PAPI's new Software-Defined Events for in-depth Performance Analysis

13th Parallel Tools Workshop

Anthony Danalis, Heike Jagode, Jack Dongarra

Dresden, Germany
Sep. 2-3, 2019
Motivating example: Fork-Join parallelism

- Fork-Join chain x 20 Tasks per fork.
- Memory bound kernel, with good cache locality.
Motivating example: Fork-Join parallelism (x20)

- 20 Independent Fork-Join chains x 20 Tasks per fork.
- Memory bound kernel, with good cache locality.
- 20 Cores on testing platform.
Typical task scheduling queue design

Core local queues

Shared Global queue (overflow)
Typical task scheduling queue design

Thread Local Queues => High Locality
Overflow & Work Stealing => Load Balance
Execution time vs Local Queue Length

[Graph showing the relationship between execution time and local queue length]
Execution time vs Local Queue Length

Tasks fit in local queues
Execution time vs Local Queue Length

Queues too small

Tasks fit in local queues
Execution time vs Local Queue Length (zoom)
Execution time vs Local Queue Length (zoom 2)
Execution time vs Local Queue Length (zoom 3)
Execution time vs Local Queue Length (zoom 4)
Execution time vs Local Queue Length (combined)
L2 Cache Misses (L3 show same pattern)
Failed Stealing Attempts

![Graph showing the relationship between local queue size and failed task-stealing attempts.](image)

- **Median Value**
- **Number of Failed Task-Stealing Attempts**
- **Local Queue Size**
Successful Close Stealing

![Graph showing the relationship between local queue size and tasks stolen from close local queues. The graph shows a peak at a local queue size of around 10, with the highest value of tasks stolen close to $2 \times 10^6$. The median value is indicated by a red line.]
Successful Close & Far Stealing

![Chart showing tasks stolen from close local queues vs. local queue size.](image-url)
Successful Shared Queue Stealing

![Graph showing the relationship between tasks stolen from a global shared queue and local queue size. The median value is indicated by a green line. The x-axis represents local queue size, ranging from 0 to 22, and the y-axis represents tasks stolen, ranging from 0 to 4.5x10^6. The graph demonstrates a decreasing trend as local queue size increases.]
Successful Local + Shared Queue Stealing
Unanswered questions

Q: So, what causes the bump?

Q: How did you measure all these things?
Unanswered questions

Q: So, what causes the bump?
A: I don’t know!

Q: How did you measure all these things?
Unanswered questions

Q: So, what causes the bump?
A: I don’t know!

Q: How did you measure all these things?
A: I am glad you asked...
Q: So, what causes the bump?
A: I don’t know!

Q: How did you measure all these things?
A: I am glad you asked... with PAPI SDEs of course!
Facts about PAPI Software Defined Events

- **New measurement possibilities:**
  Tasks stolen, matrix residuals, partial results reached, arguments passed to functions
Facts about PAPI Software Defined Events

• **New measurement possibilities:**
  Tasks stolen, matrix residuals, partial results reached, arguments passed to functions

• **Any tool can read PAPI SDEs:**
  SDEs from a library can be read with PAPI_start()/PAPI_stop()/PAPI_read().
Facts about PAPI Software Defined Events

• **New measurement possibilities:**
  Tasks stolen, matrix residuals, partial results reached, arguments passed to functions

• **Any tool can read PAPI SDEs:**
  SDEs from a library can be read with PAPI_start()/PAPI_stop()/PAPI_read().

• **Low overhead:**
  Performance critical codes can implement SDEs with zero overhead
Facts about PAPI Software Defined Events

• **New measurement possibilities:**
  Tasks stolen, matrix residuals, partial results reached, arguments passed to functions

• **Any tool can read PAPI SDEs:**
  SDEs from a library can be read with PAPI_start()/PAPI_stop()/PAPI_read().

• **Low overhead:**
  Performance critical codes can implement SDEs with zero overhead

• **Easy to use, with rich feature set:**
  Pull-mode & push-mode, read-write counters, sampling/overflowing, counters, groups, recordings, statistics, thread safety, custom callbacks
What was missing from existing infrastructure?

Events that occurred inside the software stack

There is no standardized way for a software layer to export information about its behavior such that other, independently developed, software layers can read it.

- HPC Application
- Math library
- Task runtime
- MPI
- Libibverbs
- Quantum Chemistry Method
- Distributed Factorization
- Data Dependency
- One Sided Communication
- RDMA completion
Stock HPCToolkit
Pull mode: Low overhead (down to zero)

Application

PAPI

X++;  
X+=7;  
Library w/ SDEs

register_counter(&x)
Pull mode: Low overhead (down to zero)

The application reads whenever it deems necessary.

```
X++;
X+=7;
```
Simplest SDE code (library side)

```c
static long long local_var;

void small_test_init( void ){
    local_var = 0;
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_register_counter( handle, "Evnt",
                                PAPI_SDE_RO|PAPI_SDE_DELTA,
                                PAPI_SDE_long_long,
                                &local_var );

    ...
}
```
SDE code for registering a callback function

```c
sometype_t *data;

void small_test_init( void ){
    data = ...
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_register_fp_counter(handle, "Evnt",
        PAPI_SDE_RO|PAPI_SDE_DELTA,
        PAPI_SDE_long_long,
        accessor, data);
...
}
```
Push mode: Determinism and Precision

PAPI\_overflow(callback)

create\_counter()
Push mode: Determinism and Precision

The library notifies the application when something happens.

```c
PAPI_overflow(callback)
```
SDE code for creating a counter (push mode)

```c
void *counter_handle;

void small_test_init( void ){
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_counter(handle, "Evnt",
                            PAPI_SDE_long_long,
                            &counter_handle);

    ...
}
```
void *recorder_handle;

void small_test_init( void ){
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_recorder(handle, "RCRDR",
          sizeof(double),
          cmpr_func_ptr,
          &recorder_handle);
    ...
}

SDE code for updating created counters/recorders

```c
void *counter_handle;
void *recorder_handle;

void push_test_dowork(void){
    double val;
    long long increment = 3;

    val = perform_useful_work();
    papi_sde_inc_counter(counter_handle, increment);
    papi_sde_record(recorder_handle, sizeof(val), &val);
}
```
Accessing a recorder: data pointer

```c
void *recorder_handle;

void small_test_init( void ){
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_recorder(handle, "RCRDR",
                              sizeof(double),
                              cmpr_func_ptr,
                              &recorder_handle);
    ...
}
```
Accessing a recorder: element count

```c
void *recorder_handle;

void small_test_init( void ) {
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_recorder(handle, "RCRDR",
                              sizeof(double),
                              cmpr_func_ptr,
                              &recorder_handle);

    ...}
```
Accessing a recorder: simple statistics

```c
void *recorder_handle;

void small_test_init( void )
{
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_recorder(handle, "RCRDR", sizeof(double),
                             cmpr_func_ptr, &recorder_handle);

    sde::TEST::RCRDR
    sde::TEST::RCRDR::CNT
    sde::TEST::RCRDR::MIN
    sde::TEST::RCRDR::Q1
    sde::TEST::RCRDR::MED
    sde::TEST::RCRDR::Q3
    sde::TEST::RCRDR::MAX
}
```
Accessing a recorder: simple statistics

void *recorder_handle;

void small_test_init( void )
{
    papi_handle_t *handle =
        papi_sde_init("TEST");
    papi_sde_create_recorder(handle, "RCRDR",
        sizeof(double),
        cmpr_func_ptr,
        &recorder_handle);

    sde::TEST::RCRDR
    sde::TEST::RCRDR::CNT
    sde::TEST::RCRDR::MIN
    sde::TEST::RCRDR::Q1
    sde::TEST::RCRDR::MED
    sde::TEST::RCRDR::Q3
    sde::TEST::RCRDR::MAX

    world
        It be great if PAPI
        had a C++ interface?

Linking applications without libpapi.so

PAPI SDE component comes with a weak symbols header

```c
create_counter()
return 0;

inc_counter()
return 0;
```
Performance overheads in simple benchmark

SDE overheads, Haswell E5-2650 v3

Execution time (ns)

Increment created counter  Record value
Performance overhead in PaRSEC
Performance overhead in HPCG
Performance overhead in HPCG (zoom)
Acquiring insight from SDEs

• Idle time (by measuring hardware counters like stall cycles)
  – Could be due to inevitable memory traffic.

• Idle time (by measuring SDEs from runtime)
  – Great opportunity for “investigative callback”
  – Callback must be runtime-aware

• Concurrent idle time
  – Unique insight about application design flaws
  – Additional context can lead to app. redesign
Acquiring insight from SDEs

• Idle time (by measuring hardware counters like stall cycles)
  - Could be due to inevitable memory traffic.

• Idle time (by measuring SDEs from runtime)
  - Great opportunity for “investigative callback”
  - Callback must be runtime-aware

• Concurrent idle time
  - Unique insight about application design flaws
  - Additional context can lead to app. redesign
Acquiring insight from SDEs

• Idle time (by measuring hardware counters like stall cycles)
  – Could be due to inevitable memory traffic.

• Idle time (by measuring SDEs from runtime)
  – Great opportunity for “investigative callback”
  – Callback must be runtime-aware

• Concurrent idle time
  – Unique insight about application design flaws
  – Additional context can lead to app. redesign
Open Problem for our Community:

How do we associate useful context information with SDEs?

What information to associate with CONCR_IDLE, or TASKS_STOLEN?

- Code-location
- Hardware events (e.g. cache misses)
- List of all threads’ activity
- Patterns in history (e.g. last task before stealing event)
- Patterns in call-path/stack/originating thread
Conclusions

- Libraries/runtimes generate multiple useful software “events”.
- PAPI SDE allows any software layer to export events.
- SDEs can be read using the standard PAPI functionality.
- SDEs have minimal to **zero** performance overhead.
- SDEs call for new types of analysis by tools.
- PAPI++ soon at a repo near you.