerical linear algebra library MAGMA-Sparse, the NWChem quantum chemistry application, and the compiler-based performance analysis tool Byfl. The overhead analysis demonstrated that even for the most expensive SDE functionality (*Recorder*) the monitoring overhead is very low (tens of nanoseconds) under extreme use with benchmarks.

In summary, scientific application developers can monitor SDEs together with traditional hardware performance counter data to acquire a more complete picture of the entire application performance. Performance analysis tools that depend on PAPI for counter monitoring (e.g., Vampir, TAU, Score-P) will *automatically inherit* the PAPI SDE functionality. Using PAPI SDEs, both types of events can be monitored without the need for users to modify their applications or learn a new set of library and instrumentation primitives.

Acknowledgements

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Additionally, some of this material is based upon work supported in part by the National Science Foundation NSF under grant 1642440 "SI2-SSE: PAPI Unifying Layer for Software-Defined Events (PULSE)".

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Additionally, some of this material is based upon work supported in part by the National Science Foundation NSF under grant 1642440 "SI2-SSE: PAPI Unifying Layer for Software-Defined Events (PULSE)".

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

References

Anzt H, Gates M, Dongarra J, Kreutzer M, Wellein G and Köhler M (2017) Preconditioned Krylov solvers on GPUs. *Parallel Computing* 68: 32 – 44. DOI:https://doi.org/10.1016/j.parco.2017.05.006. Applications for the Heterogeneous Computing Era.

Davis TA and Hu Y (2011) The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38(1): 1:1–1:25. DOI: 10.1145/2049662.2049663. URL http://doi.acm.org/10.1145/2049662.2049663.

Eichenberger AE, Mellor-Crummey J, Schulz M, Wong M, Copty N, Dietrich R, Liu X, Loh E and Lorenz D (2013) OMPT: An OpenMP tools application programming interface for performance analysis. In: Rendell AP MM Chapman BM (ed.) *OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013*. Springer, Berlin, Heidelberg. Lecture Notes in Computer Science, vol 8122.

Islam T, Mohror K and Schulz M (2016) Exploring the MPI tool information interface: features and capabilities. *The International Journal of High Performance Computing Applications* 30(2): 212–222. DOI:10.1177/1094342015600507.

Kowalski K, Krishnamoorthy S, Olson RM, Tipparaju V and Aprà E (2011) Scalable implementations of accurate excited-state coupled cluster theories: Application of high-level methods to porphyrin-based systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. New York, NY, USA: ACM. ISBN 978-1-4503-0771-0, pp. 72:1–72:10. DOI: 10.1145/2063384.2063481. URL http://doi.acm.org/10.1145/2063384.2063481.

Lawrence Livermore National Laboratory (????) Caliper: Application Introspection System. https://computation.llnl.gov/projects/caliper.

MPI Forum (2015) MPI: A Message-Passing Interface Standard Version 3.1. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

OpenMP Architecture Review Board (????) OpenMP Application Program Interface, Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

OpenMP Tools Working Group (????) OpenMP Technical Report 2 on the OMPT Interface. http://openmp.org/mp-documents/ompt-tr2.pdf.

Pakin S and McCormick P (2014) Hardware-independent application characterization. In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*, volume 00. pp. 111–112. DOI:10.1109/IISWC.2013.6704676. URL doi.ieeecomputersociety.org/10.1109/IISWC.2013.6704676.

Shende SS and Malony AD (2006) The TAU Parallel Performance System. *The International Journal of High Performance Computing Applications* 20(2): 287–311. DOI:10.1177/1094342006064482.

Terpstra D, Jagode H, You H and Dongarra J (2009) Collecting Performance Data with PAPI-C. In: *Tools for High Performance Computing 2009, Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany*. pp. pp. 157–173.

The Clang Project (2018) Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

The LLVM Project (2018) The LLVM Compiler Infrastructure. https://llvm.org/.

Valiev M, Bylaska EJ, Govind N, Kowalski K, Straatsma TP, Van Dam HJJ, Wang D, Nieplocha J, Aprà E, Windus TL and de Jong W (2010) NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181(9): 1477–1489. DOI: 10.1016/j.cpc.2010.04.018.