INNOVATIVE
COMPUTING LABORATORY

SI2-SSE:

PAPI Unifying Layer for Software-Defined Events (PULSE)

Heike Jagode
Anthony Danalis
UNIVERSITY OF TENNESSEE

The Performance API (PAPI) provides tool designers and application engineers with a consistent interface and methodology for the use of low-level performance counter hardware found across the entire system (i.e., CPUs, GPUs, on/off-chip memory, interconnects, I/O system, energy/power, etc.). PAPI enables users to see, in near real time, the relationship between software performance and hardware events across the entire system.

Software-Defined Events in PAPI

GOAL

Offer support for software-defined events (SDE) to extend PAPI's role as a standardizing layer for perfor -mance counter monitoring.

VISION

Enable NSF software layers to expose SDEs that performance analysts can use to form a complete picture of the entire application performance.

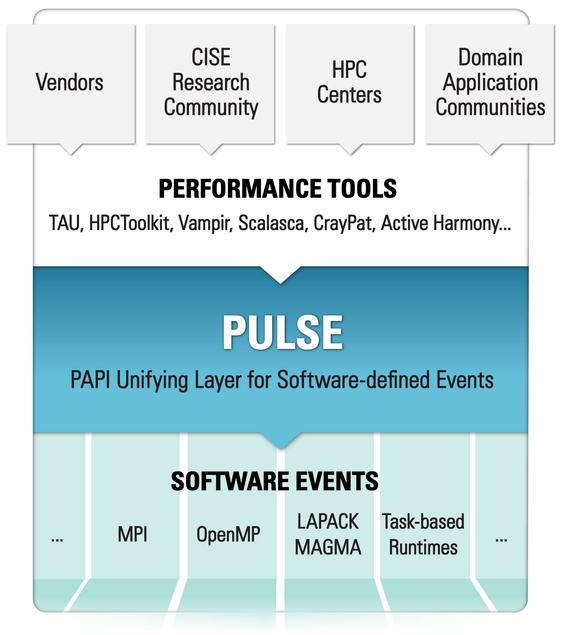
BENEFIT

Scientists will be better able to under -stand the interaction of the different applications layers, and interactions with external libraries and runtimes.

PULSE SCOPE

PULSE builds on the latest PAPI project and extends it with **software-defined events (SDE)** that originate from the HPC software stack and are currently treated as black boxes (i.e., com -munication libraries, math libraries, task-based runtime systems, applications).

The objective is to enable **monitoring of both types of performance events**---hardware- and software-related events---**in a uniform way**, through one consistent PAPI interface. Therefore, third-party tools and application developers have to handle only **a single hook to PAPI** to access all hardware performance counters in a system, including the new software-defined events.



PAPI's New SDE API

- API for reading SDEs remains the same as the API for reading hardware events, i.e., PAPI_start(), etc.
- SDE API calls are only meant to be used inside libraries to export SDEs from within those libraries.
- All API functions will be available in C and FORTRAN.

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

Initializes 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.

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

Must be called for every program variable/metric that the library wishes to register as an event.

`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.
`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.

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

Registers a function pointer to an accessor function provided by the library. Allows the user to export an event whose value does not map to the value of a single program variable/metric of the library.

`fp_counter` is a pointer to the accessor function with return type `void *` to support user-defined event types.
`param` is an opaque object that the library passes to PAPI, and PAPI passes it as a parameter to the accessor function.

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

Performance Counter Monitoring Capabilities

SUPPORTED ARCHITECTURES

AMD	arm Cortex A8, A9, A15, ARM64	CRAY THE SUPERCOMPUTER COMPANY Gemini and Aries interconnect, power
IBM Blue Gene Series, Q: 5-D Torus, I/O System, EMON power, energy	IBM Power Series	IBM Power9 NEST event support via Performance Co-Pilot (PCP) PAPI component
intel Westmore, Sandy/Ivy Bridge, Haswell, Broadwell, Skylake(-X), Kaby Lake	intel KNC, KNL, Knights Mill including power/energy	intel RAPL (power/energy), power capping
INFINIBAND TRADE ASSOCIATION	lustre	NVIDIA Tesla, Kepler: CUDA support for multiple GPUs; PC Sampling
NVIDIA NVML	KVM Virtual Environment	vmware Virtual Environment

PROJECTS AND THIRD-PARTY TOOLS APPLYING PAPI

PaRSEC UTK http://icl.utk.edu/parsec/	**Caliper** LLVM github.com/LLNL/caliper-compiler	**Kokkos** SNL https://github.com/kokkos
TAU University of Oregon http://tau.uoregon.edu/	**HPCToolkit** Rice University http://hpctoolkit.org	**Score-P** http://score-p.org
Vampir TU Dresden http://www.vampir.eu/	**Scalasca** FZ Juelich, TU Darmstadt http://scalasca.org/	**PerfSuite** NCSA http://perfsuite.ncsa.uiuc.edu/
Open|Speedshop Open|SpeedShop https://openspeedshop.org/	**SvPablo** RENCI at UNC www.renci.org/research/pablo	**ompP** LMU Munich http://www.ompp-tool.com/

CASE STUDY: **Integration of PAPI SDE in NWChem**

- As our application case study, we chose the NWChem (v. 6.8) iterative coupled cluster model with single and double excitations (CCSD) → best and most reliable method for accurate quantum-mechanical description of ground and excited states of chemical systems.
- For our first implementation, we registered four SDE counters in NWChem CCSD via our FORTRAN'08 interface for the function call **`papi_sde_register_counter()`**.

Table 1 provides a sample of PAPI-NWChem performance metrics that are exposed per CCSD sub-kernel

PAPI-NWChem Performance Counter	Counter Description	Type int, long long, float, double	Mode delta, instantaneous
`sde:::NWCHEM::t28_chain_cnt`	Total Number of chains with sequential DGEMMs in CCSD kernel t2_8()	64-bit integer values	delta
`sde:::NWCHEM::t28_max_chain_length`	Maximum number of sequential DGEMMs per chain in CCSD kernel t2_8()	64-bit integer values	instantaneous
`sde:::NWCHEM::t28_dgemm_cnt`	Total number of DGEMMs in CCSD kernel t2_8()	64-bit integer values	delta
`sde:::NWCHEM::t28_flop_cnt`	Total number of floating-point operations in CCSD kernel t2_8()	64-bit integer values	delta

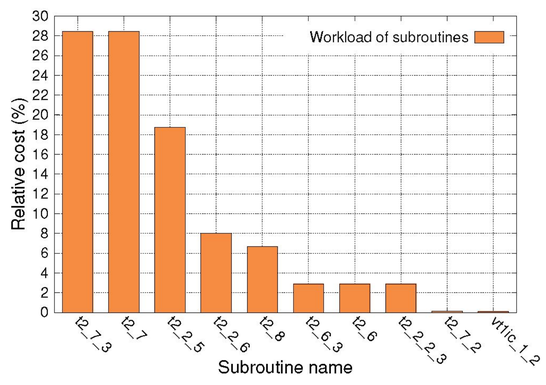


Figure 1 shows the relative workload of different subroutines (omitting those that fell under 0.1%). To calculate this load, we used the SDE counters that record the total number of floating-point operations for each CCSD kernel (e.g., sde:::NWCHEM::t28_flop_cnt counter for kernel t2_8).

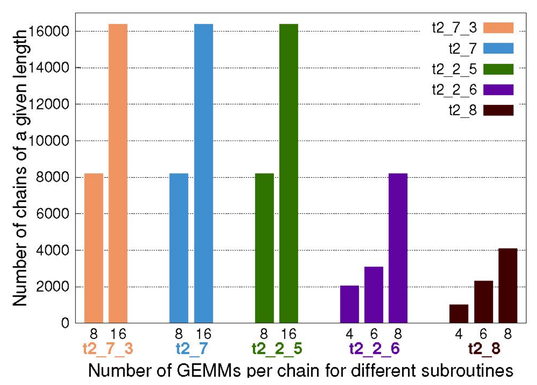


Figure 2 shows the distribution of chain lengths for the five subroutines with the highest workload. For this, we used the SDE counters that record the maximum number of sequential DGEMMs per chain for each CCSD kernel (e.g., sde:::NWCHEM::t28_max_chain_length counter for kernel t2_8).

CC subroutines	M	N	K	DGEMM FLOP count	Total # of DGEMMs
t2_7_3	1107	1107	1107	2,713,144,086	215,040
t2_7	1107	1107	1107	2,713,144,086	215,040
t2_2_5	1107	729	1107	1,786,704,642	215,040
t2_2_6	729	1107	1681	2,713,144,086	60,480
t2_8	729	1681	1681	4,119,959,538	33,264
t2_6_3	729	729	1681	1,786,704,642	33,264
t2_6	1681	729	729	1,786,704,642	33,264
t2_2_2_3	729	729	1681	1,786,704,642	33,264
t2_7_2	27	45,387	41	100,486,818	32,256
vt1ic_1_2	27	29,889	41	66,174,246	32,256

Table 2 lists additional statistics for the 10 most computationally expensive CCSD subroutines, such as the size and shape of individual DGEMM operations, the amount of floating-point operations per DGEMM, and the total number of such DGEMM operations.

INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE
NSF