Caliper:

A Performance Analysis Toolbox in a Library

31st VI-HPS tuning workshop

David Boehme

LLNL-PRES-772412

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory
under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Caliper: Performance Analysis Toolbox in a Library

- Goal: ubiquitous performance data collection and analysis

https://github.com/LLNL/Caliper

- Embed & control performance analysis capabilities within the target program: vastly simplifies recurring performance measurements
- Also a building block for other tools: Provides application context and performance introspection

Caliper Features

- Source-code annotation API
  - C, C++, Fortran

- Performance measurement services
  - Profiling, tracing, call-stack unwinding, sampling, MPI, communication analysis, PAPI and libpfm hardware counters, memory analysis, CUDA

- Map annotations to third-party tools
  - TAU, NVidia Visual Profiler, ARM Forge MAP (coming soon)

- Flexible data aggregation and output
  - Human-readable hierarchical or table text formatters
  - Native .cali or JSON output for post-processing

Installing Caliper:
Spack or directly from https://github.com/LLNL/Caliper + CMake

```
$ spack info caliper
CMakePackage:   caliper

Description:
    Caliper is a program instrumentation and performance measurement
    framework. It provides data collection mechanisms and a source-code
    annotation API for a variety of performance engineering use cases, e.g.,
    performance profiling, tracing, monitoring, and auto-tuning.

Homepage: https://github.com/LLNL/Caliper

Tags:
    None

Preferred version:
    2.0.1     [git] https://github.com/LLNL/Caliper.git at tag v2.0.1
```

```
$ spack install caliper@master
```

Getting started:
Add instrumentation with source-code annotations

```
#include <caliper/cali.h>

int main()
{
  CALI_CXX_MARK_FUNCTION;

  CALI_MARK_BEGIN(“initialization”);
  // ...
  CALI_MARK_END(“initialization”);

  CALI_CXX_MARK_LOOP_BEGIN(loop, ”main loop”);
  for (int i = 0; i < 4; ++i) {
    CALI_CXX_MARK_LOOP_ITERATION(loop, i);

    foo();
  }
  CALI_CXX_MARK_LOOP_END(loop);
}
```

Getting started:
Configure measurements via config file or env variables …

```
$ CALI_CONFIG_PROFILE=runtime-report ./app
```

```
Path          Inclusive time (usec) Exclusive time (usec) Time %
main                      38.000000             20.000000   52.6
  main loop                8.000000              8.000000   21.1
  initialization          10.000000             10.000000   26.3
```

Getting started:
… OR enable measurements through the control channel API

```
#include <caliper/cali.h>

int main(int argc, char* argv[])
{
  if (argc > 1 && std::string(argv[1] == “-P”))
    cali::create_channel(“profile”, 0, {
        { “CALI_CONFIG_PROFILE”, “runtime-report” }
      });
}
```

```
$ ./app -P
```

```
Path         Inclusive time (usec) Exclusive time (usec) Time %
main                     38.000000             20.000000   52.6
  main loop               8.000000              8.000000   21.1
  initialization         10.000000             10.000000   26.3
```

Contact & Links

- Github repository:

  https://github.com/LLNL/Caliper

- Documentation:

  https://llnl.github.io/Caliper

- Examples & tutorial:

  https://github.com/LLNL/caliper-examples

- Contact:

David Boehme
boehme3@llnl.gov

Part II:
Build and Link

Build requirements

- CMake 3.1+
- C++11 compiler
- Python
- POSIX threads
- Optional
  - CUpti
  - Doxygen
  - Dyninst (for symbol lookup)
  - GOTCHA
  - LibPFM
  - Libunwind
  - MPI
  - OMPT
  - PAPI
  - Sphinx (documentation generation)

CMake configuration

- Build *everything:*

```
$ cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_INSTALL_PREFIX=<install dir> \
    -DCMAKE_C_COMPILER=<C compiler> -DCMAKE_CXX_COMPILER=<C++ compiler> \
    -DWITH_MPI=On -DMPI_C_COMPILER=<mpi c compiler> -DMPI_CXX_COMPILER=<mpi c++ compiler> \
    -DWITH_CALLPATH=On \
    -DWITH_SAMPLER=On \
    -DWITH_GOTCHA=On \
    -DWITH_PAPI=On -DPAPI_PREFIX=<papi install dir> \
    -DWITH_LIBPFM=On \
    -DWITH_DYNINST=On -DDyninst_DIR=<path to Dyninst-config.cmake> \
    -DWITH_CUPTI=On -DCUDA_TOOLKIT_ROOT_DIR=<cuda dir> -DCUPTI_PREFIX=<path to cupti> \
    -DWITH_NVPROF=On \
    -DWITH_VTUNE=On -DITT_PREFIX=<path to vtune>
```

Linking the Caliper library

- Link libcaliper.so

```
$ g++ -o app $(OBJECTS) –L$(CALIPER_DIR)/lib64 -lcaliper
```

- For MPI programs: Link libcaliper-mpi.so as well

```
$ mpicxx -o app $(OBJECTS) –L$(CALIPER_DIR)/lib64 –lcaliper-mpi -lcaliper
```

- CMake support

```
find_package(caliper)

target_include_directories(app ${caliper_INCLUDE_DIR})
target_link_libraries(app PRIVATE caliper caliper-mpi)
```

```
$ cmake –Dcaliper_DIR=<caliper installation dir>/share/cmake/caliper
```

Part III:
Annotation API

Caliper Source-code Annotation APIs

- High-level macros:
  - Easy method for common cases (functions, code regions, loops, loop iterations, statements)
  - Use pre-defined attribute labels

- C++ annotation class
  - Allows custom annotations with user-defined attribute labels

- C and Fortran annotation API
  - Custom annotations in C and Fortran

- Caliper runtime library class
  - Low-level, more involved, interface subject to change

Macro API: Functions

In C++:

```
#include <caliper/cali.h>

void foo() {
  CALI_CXX_MARK_FUNCTION;
}

void bar(int i) {
  CALI_CXX_MARK_FUNCTION;
  if (i < 0)
    return;
}
```

In C:

```
#include <caliper/cali.h>

void foo() {
  CALI_MARK_FUNCTION_BEGIN;
  /**/
  CALI_MARK_FUNCTION_END;
}

void bar(int i) {
  CALI_MARK_FUNCTION_BEGIN;
  if (i < 0) {
    CALI_MARK_FUNCTION_END; /* mark all exits! */
    return;
  }
  CALI_MARK_FUNCTION_END;
}
```

- Exports function name as `function=<function name>`
  - Determines function name from __FUNC__ compiler macro
- Must mark ***all*** function exits in C (and Fortran)

Macro API: Code Regions

```
#include <caliper/cali.h>

void main() {
  CALI_MARK_BEGIN(”init”);

  do_init();

  CALI_MARK_END(”init”);
}
```

- Mark arbitrary code regions
- Exports code region as `annotation=<user-defined name>`

Macro API: Loops

In C++:

```
CALI_CXX_MARK_LOOP_BEGIN(mainloop_id, "mainloop");

for (int i = 0; i < ITER; ++i) {
  CALI_CXX_MARK_LOOP_ITERATION(mainloop_id, i);

  if (test(i) == 0)
    continue;
}

CALI_CXX_MARK_LOOP_END(mainloop_id);
```

In C:

```
CALI_MARK_LOOP_BEGIN(mainloop_id, "mainloop");

for (int i = 0; i < ITER; ++i) {
  CALI_MARK_ITERATION_BEGIN(mainloop_id, i);

  if (test(i) == 0) {
    CALI_MARK_ITERATION_END(mainloop_id);
    continue;
  }

  CALI_MARK_ITERATION_END(mainloop_id);
}

CALI_CXX_MARK_LOOP_END(mainloop_id);
```

- Loop annotation exports `loop=<user-defined name>`
- Iteration annotation exports `iteration#<loop name>=<iteration number>`

Custom Annotations

- Annotation APIs allow creation of user-defined *attributes*

- Attribute labels have:
  - Name
  - Data type
  - Scope (thread or process)
  - (Optional) flags

- General pattern:
  - `begin(attribute, value) – end(attribute)` for regions
  - `set(attribute, value)` for individual variables

C++ Annotation API

```
#include <caliper/cali.h>

int main() {
  cali::Annotation(”my.param”).set(42.42);
  cali::Annotation
    phase_ann(”my.phase”, CALI_ATTR_SCOPE_PROCESS | CALI_ATTR_NESTED);

  phase_ann.begin(”outer”);
  {
    cali::Annotation::Guard
      g(phase_ann.begin(”inner”));
    // ...
  }
  phase_ann.end();
}
```

- cali::Annotation constructor takes attribute name and flags, begin/set methods take values. Data type is automatically derived from begin/set overloads.

- Guard helper class automatically ”ends” annotations

C Annotation API

```
cali_id_t phase_attr =
  cali_create_attribute(”my.phase”, CALI_TYPE_STRING, CALI_ATTR_SCOPE_PROCESS);

cali_begin_string(phase_attr, ”outer”);

cali_set_double_byname(”my.param”, 42.42);

{
  cali_begin_string(phase_attr, ”inner”);
  cali_end(phase_attr);
}

cali_end_byname(”my.phase”);
```

- API function pattern: begin|set_<datatype>[_byname]
  - ”Overloads” for different data types

- ”_byname” variants refer to attributes directly by name, otherwise attribute ID from cali_create_attribute() required

Metadata Annotations

```
#include <caliper/cali.h>

void main(int argc, char* argv[]) {
  int config1 = atoi(argv[1]);
  const char* config2 = argv[2];

  cali_set_global_int_byname(“config1”, config1);
  cali_set_global_string_byname(“config2”, config2);
}
```

- Collect metadata that describes the run (config arguments etc.)
- cali_set_global_<int|double|uint|string>_byname(*key*, *val*)

Annotations maintain nesting information

```
#include <caliper/cali.h>

int main()
{
  CALI_CXX_MARK_FUNCTION;

  CALI_MARK_BEGIN(“initialization”);
  CALI_MARK_END(“initialization”);

  CALI_CXX_MARK_LOOP_BEGIN(loop, ”main loop”);
  for (int i = 0; i < 4; ++i) {
    cali::Annotation::Guard
      g(cali::Annotation(“inner region”, CALI_ATTR_NESTED).begin(“compute”));
  }
  CALI_CXX_MARK_LOOP_END(loop);
}
```

```
Path                             Time
main                               30
  initialization                    8
  loop                             15
    compute                       100
```

Part IV: Runtime Configuration and Control API

Configuration Basics

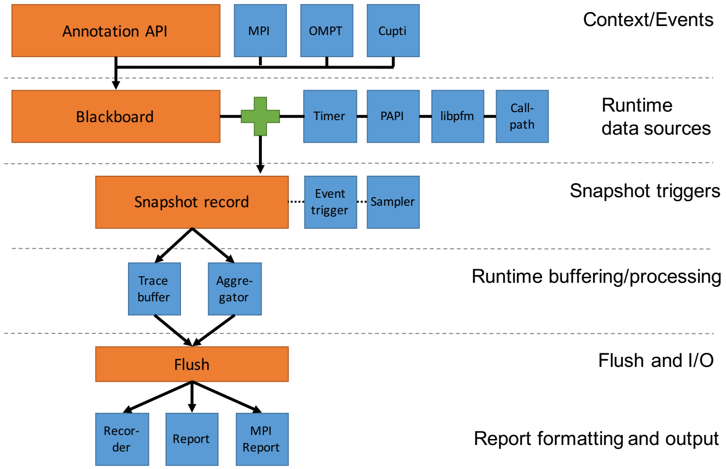
- Configuration via environment variables, configuration file, or channel API
  - Read caliper.config file in current working directory or from value of CALI_CONFIG_FILE

```
# [sample-profile]
CALI_SERVICES_ENABLE=aggregate,sampler,recorder,symbollookup
CALI_AGGREGATE_KEY=cali.sampler.pc
CALI_SAMPLER_FREQUENCY=100

# [event-trace]
CALI_SERVICES_ENABLE=event,recorder,timestamp,trace
CALI_RECORDER_FILENAME=trace.cali
```

```
$ CALI_CONFIG_PROFILE=event-trace ./app
```

Selecting Services



Annotation API
MPI
OMPT
Cupti
Context/Events

Blackboard
Timer
PAPI
libpfm
Call-path
Runtime data sources

Snapshot record
Event trigger
Sampler
Snapshot triggers

Trace buffer
Aggre-gator
Runtime buffering/processing

Flush
Flush and I/O

Recor-der
Report
MPI Report
Report formatting and output

Caliper services (blue)

```
$ CALI_SERVICES_ENABLE=aggregate,event,report,timestamp
```

Select services via CALI_SERVICES_ENABLE

- Generating output through Caliper requires:
  - A snapshot trigger service (e.g., event)
  - A snapshot processing service (e.g., trace)
  - An output service (e.g., recorder)
- Other services provide measurement or context data
- Services can be enabled in any combination

MPI Programs

- libcaliper-mpi.so provides MPI-specific functionality

- *recorder* and *report* services write output on each process

```
# [process-trace]
CALI_SERVICES_ENABLE=event,mpi,recorder,timestamp,trace
CALI_RECORDER_FILENAME=“trace-%mpi.rank%.cali”
```

- *mpireport* service aggregates data across processes & writes output on one rank

```
# [aggregate-report]
CALI_SERVICES_ENABLE=aggregate,event,mpi,mpireport,timestamp
CALI_MPIREPORT_CONFIG=“select *,avg(sum#time.duration),max(sum#time.duration) group by prop:nested
  format tree”
CALI_MPIREPORT_FILENAME=“report.cali”
```

Channel API

- The Channel API manages a Caliper configuration and data processing pipeline
- Multiple channels can be active simultaneously

```
cali_id_t channel = cali::create_channel(“profile”, CALI_CHANNEL_LEAVE_INACTIVE, {
    { “CALI_CONFIG_PROFILE”, “runtime-report” },
  } );

cali_activate_channel(channel);
// ...
cali_deactivate_channel(channel);
```

Using Channels: Reports and flush

- Flush data and process queries

```
cali_id_t channel = cali::create_channel(“trace”, 0, {
    { “CALI_SERVICES_ENABLE”, “event,trace,timestamp” },
  });

cali_channel_flush(channel, 0);

cali::write_report_for_query(channel,
  “select *,sum(time.durarion) group by function format tree”,
  CALI_FLUSH_CLEAR_BUFFERS, std::cerr);
```

- Snapshots (Tool API)

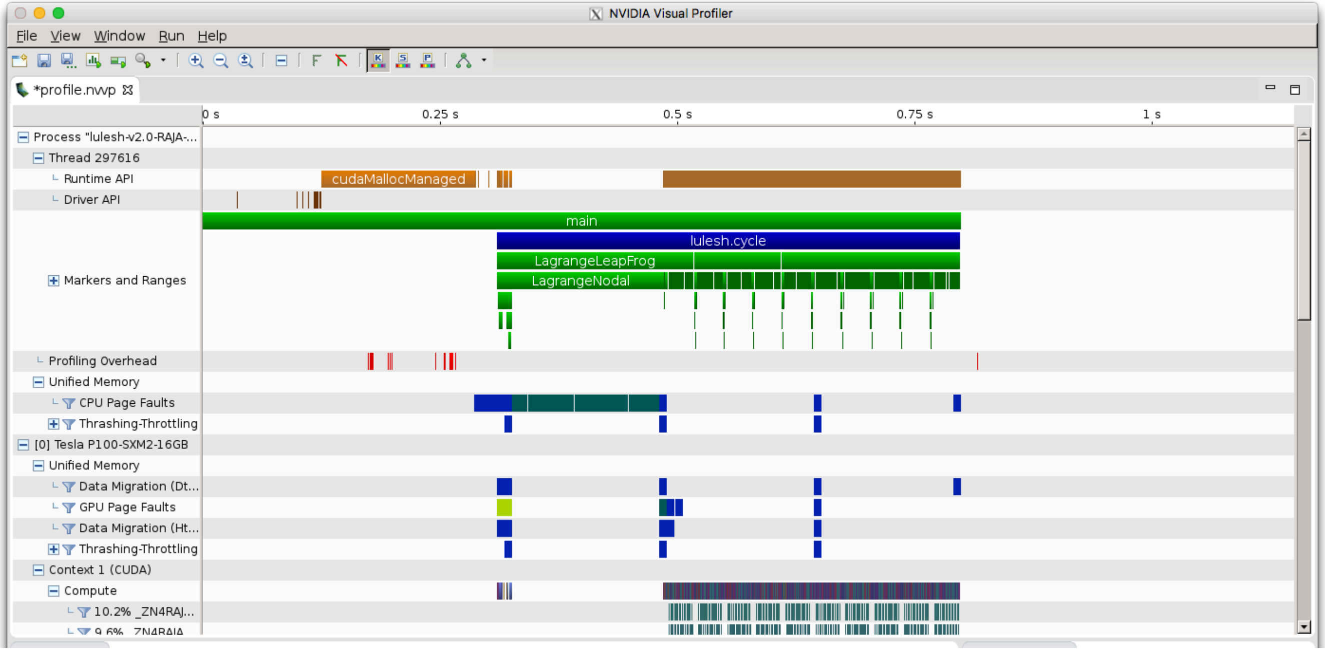
```
unsigned char buffer[80];
scopes = CALI_SCOPE_THREAD | CALI_SCOPE_PROCESS;

size_t bytes_read, len =
  cali_channel_pull_snapshot(channel, scopes, 80, buffer);

cali_unpack_snapshot(buffer, &bytes_read, proc_fn, NULL);
```

Connect to third-party tools:
Map Caliper annotations to other tool APIs, e.g. NVidia nvtx

```
$ CALI_SERVICES_ENABLE=nvprof nvprof <nvprof-opts> ./app
```



Part V:
Data Processing

Data Processing Basics

- Caliper outputs native .cali or processed data
- Use *cali-query* command-line tool to processes .cali files
- Use CalQL queries to configure data processor
  - Select, aggregate, filter, and format data

```
SELECT function,sum(time.duration) WHERE iteration=5 GROUP BY function FORMAT table
```

Caliper Query Language (CalQL) Clauses

```
SELECT <list>                # Select attributes and define aggregations (i.e., select columns)
  *                          # select all attributes
  <attribute>                # select <attribute>
  count()                    # number of input records in grouping
  sum(<attribute>)           # compute sum of <attribute> for grouping
  min(<attribute>)           # compute min of <attribute> for grouping
  max(<attribute>)           # compute max of <attribute> for grouping
  avg(<attribute>)           # compute average of <attribute> for grouping
  percent_total(<attribute>) # compute percent of total sum for <attribute> in grouping
  ... AS <name>              # use <name> as column header in tree or table formatter

GROUP BY <list>              # Define aggregation grouping
                             #   (what to aggregate over, e.g. “GROUP BY function,mpi.rank”)
  <attribute>                # include <attribute> in grouping
  prop:nested                # include all attributes with NESTED flag in grouping

WHERE <list>                 # define filter (i.e., select records/rows)
  <attribute>                # records where <attribute> exists
  <attribute>=<value>        # records where <attribute>=<value>
```

Caliper Query Language (CalQL) Clauses, Continued

```
FORMAT <formatter>           # Define output format
  cali                       # .cali format
  expand                     # “<attribute1>=<value1>,<attibute2>=<value2>,...”
  json                       # json records { “attribute1”: “value1”, “attribute2”: “value2” }
  json-split                 # json format w/ separate node hierarchy for hatchet library
  table                      # human-readable text table
  tree                       # human-readable text tree output

ORDER BY <list>              # Sort records in output (table formatter only)
  <attribute>                # order by <attribute>
  ... ASC                    # sort in ascending order
  ... DESC                   # sort in descending order
```

Part VI:
Lulesh Hands-on Example

Lulesh tutorial setup on stampede2

- Lulesh tutorial directory:

```
$ $ ls ~tg857082/tutorial/lulesh
caliper.config  lulesh-build-metadata.cc.in  lulesh-comm.cc  lulesh-init.cc   lulesh-util.cc  README
CMakeLists.txt  lulesh.cc                    lulesh.h        lulesh_tuple.h   lulesh-viz.cc   TODO
```

- Setup:

```
$ source ~tg857082/tutorial/setup.sh
```

- Build instrumented Lulesh:

```
$ ~tg857082/tutorial/build-lulesh.sh
```

What was added in Lulesh?
Function and main loop instrumentation

```
static inline
void TimeIncrement(Domain& domain)
{
   CALI_CXX_MARK_FUNCTION;
   // ...
}
```

```
   CALI_CXX_MARK_LOOP_BEGIN(mainloop, "lulesh.cycle");

   while ((locDom->time() < locDom->stoptime()) && (locDom->cycle() < opts.its)) {
      CALI_CXX_MARK_LOOP_ITERATION(mainloop, static_cast<int>(locDom->cycle()));

      TimeIncrement(*locDom) ;
      LagrangeLeapFrog(*locDom) ;
   }

   CALI_CXX_MARK_LOOP_END(mainloop);
```

What was added in Lulesh?
Lulesh “region” instrumentation

```
cali::Annotation r_ann("lulesh.region", CALI_ATTR_SCOPE_PROCESS);

for (Int_t r=0 ; r<domain.numReg() ; r++) {
   cali::Annotation::Guard
      g(r_ann.begin(static_cast<int>(r)));

   // ...
}
```

What was added in Lulesh?
Caliper setup and profiling measurement channel (“-P”)

```
cali_config_preset("CALI_LOG_VERBOSITY", "0");
cali_config_preset("CALI_CALIPER_ATTRIBUTE_DEFAULT_SCOPE", "process");

cali_mpi_init();
```

```
if (opts.profile) {
   if (numRanks > 1) {
      cali::create_channel("mpi-runtime-report", 0, {
             { "CALI_CONFIG_PROFILE", "mpi-runtime-report" }
         });
   } else {
      cali::create_channel("runtime-report", 0, {
             { "CALI_CONFIG_PROFILE", "runtime-report" }
         });
   }
}
```

What was added in Lulesh?
Metadata collection

```
void RecordCaliperMetadata(const struct cmdLineOpts& opts)
{
   cali_set_global_int_byname("Iteration",      opts.its);
   cali_set_global_int_byname("Problem size",   opts.nx);
   cali_set_global_int_byname("Number of regions", opts.numReg);
   cali_set_global_int_byname("Region balance", opts.balance);
   cali_set_global_int_byname("Region cost",    opts.cost);

   cali_set_global_uint_byname("Start time", static_cast<uint64_t>(time(NULL)));

   // add build metadata (generated by CMake)
   for (size_t i = 0; buildMetadata[i][0]; ++i)
      cali_set_global_string_byname(buildMetadata[i][0], buildMetadata[i][1]);
}
```

A caliper.config file

```
# [mpiP]
...
# [loop-profile]
...
# [region-profile]
...
# [json-region-profile]
...
# [mpi-comm-size]
...
# [event-trace]
...
# [callpath-sample-report]
...
# [mpi-msg-trace]
```

Tutorial setup on stampede2

- Setup:

```
$ source ~tg857082/tutorial/setup.sh
```

- Build instrumented Lulesh:

```
$ ~tg857082/tutorial/build-lulesh.sh
```

- (Optional: Start an interactive MPI session with 8 ranks):

```
$ srun -N 1 --exclusive -t 0:30:00 -n 8 --reservation VI-HPS_SKX_DAY5 -p skx-normal --pty bash
```

Lawrence Livermore
National Laboratory