The Case for a Common Instrumentation Interface for HPC Codes

David Boehme, Kevin Huck,
Jonathan R. Madsen, Josef Weidendorfer

boehme3@llnl.gov, khuck@cs.uoregon.edu,
jrmadsen@lbl.gov, Josef.Weidendorfer@lrz.de

Lawrence Livermore National Laboratory

UNIVERSITY OF OREGON

BERKELEY LAB

lrz

Instrumentation Reality (everyone does it)

Runtime behavior of production runs is a black box without it

- Explicit, *targeted* instrumentation allows for
  - Low overhead, always-active notifications about runtime behavior
- Exposing internal timer data is helpful for
  - Light-weight profiling (report-on-exit)
  - Interactive bottleneck analysis (by users/sites)
  - Monitoring for regressions due to toolchain/source changes
  - Dynamic tuning in runtimes / system tools
- Many/most/all applications already have some instrumentation - but mostly one-off implementations
- A common, agreed-on interface could benefit everybody

Several potential examples

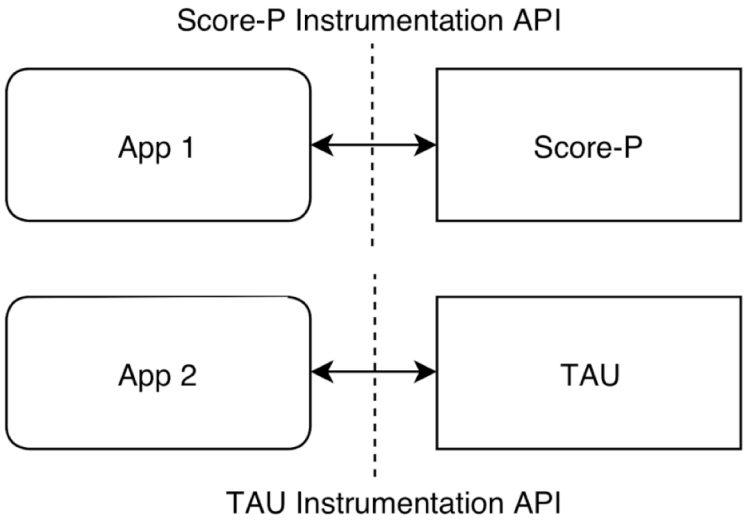
- Caliper
- TiMemory
- PerfStubs
- GOTCHA
- TAU
- Score-P
- -finstrument-functions
- Others...

Targeted instrumentation:

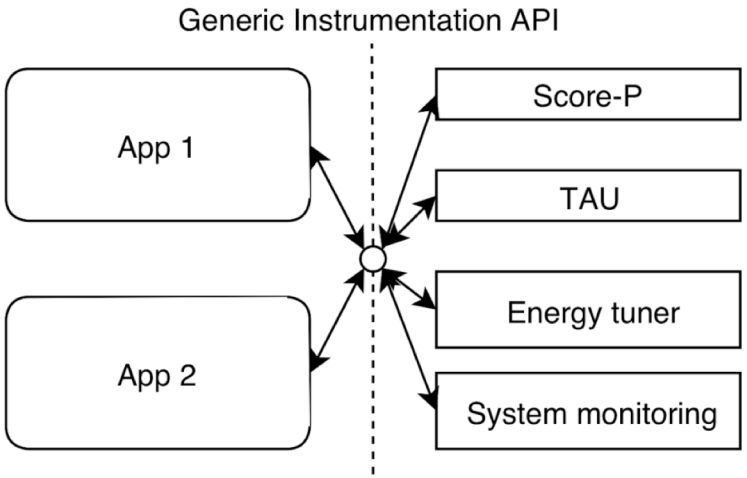
- Granularity control
- Explicit interpretation
- Reproducibility
- Robustness

But adding a tool to a build can be...frustrating

Always-On Instrumentation Interfaces



(a) only enabled during profiling experiments



(b) adaptive, always-on instrumentation interface

Caliper

- Performance Analysis Toolbox in a Library
  - Instrumentation API
  - Measurement config API
- Supports lightweight always-on profiling and detailed performance debugging
- Configuration at runtime via API or environment

https://github.com/LLNL/Caliper

```
int main(int argc, char* argv[])
{
    cali::ConfigManager mgr;
    mgr.add("runtime-report");
    mgr.start();

    CALI_MARK_FUNCTION_BEGIN;

    CALI_CXX_MARK_LOOP_BEGIN(mainloop, "main loop");
    for (int i = 0; i < count; ++i) {
        CALI_CXX_MARK_LOOP_ITERATION(mainloop, i);
        t += foo(i);
    }
    CALI_CXX_MARK_LOOP_END(mainloop);

    CALI_MARK_FUNCTION_END;
    mgr.flush();
}
```

TiMemory

- Arbitrarily-composable, type-safe C++ API
  - C and Python bindings
- Users can write their own fully integratable measurement or analysis tools
  - Can store data of any type and reprocess data to any other type
- Select tools at compile-time or run-time
- Single handler for multiple tools

```
auto_tuple<real_clock, cpu_roofline<double>>;
```

- No nesting restrictions
  - Start/stop in any order w/o artifacts
  - Geant4 attaches timers to particle objects
- Template interface simplifies GOTCHA generation to a single LOC
  - Extracts return type and args
  - Mangler for non-templated C++ functions
  - Inspect incoming arguments before function call and result before returning

```
TIMEMORY_C_GOTCHA(tool, idx, MPI_Allreduce);
```

https://github.com/NERSC/timemory

TiMemory

Sample Components

- wall_clock
- cpu_clock
- peak_rss
- gperf_cpu_profiler
- caliper
- nvtx_marker
- cupti_activity
- cupti_counters
- cpu_roofline<Types…>
  - e.g. cpu_roofline<double>
- gpu_roofline<Types…>
  - e.g. gpu_roofline<half2, float>

Sample Bundles

- type A = component_tuple<wall_clock, peak_rss>;
- type B = component_list<caliper, nvtx_marker>;
- type C = component_list<B, gpu_roofline<float>>;
- type D = auto_hybrid<A, C>;
- D obj;
  - obj.mark_begin(cudaStream_t);
  - obj.get<wall_clock>();
    - Get reference to individual tool
  - storage<wall_clock>::instance()->get();
    - Get entire call-stack

TiMemory instrumentation examples

```
// LIBRARY (C, C++)
#define INSTRUMENT_CREATE   uint64_t id;
#define INSTRUMENT_START    timemory_begin_record(__FUNCTION__, &id);
#define INSTRUMENT_STOP     timemory_end_record(id);
```

```
// AUTO_TIMER (C)
#define INSTRUMENT_START    void* timer = TIMEMORY_AUTO_TIMER("");
#define INSTRUMENT_STOP     FREE_TIMEMORY_AUTO_TIMER(timer);
```

```
// AUTO_TIMER (C++)
#define INSTRUMENT_START    TIMEMORY_AUTO_TIMER("");
```

```
# import
from timemory.util import auto_timer

# decorator for function
@auto_timer("")

# context-manager
with auto_timer("nested", mode="blank"):
```

PerfStubs

- Thin, stubbed-out, “adapter” interface for instrumentation
- The library *doesn’t do any measurement*
- Uses dlsym() or weak/strong symbol replacement to discover timer library implementation symbols
  - If found, sets function pointers to perform measurement
- Could be implemented as *observer* pattern but currently only allows 1 tool to implement the API
- Used with Alpine/Ascent, PETSc, PapyrusKV, proxy/demo applications, will use with ADIOS2
- C/C++ and Fortran macros

https://github.com/khuck/perfstubs

PerfStubs Example

```
1 #define PERFSTUBS_TIMER_START_FUNC(_timer) \
2     static void * _timer = NULL; \
3     if (_timer == NULL) \
4         _timer = psTimerCreate(psMakeTimerName(__FILE__, __func__, __LINE__)); \
5     psTimerStart(_timer);
6
7 #define PERFSTUBS_TIMER_STOP_FUNC(_timer) \
8     psTimerStop(_timer);
9
10 void myfunc(void) {
11     /* Will generate something like: "myfunc [{filename.c} {123,0}]" */
12     PERFSTUBS_START_FUNC(timer);
13     ...
14     PERFSTUBS_STOP_FUNC(timer);
15 }
```

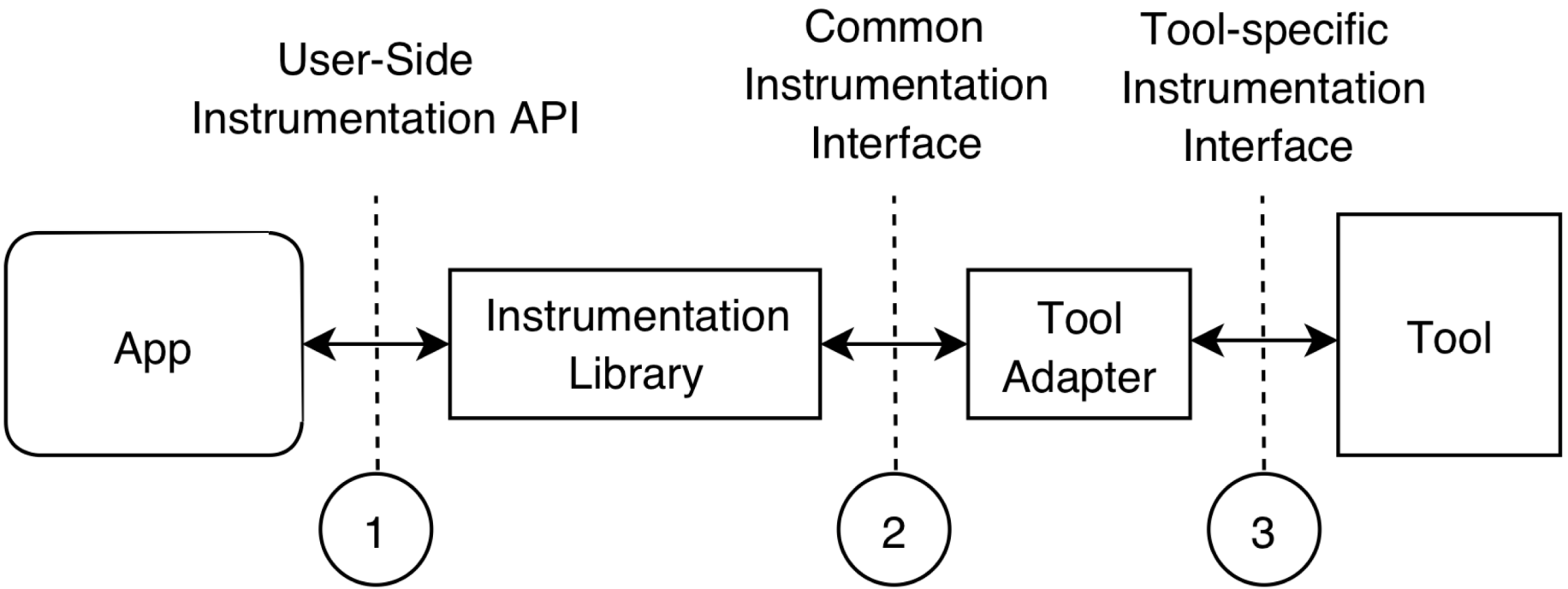
Potential use cases for common API:

- Performance regression testing / CI
- On-site application / library support
- Refactoring efforts
- 1st step in performance diagnosis
- Application/system monitoring
- Phase announcement (not detection) for energy saving strategies
- Feedback/control systems (assumes reconfigurable app)
- Startup tool discovery, instantiation
- Dynamic instantiation?

An Instrumentation Adapter Interface

- Should provide notification about
  - region enter/exit events
  - custom execution contexts / metrics (simulated years per day)
  - Metadata?
- Versioning / query for support of optional capabilities
  - enables extensibility / adaptation to custom use-cases
- Selective disabling of instrumentation points
  - enables methods for overhead control
- Multiplexing to multiple tools
- Back-channels from tool to application
  - enables feedback / setting application tuning parameters

An Instrumentation Adapter Interface



Instrumentation Evaluation Benchmark

- https://github.com/NERSC/instrumentation-benchmark
  - Suite of benchmark problems
    - Matrix multiplication in C and C++
    - Fibonacci calculation in C++
  - Generic specification of an instrumentation API
    - E.g. INSTRUMENT_CREATE(...)
    - Generates Python bindings for each benchmark problem per-API and makes them accessible as submodules
    - Python scripts to loop over all submodules, run benchmarks, generate statistics on overhead, and plot
  - Facilitate development of a front-end API
  - Facilitate research into optimal balance of flexibility and performance

Conclusion

- Need for common, high-level, runtime-available, context-aware application/library instrumentation
- Many applications/libraries already instrument
- Some implementations exist, can we figure out a high-level interface for pulling in useful tools?
- How much abstraction needed? 3 levels? 2?
- Project to evaluate solution(s) for quality, overhead
  - https://github.com/NERSC/instrumentation-benchmark
- Contact us!

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Berkeley National Laboratory under contract DE-AC02-05CH11231 and Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This research is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE-AC05-00OR00725, subcontract 4000159855, and used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231. LLNL-CONF-792721.