Ubiquitous Performance Analysis

David Boehme([✉]), Pascal Aschwanden, Olga Pearce, Kenneth Weiss, and Matthew LeGendre

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
boehme3@llnl.gov

Abstract. In an effort to guide optimizations and detect performance regressions, developers of large HPC codes must regularly collect and analyze application performance profiles across different hardware platforms and in a variety of program configurations. However, traditional performance profiling tools mostly focus on ad-hoc analysis of individual program runs. *Ubiquitous performance analysis* is a new approach to automate and simplify the collection, management, and analysis of large numbers of application performance profiles. In this regime, performance profiling of large HPC codes transitions from a sporadic process that often requires the help of experts into a routine activity in which the entire development team can participate. We discuss the design and implementation of an open source ubiquitous performance analysis software stack with three major components: the Caliper instrumentation library with a new API to control performance profiling programmatically; Adiak, a library for automatic program metadata capture; and SPOT, a web-based visualization interface for comparing large sets of runs. A case study shows how ubiquitous performance analysis has helped the developers of the Marbl simulation code for over a year with analyzing performance and understanding regressions.

Keywords: Performance · Measurement · Instrumentation · Caliper

1 Introduction

Lawrence Livermore National Laboratory hosts several application teams who develop and maintain large multi-physics simulation codes. These production codes are under continuous development, run in a wide variety of configurations, and on complex, heterogeneous HPC systems where frequent hardware and software updates create a constantly evolving execution environment. To guide optimizations and detect unexpected performance problems, developers must proactively monitor the performance of their codes throughout the application lifecycle, both during development and in production. To support this need, we have developed and deployed software infrastructure to simplify and automate application-level performance data collection, storage, and analysis.

Traditional HPC performance profiling tools typically focus on analyzing individual program runs. They employ powerful mechanisms to collect detailed data for finding performance bottlenecks, but are often difficult to automate or too intrusive to be used in production runs. With *ubiquitous* performance analysis, we instead aim to collect application performance data whenever possible, and provide a central interface for developers to analyze the collected data. We address several challenges to accomplish this. First, we want to avoid complex measurement setup or postprocessing steps: performance profiling should be available for any user, at any time, and for any run. We therefore integrate a performance profiling library into applications and control measurements programmatically, for example through a command-line option. As we collect data from many runs, the performance analysis focus shifts from analyzing individual program runs to comparing data across runs or across HPC platforms. To facilitate this, we have developed a web interface with novel analysis and visualization tools for analyzing large collections of runs. Finally, to effectively work with such collections, we need descriptive metadata about the program and system configuration for each run. We collect this data automatically with code annotations using a new metadata collection library.

To adopt ubiquitous performance analysis, application developers augment their codes with instrumentation markers, metadata annotations, and initialization code to configure and activate performance profiling. With performance measurement capabilities built into applications, it is easy to enable profiling in production runs or in automated workflows like nightly Continuous Integration (CI) tests. It also simplifies performance profiling for application end users, who may not be familiar with traditional developer-oriented HPC profiling tools. Performance analysts can thus observe real program usage in practice and identify problems due to misconfiguration. Central data storage and access through our analysis web frontend simplifies sharing of performance data across a development team and with other stakeholders. Developers are no longer limited to infrequent ad-hoc profiling of individual runs, but can analyze a complete record of program performance covering many different program configurations over the entire lifespan of the code. Stated simply, ubiquitous performance analysis represents a shift in how we view performance tracking within long-lived HPC codes. It transitions performance analysis from a process that the team performs sporadically, often only with the help of external experts, to a routine activity in which the entire development team can easily, or even unknowingly, participate.

Contributions. Our ubiquitous performance analysis system builds upon the Caliper instrumentation and profiling library [11], whose low runtime overhead affords it to be compiled into HPC applications. In this paper, we introduce additional frontend and backend components to implement a full ubiquitous performance analysis software stack:

– ConfigManager, a profiling control API in Caliper to let applications control performance measurements programmatically;
– Adiak, a library for collecting user-defined metadata; and
– SPOT, a web-based data analysis and visualization interface with novel visualizations to explore large collections of runs.

More importantly, we explain the motivation, key concepts, and design behind the ubiquitous performance analysis approach, and discuss our experiences implementing ubiquitous performance analysis in the LULESH proxy app and the Marbl production code.

2 State of the Art

In this section, we compare our approach to the current state-of-the-art in tools and methodologies for HPC performance analysis.

There is a wide range of community-driven and commercial HPC performance analysis tools covering different measurement methodologies, systems, and use cases. Frameworks like HPCToolkit [7], Score-P [22], TAU [30], and OpenSpeedShop [32] collect detailed per-thread execution profiles or traces for in-depth analyses, such as automatic bottleneck detection [14] or profile analysis [15,24]. Vampir [12] and Paraver [27] visualize large-scale parallel execution and communication traces. Many tools support the collection of CPU, GPU, and on-core hardware counters via PAPI [26] or similar APIs, as well as analysis of communication, multithreading, and GPU usage through the MPI profiling interface, the OpenMP tools interface [13], and NVIDIA's CUPTI API [4]. Generally, these tools are best characterized as *performance debugging tools*, designed around interactive measure-analyze-refine debugging workflows and focused on finding root causes of performance bottlenecks for individual program runs. Measurement setup can be complex. Instrumentation-based tools like Score-P and TAU require the target code to be re-compiled with instrumentation turned on, while sampling-based tools like HPCToolkit require postprocessing steps to map binary addresses to symbol names. The tools use custom profile and trace data formats, and require tool-specific graphical applications for data analysis. Due to the complex measurement setup, usage of performance debugging tools within regular application development and production workflows is often limited, and relegated to expert users with specific performance debugging needs.

Many HPC codes employ some form of built-in *lightweight always-on profiling* to keep track of time spent in major application subsystems or kernels for monitoring and benchmark purposes. Some codes use libraries like GPTL [29], Caliper [11], or TiMemory [23] for this purpose, while others include custom time measurement solutions, typically using small marker functions or macros placed around code regions of interest. Our system can replace custom lightweight timing solutions, and offers rich measurement capabilities that can be activated by the application without complex setup steps.

Performance data management tools such as PerfDMF [16] and PerfTrack [18,19,21] provide the ability to analyze and compare performance data collected from different runs of an application. PerfDMF provides robust, interoperable components for performance data management. PerfDMF is the SQL-based storage backend for PerfExplorer2 [17], a data mining framework with capabilities to correlate performance data and metadata, allowing many types of analyses to compare performance data from multiple experiments (e.g., scaling studies). The PerfTrack performance experiment management tool also uses a SQL database to store profile data from multiple experiments. It includes interfaces to the data store, a GUI for interactive analysis, and modules to automatically collect experiment metadata. The IPM [31] performance monitoring framework gathers MPI function profiles together with environment and application information for cross-run performance comparisons such as scaling studies. Ubiquitous performance analysis builds upon many of the elements developed in these performance data managers. We provide an analysis and visualization web frontend to access data without specialized GUI tools, and a library for collecting user-defined program metadata automatically.

Some *commercial cloud and data center* operators have developed in-house automatic performance analysis solutions for large-scale distributed applications. Among the ones that are known are Alibaba's P-Tracer [25] and Google's Google-Wide Profiling [28] (GWP). P-Tracer samples call-stack traces from applications, while GWP continuously records performance data, including application-level call-stack profiles, across Google data centers. Both P-Tracer and GWP provide web-based query interfaces for data analysis. Unlike our system, P-Tracer and GWP are proprietary, and lack the ability to compare performance based on application-specific metadata (e.g., program configuration).

3 Ubiquitous Performance Analysis

Ubiquitous performance analysis aims to simplify application performance analysis for HPC software development teams, and integrate it better into their software development workflows. This section discusses our approach in detail.

3.1 Overview

The major components of our system are the Caliper instrumentation and profiling library [11], the Adiak metadata collection library [1], and the SPOT web frontend [6]. Application developers integrate Caliper and Adiak into their codes by marking major components (kernels, application phases) with Caliper's annotation macros and exporting program metadata with Adiak. Performance measurements can then be enabled by the application through Caliper's new ConfigManager API. Caliper can perform lightweight always-on time profiling of the annotated code regions, but also collect data for more sophisticated performance analysis experiments. Performance data for an application run is initially written to a file, which can be copied to a directory or imported into a SQL database. Users then analyze the collected performance data in SPOT. Policies for instrumentation, performance measurement, and data collection are defined by the application developers and can be tailored to each code. Performance analysts work together with application developers to define appropriate strategies.

3.2 Code Instrumentation

We primarily rely on manual source-code instrumentation for application profiling, where developers place annotation macros into the source code to mark code regions of interest. Many performance debugging tools use symbol translation or automatic instrumentation approaches which do not require source code modifications for profiling. However, for our purposes, manual instrumentation provides distinct advantages:

- *Control.* Manual instrumentation allows for precise control of measurement granularity. Automatic instrumentation methods easily over- or under-instrument programs, resulting in high measurement overheads or clutter.
- *Interpretability.* Manual annotations describe high-level logical program abstractions such as kernels or phases that developers are familiar with. Automated approaches that rely on compiler-generated identifiers often produce obscure associations, particularly with modern C++ template abstractions.
- *Consistency.* Much of our work involves performance comparisons between different program versions. Identifiers like function names and source line numbers change frequently during development, making comparisons based on such associations difficult. In contrast, the logical program structure expressed in manually instrumented regions typically remains much more stable, allowing for meaningful performance comparisons over long time spans.
- *Reliability.* Many traditional profiling tools rely on binary analysis and the DWARF debugging information to correlate performance metrics to code. This is a common source of complexity and fragility, as not all compilers prioritize correct DWARF information or easily analyzable binary code. By relying on manual instrumentation with tight application integration, we can avoid the traditional attribution complexity and easily integrate our profiling infrastructure with an application's regular testing framework.

The placement of instrumentation annotations follows the logical subdivisions of the code, such as computational kernels and communication or I/O phases. While the one-time setup costs for adding instrumentation annotations could be prohibitive for one-off performance debugging tasks, they are less of a concern for implementing long-term, continuous monitoring strategies. The annotations are not meant to pinpoint specific bottlenecks, but should allow developers to monitor and study the performance evolution of the code. If developers find performance issues in an annotated code region and need more detailed information, they can conduct follow-up experiments with Caliper's complementary sampling-based measurement mechanisms or third-party performance debugging tools to identify root causes.

The Caliper library provides high-level macros to mark functions, loops, or arbitrary code regions in C, C++, and Fortran programs. In addition, many of LLNL's large, long-lived codes already have existing instrumentation for lightweight timing functionality, which we can adapt to invoke Caliper calls instead. Caliper preserves the nesting of stacked regions, and combines annotations from independent components (e.g., libraries), providing complete context information for the combined program across all layers of the software stack. Once in place, the annotations can stay in the code permanently. New annotations can be added incrementally as needed.

3.3 ConfigManager: A Measurement Control API in Caliper

Complementing the instrumentation API, Caliper includes a wide range of profiling capabilities. Essentially, Caliper serves as a built-in profiling tool embedded in the application codes.

We have enhanced Caliper with the ConfigManager API that lets applications control performance profiling activities programmatically. ConfigManager accepts profiling commands in the form of short configuration strings. This configuration string is typically provided by the user as an application configuration file or as a command-line parameter. The configuration specifies an *experiment*, which determines the kind of profiling to be performed, and *options* to customize output or enable additional functionality. Some experiments print human-readable output, while others write machine-readable files for post-mortem analysis in SPOT or other tools. For example, `runtime-report` prints a tree with the time spent in the instrumented regions; `hatchet-region-profile` writes a per-thread region time profile for processing with the Hatchet call-tree analysis library [9]; `event-trace` records a timestamp trace of enter and leave events for the instrumented regions; and `spot` writes a region time profile for analysis with the SPOT web interface. In addition to basic runtime profiling, Caliper provides advanced measurement functionality for specific analyses that can be enabled via runtime options for the selected configuration. Available options include time-series analysis for loops, MPI function profiling, memory high-water mark analysis, I/O profiling, CUDA profiling, hardware-counter access, and top-down analysis for Intel CPUs. Measurements are only enabled on demand, and we take care to avoid interference with production runs or third-party profiling and tracing tools.

The ability to enable complex profiling configurations through a simple application switch greatly simplifies performance measurements, especially for application end users. Some of Caliper's built-in experiments support basic performance debugging tasks: Examples include call-path sampling experiments to capture application details beyond user-defined source code annotations. Caliper also interoperates with other performance tools. For example, we provide adapters that forward Caliper annotations to third-party instrumentation libraries, so that the Caliper-annotated regions are visible in tools like NVIDIA NSight or Intel VTune - a tremendous benefit for developers who regularly use these specialized tools on large codes. In turn, Caliper is available as a backend for the ultra low-overhead TiMemory [23] instrumentation framework.

3.4 Adiak: A Library for Recording Program Metadata

Ubiquitous performance analysis lets users compare performance results from many different application runs. To make meaningful analyses, we need descriptive metadata to capture the provenance of each dataset: for example, it makes little sense to compare the performance of a 1-dimensional test problem against a 3-dimensional multi-physics problem. Metadata helps the user group or filter out datasets when comparing runs. Useful metadata can include environment information such as the machine the program was running on, the launch date and time, or the user running the program; program information such as program version, build date, and compiler vendor and options; and job configuration such as MPI job size and number of threads. In addition, developers often run performance studies based on application-specific input and configuration parameters, such as problem description, problem size or enabled features. We need a customizable solution that can capture these application-specific parameters. We also want to collect this data automatically and avoid manual data input for each run. Therefore, similar to the region annotations for profiling, we record metadata programmatically through an API. We created the Adiak [1] library for this purpose. Adiak records user-defined metadata in the form of key/value pairs. It also includes functionality to fetch common metadata like MPI job size or launch date automatically. The recorded metadata values are stored in the Caliper performance profile datasets.

3.5 SPOT: A Web Interface for Ubiquitous Performance Analysis

Web-based visualization tools are extremely convenient as they do not require the installation of specialized visualization tools. SPOT, our data visualization frontend, is a custom web interface for ubiquitous performance analysis. Compared to traditional profiling tool GUIs, which deep-dive on the performance of individual runs, SPOT analyzes and tracks the performance of many runs over an application's lifetime. At LLNL, SPOT is hosted locally by Livermore Computing (LC) and is available to every LC account holder via LC's web portal. We also provide a containerized version [6] that can be deployed at other sites. SPOT reads data directly from a user-provided directory on a shared filesystem or a database link through a background data-fetching process, which runs as the logged-in user. Thus, filesystem or SQL database permissions ensure that users can only access performance data for which they have appropriate permissions. SPOT provides tools to filter, visualize, and compare performance data, with novel visualizations specifically targeting the analysis of large collections of performance data. Users can create plots to display any of the collected metadata values and performance metrics. They can also open SPOT datasets in Jupyter [5] notebooks directly from the SPOT web page to create custom analysis scripts and visualizations. We discuss specific visualization examples in Sect. 4.4.

3.6 Ubiquitous Data Collection

Caliper provides the `spot` profiling configuration that produces datasets for analysis with the SPOT web interface. As a baseline, these datasets contain a summary time profile with the total, minimum, maximum, and average time spent in each annotated region across MPI ranks, as well as all recorded metadata for a program run. The datasets are usually quite small, in the order of kilobytes.

For comparisons studies in SPOT, all recorded datasets are copied to a shared directory or a SQL database. Depending on the use case, developers and users can manage these datasets manually, or set up automated workflows for long-term, continuous data collection. They can define and implement data retention or purge policies as needed, otherwise storage requirements grow linearly as datasets are added. The SPOT web frontend has options for limiting the amount of data to be imported, e.g. only the last N days, to maintain scalability.

4 Example: LULESH

In this section, we describe the practical implementation of ubiquitous performance analysis in an HPC code using the Lulesh proxy application [3,20] as an example. As a baseline, we use Lulesh 2.0 with MPI and OpenMP parallelization. We show how the code is prepared for profiling and illustrate the analysis capabilities of our web interface.

4.1 Region Instrumentation with Caliper

Lulesh contains 39 computational functions and 5 communication functions in C++, as well as a number of data initialization and utility functions. In Lulesh, function names and the logical subdivision of code semantics along function boundaries provide a good basis for meaningful performance analysis. We instrumented 17 of its top-level computational functions, the 5 communication functions, and the main loop with Caliper annotation macros. To keep clutter and measurement overhead low, utility functions and very small functions were not instrumented. The `CALI_CXX_MARK_FUNCTION` macro in `LagrangeLeapFrog` in Listing 1.1 demonstrates function annotations in Lulesh. Here, Caliper creates a function region from the location of the macro to the function exit, with the name taken from the compiler-provided `__FUNCTION__` macro.

4.2 Metadata Collection with Adiak

In addition to the function instrumentation, we added Adiak calls in Lulesh to collect run metadata. As shown in Listing 1.1, Adiak provides two types of calls: The first form accesses built-in functionality to collect common information, such as the `adiak::user()` call to record the user name, while the second, generic `adiak::value()` form lets developers provide custom metadata in the form of key-value pairs. Adiak can record a variety of datatypes, including integer and floating-point scalars, strings, tuples, and composite types such as lists.

In Lulesh, we record basic environment information like the user name, machine, launchdate, and MPI job size. In addition, we record the Lulesh problem settings, such as the maximum number of iterations, problem size, number of regions, region costs, and region balance. We also record the user-defined "figure of merit" performance number computed by Lulesh at the end of the run. Note that Listing 1.1 shows only a subset of the Adiak calls.

Listing 1.1. Configuring Caliper and recording metadata in Lulesh

```
void LagrangeLeapFrog(Domain& domain) {
  CALI_CXX_MARK_FUNCTION;
  // (...)
}

int main(int argc, char* argv[]) {
  // (...)
  cali::ConfigManager mgr(opts.caliper_config);
  mgr.start();

  adiak::user();
  adiak::launchdate();
  adiak::value("iterations", opts.its);
  adiak::value("problem_size", opts.nx);
  // (...)

  CALI_MARK_FUNCTION_BEGIN;
  // (...)
  CALI_MARK_FUNCTION_END;

  mgr.flush();
}
```

4.3 Integrating the Caliper ConfigManager API

To enable and control performance measurements in Lulesh, we use the Caliper ConfigManager API. Listing 1.1 shows the relevant steps: First, we create a `ConfigManager` object and initialize it with a user-provided configuration string. The ConfigManager class parses the configuration string and sets up Caliper's performance measurement and data recording components. Next, we invoke the ConfigManager's `start()` method to begin profiling based on the given performance measurement configuration. At the end of the program, we invoke the `flush()` method to stop profiling and write out the recorded performance data.

In Lulesh, users provide the Caliper configuration string via a command-line parameter. As an example, we can enable the runtime-report experiment on the command line to print out an aggregate time profile of the user-annotated regions at the end of the execution. With the profile.mpi option, the experiment also wraps and measures all MPI calls:

```
$ ./lulesh2.0 -P "runtime-report(profile.mpi)"
```

In our experience, controlling performance profiling through application-specific means like configuration files or a command line parameter has proven to be very convenient for users, and we encourage developers to provide this capability when adopting Caliper.

4.4 Data Analysis and Visualization in SPOT

For demonstration purposes, we recorded 1,149 profile datasets with Lulesh using different program configurations. For analysis, users load the SPOT website and point it to a directory or SQL database with the recorded datasets. SPOT then populates the landing page, where users can start their analysis.

Landing Page. The SPOT landing page serves as entry point for performance studies, and lets users filter runs of interest out of potentially thousands of profiling datasets. The landing page is populated with charts that show summary histograms for selected metadata attributes, for example the runs performed by a particular user or runs that invoked a particular physics package. The histogram charts on the landing page are interactive and connected through a crossfilter system [2]. Users can select subsets of data in one or more charts, causing the remaining charts to adapt to include only the selected datasets. This is useful to select specific subsets and to discover correlations between metadata variables.

For our Lulesh example, Fig. 1 shows the distribution of runs with a given compiler, "figure of merit" (FOM), input problem size, and number of threads in the 1,149 Lulesh runs. In Fig. 2, the user applied a crossfilter to select the runs that had the highest figure of merit, which shows that those runs predominately were done with binaries produced by the Intel compiler, input problem size 30, and one thread. The original 1,149 datasets were reduced to 24 entries by the "figure of merit" selection.

Comparison Page. The SPOT comparison page is a powerful tool for comparing performance profiles from multiple application runs. Users can select datasets on the landing page using the crossfilter, and open the comparison page to show the performance for all selected runs in a stacked line graph. Users can also group data using additional metadata flags, for example to compare performance between different compilers or MPI versions. A typical comparison configuration for tracking nightly test performance might show a chart per group of tests (where the tests in a group could be defined by metadata values), the test date on the x-axis, and the sum of walltime performance for every test in the group on the y-axis.

Figure 3 shows the runtime in different instrumented code regions for a set of runs in our Lulesh datasets, ordered by the launchdate of the job and grouped by compiler. The colors in the chart correspond to the different instrumented code regions. Users can select the regions shown in the chart in the region hierarchy overview in the lower left. The bottom part of the comparison page shows detailed information for the dataset selected in the chart with the black bar.

Users can group and order datasets using any of the recorded metadata attributes, providing a great deal of flexibility to conduct a wide variety of analyses. Figure 4 shows an interesting example. Here, we ran additional experiments with Lulesh with 343 MPI processes, using three different MPI implementations (mvapich2 v2.3, OpenMPI 2, and OpenMPI 4) and different problem sizes, with

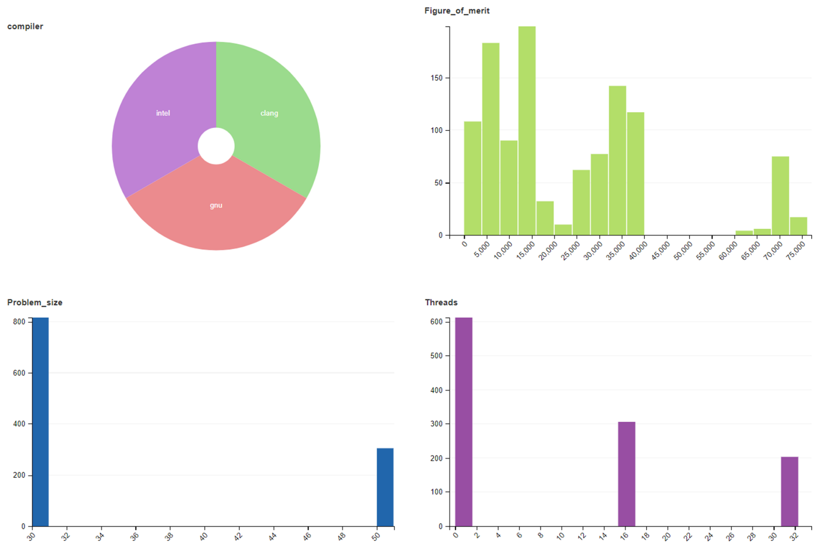


Fig. 1. The SPOT landing page featuring four histogram charts for a set of Lulesh runs. Charts show the numbers of runs with certain metadata values; here: compiler, figure-of-merit (FOM), input problem size, and number of threads.

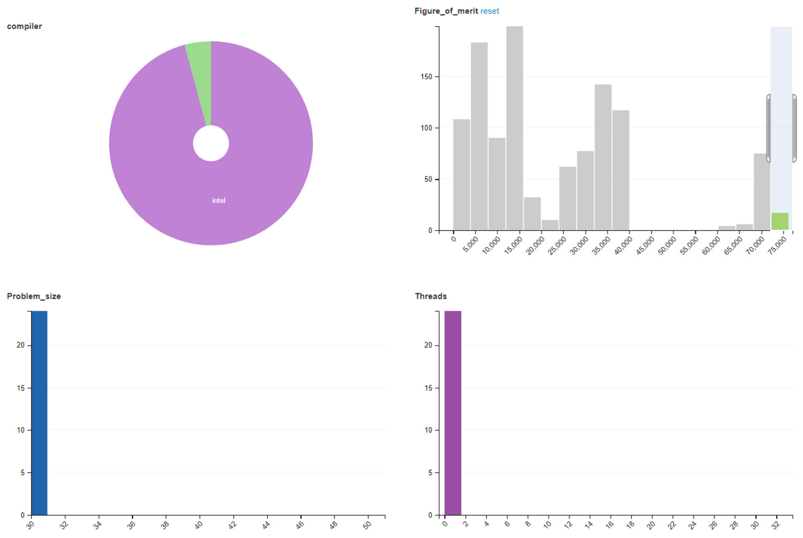
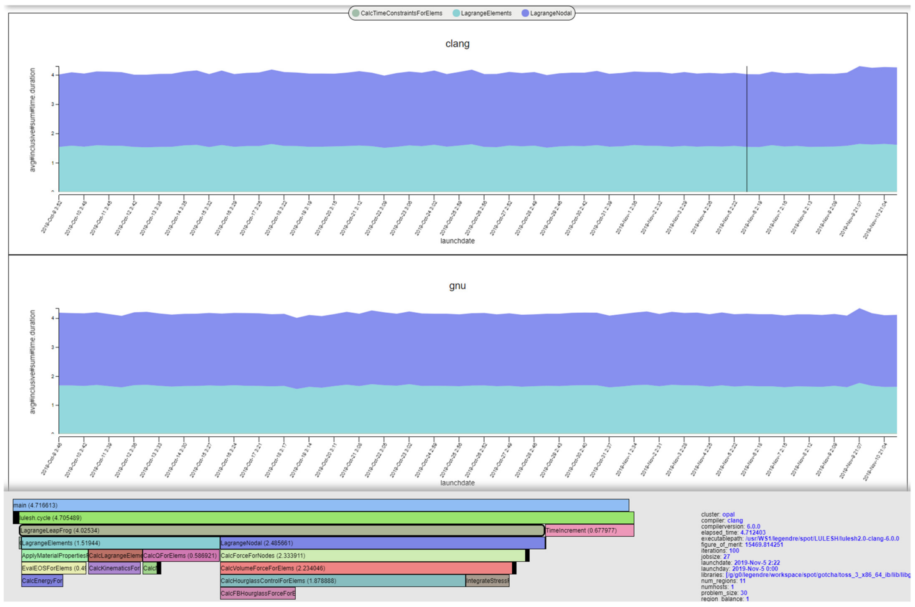


Fig. 2. The landing page charts from Fig. 1 with a crossfilter applied. Selecting runs with highest FOM (top right) shows relationship to compiler (top left), problem size (bottom left), and threads (bottom right).



cluster: opal
compiler: clang
compilerversion: 6.0.0
elapsed_time: 4.712403
executablepath: /usr/WS1/legendre/spot/LULESH/lulesh2.0-clang-6.0.0
figure_of_merit: 15469.814251
iterations: 100
jobsize: 27
launchdate: 2019-Nov-5 2:22
launchday: 2019-Nov-5 0:00
libraries: [/g/g0/legendre/workspace/spot/gotcha/toss_3_x86_64_ib/lib/libg
num_regions: 11
numhosts: 1
problem_size: 30
region_balance: 1

Fig. 3. The SPOT comparison page, here showing runtime (y-axis) for a set of Lulesh runs, ordered by job launch date (x-axis) and grouped by compiler (top and bottom charts). Colors correspond to instrumented code regions. The bottom pane shows details for the highlighted dataset (marked by the black bar in the upper chart).

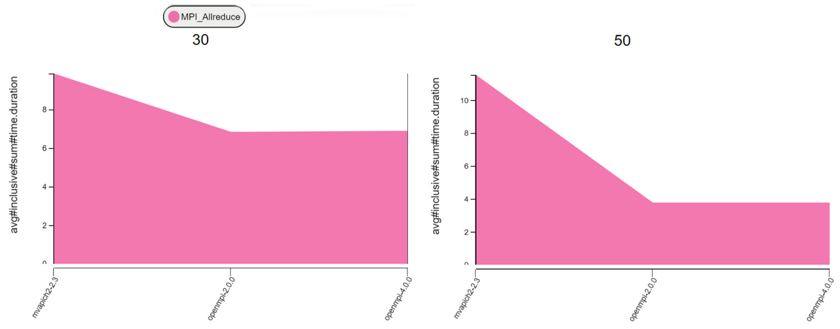


Fig. 4. Users can order datasets in the SPOT comparison page by any recorded meta-data attribute. Here, we compare the average total time in `MPI_Allreduce` per rank (y-axis, seconds) in Lulesh in different MPI implementations (x-axis), for two different input problem sizes (30 and 50; left and right charts).

all other configuration parameters fixed. In the chart, we show average total runtime spent in `MPI_Allreduce`, ordered by MPI version on the x-axis, and grouped by Lulesh problem size. We see that in our tests, OpenMPI outperformed mvapich, especially at large problem sizes.

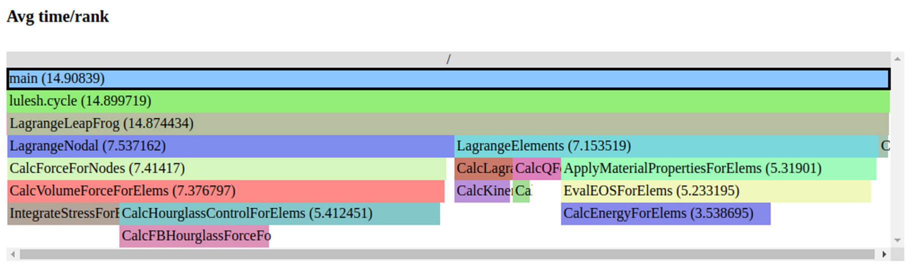


Fig. 5. A detailed performance profile view for a single Lulesh dataset in SPOT showing code hierarchy plots for recorded performance metrics (here: average time in seconds per MPI rank) within instrumented code regions.

Detail Views. From the landing page, users can open detail views for individual datasets, such as a flame graph visualization showing the time spent in each annotated regions. Figure 5 shows a flame graph visualization for the time spent in the instrumented code regions for a single Lulesh dataset.

5 Overhead Evaluation

It is critical that measurement activities do not negatively impact program performance. We quantify the measurement overheads in our Lulesh example when recording SPOT data. We compare four different configurations: an uninstrumented executable ("No instrumentation"), the Caliper-instrumented version with no measurements enabled ("No measurement"), recording a basic region time profile for SPOT ("Spot"), and recording region profile for SPOT with MPI function profiling enabled ("Spot+MPI"). Our experiments ran on Quartz, a 2,634-node cluster system at LLNL with Intel OmniPath interconnect, dual 18-core Intel Xeon E5-2695 2.1 GHz processors, and 128 gigabytes of memory per node. We use Caliper v2.3.0 and Adiak v0.1.1. Both Lulesh and Caliper were built with gcc 4.9.3. Caliper was compiled with optimization level `-O2`, Lulesh with `-O3`. We ran this experiment on a single allocated node using 8 MPI processes and 4 OpenMP threads per process using the Lulesh default input problem.

We ran each configuration 5 times and report the minimum, maximum, and average runtime with each configuration. Figure 6 shows the results. The runtime of the uninstrumented Lulesh executable was between 32.8 and 33 s, with an average of 32.9 s. The average runtime of the instrumented program is virtually unchanged with 33.0 s. When recording basic region time profiles for SPOT, we see a 1.3% runtime overhead in the instrumented Lulesh. The overhead increases slightly to 2% with MPI profiling turned on. Measurement overhead depends heavily on the instrumentation granularity. For our typical ubiquitous performance analysis use cases, we only instrument high-level program regions; therefore, measurement overheads generally stay low in produc-

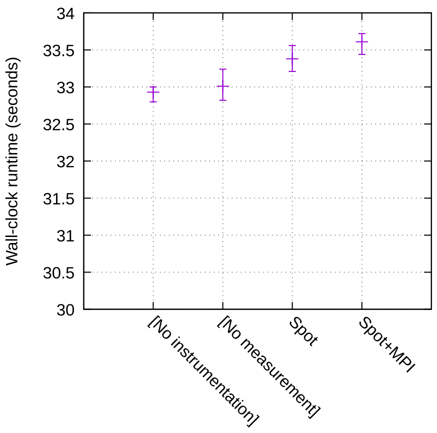


Fig. 6. Caliper instrumentation and measurement overhead in Lulesh with different embedded performance measurement configurations enabled. The bars show the average wall-clock runtime and runtime variation over 5 runs for each configuration.

tion use. In absolute terms, results from the Caliper-provided cali-annotation-perftest benchmark program on our test machine show average costs for a single Caliper instrumentation event (i.e., enter or exit of an instrumented region) of 0.65 µs in the Spot runtime profiling configuration, and 0.12 µs with no active profiling configuration.

The data collection step producing the SPOT output file uses Caliper's flexible aggregation mechanism, which offers $O(\log N)$ scalability over N MPI ranks [10]. Otherwise, Caliper performs no inter-process communication during program execution. Because we record only aggregate information, the amount of data stored on each process during execution remains constant, and only depends on the number of instrumented code regions. The resulting profiling datasets for individual program runs are quite small: in our Lulesh example, the dataset size is 10 KiB per run for the basic region time profile and 14 KiB for the time profile with MPI functions.

6 Case Study: Marbl

This section discusses our experience integrating ubiquitous performance analysis into Marbl, a large multi-physics production code that simulates high energy density and focused physics experiments driven by high-explosive magnetic or laser based energy sources.

Integrating Caliper and Adiak into Marbl was relatively easy and took approximately two man-weeks of developer effort, including coding, testing, reviews, and integration. The details of the integration effort were largely similar to the Lulesh example in Sect. 4. One notable addition is that Marbl also exposes its annotations to users in the form of `lua` functions that can be added at runtime:

- `annotation_begin(name)`
- `annotation_end(name)`
- `annotation_metadata(key,value,category)`.

This allows users (and CI suites) to easily tag and compare the performance for different configurations of a problem.

Unlike many other large LLNL applications, Marbl did not already have built-in timers where we could hook in Caliper annotations. We used HPCToolkit [7] to quickly identify approximately a dozen interesting regions of code that we then annotated, which was enough for Marbl's developers to start using SPOT. The Marbl development team then iteratively refined and added code annotations as they used the tool. A Caliper experiment that counted annotation executions was useful for identifying annotations that were too low-level, such as when an annotation was added in an inner loop and briefly caused a performance regression (seen as a spike from December 20–30 in Fig. 7).

A motivating factor for integrating SPOT into Marbl was to track performance regressions in nightly tests. Marbl's nightly continuous integration (CI) test scripts were modified to drop a Caliper performance file into a persistent directory, which the SPOT web interface uses as a data source. This required only minor changes to the existing CI scripts, specifically, enabling the `spot` ConfigManager configuration for a subset of test instances designated for performance testing. The nightly tests track performance on CPU and GPU architectures over several different configurations of around ten benchmark problems. Each of the ~80 test runs generates a ~30KiB dataset. Since there were too many tests for a human to look at each test's individual daily performance, we grouped tests by their set of utilized physics packages using Adiak-collected metadata. SPOT's comparison view was set to show each test group's (determined by tests that utilized the same physics packages) aggregate performance over a time period. If a test group shows a performance anomaly, a Marbl developer can then use SPOT to view the performance of individual tests in the group, or the aggregate performance of certain code regions in a test or test group. The SPOT configuration that shows any particular view is reflected in the URL, so Marbl developers can bookmark the test results page or send a particular view to a colleague.

The Marbl development team had other uses for SPOT. During development of a new algorithm they wanted to measure the scaling performance and memory overhead of a region of code. They ran before-and-after versions of the code at various scales and collected the automatically-generated performance files into a directory. By pointing SPOT at that directory and selecting a few options in the comparison view, they were able to easily create before-and-after scaling graphs of that code region. This effort generated a request for more complicated graph types, which eventually led to a SPOT feature to automatically export performance data for sets of runs into a Jupyter notebook, where Python's powerful data analytics tools and graphing infrastructure can be used to slice data into highly-customizable visualizations and graphs (see Sect. 3.5).

Ubiquitous performance analysis has also been instrumental in helping the Marbl development team track and understand the code's performance as they

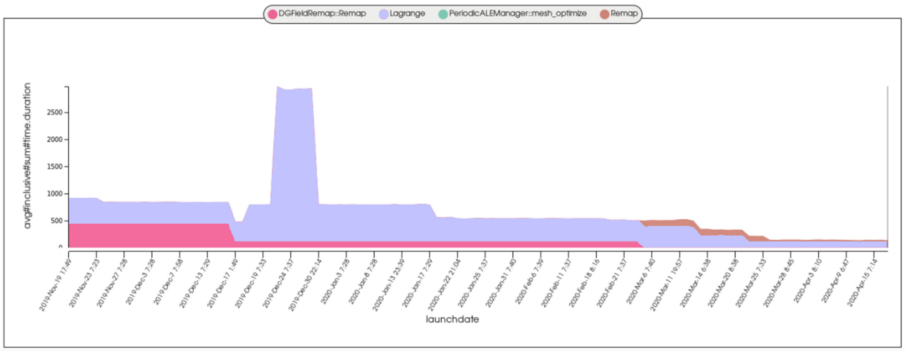


Fig. 7. SPOT performance tracking for Marbl's `Triple-Point-3D` problem on a 4 GPU compute node over the course of several months during its ongoing GPU port. Ubiquitous performance analysis made it easy to detect a performance regression (in late December 2019) and can seamlessly handle changing annotation labels, such as when the "DGFieldRemap::Remap" annotation (pink) was renamed to "Remap" (brown) around March 2020. (Color figure online)

port the codebase to new architectures, an ongoing effort which began in Fall 2019. Figure 7 shows the performance of a 3D Triple-Point hydrodynamics problem on a single IBM Power9 node with 4 NVIDIA Volta GPUs over about a six month period. Automatic performance capture has also helped the team ensure that there have not been performance regressions on other platforms during the porting process. Similarly, ubiquitous performance analysis made it easy to set up Node-to-Node performance scaling studies in Marbl to compare the code's performance across several HPC architectures including Intel- and ARM-based CPU clusters as well as a GPU-based cluster [8]. The integrated Caliper annotations enabled comparisons across different phases of the simulation, custom metadata annotations enabled filtering by scaling study type (e.g. strong-scaling, weak-scaling and throughput scaling) and the Jupyter integration streamlined the process of analyzing and charting the data.

7 Conclusion

Ubiquitous performance analysis seamlessly integrates performance profiling into HPC software development workflows. It facilitates continuous recording, analysis, and comparison of program performance data for long-lived HPC codes. We have created and deployed new software infrastructure to accomplish this goal: the ConfigManager API in Caliper to embed programmatically controlled, always-on profiling capabilities into applications; the Adiak metadata collection library; and the SPOT web interface with novel visualization and analysis tools to explore large collections of performance datasets. Our entire ubiquitous performance analysis stack is developed and released as open source packages [1,6,11].

The Marbl case study shows how ubiquitous performance analysis enables automated performance regression testing and custom cross-platform studies, and greatly simplifies collaborative performance optimization work in large development teams.

At LLNL, we continue to integrate Caliper and SPOT into additional in-house production codes. We also continue to develop new turnkey-style measurement options in the ConfigManager interface with matching visualization tools in SPOT for specific analyses. In that regard, we see the ubiquitous performance analysis software stack as an ideal platform to deploy new performance analysis methodologies. Finally, we recognize that a large amount of long-term performance data can be obtained through automatic data collection, and we expect that this data will enable a wealth of new automated performance analysis approaches based on data mining and machine learning.

Acknowledgment. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and supported by the Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program. The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. LLNL-CONF-808977.

References

1. Adiak: Standard interface for collecting HPC run metadata. https://github.com/LLNL/Adiak. Accessed 16 Mar 2020
2. dc.js - dimensional charting javascript library. https://dc-js.github.io/dc.js/. Accessed 7 Apr 2019
3. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). http://computation.llnl.gov/casc/ShockHydro
4. NVIDIA CUDA Profiling Tools Interface. https://developer.nvidia.com/CUPTI-CTK10_2. Accessed 8 Apr 2020
5. Project jupyter. https://jupyer.org/. Accessed 10 Apr 2019
6. SPOT Container. https://github.com/llnl/spot2_container. Accessed 31 Mar 2021
7. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurrency Comput. Pract. Experience **22**(6), 685–701 (2010)
8. Anderson, R., et al.: The Multiphysics on Advanced Platforms Project. Technical Report LLNL-TR-815869, LLNL (2020). https://doi.org/10.2172/1724326
9. Bhatele, A., Brink, S., Gamblin, T.: Hatchet: Pruning the overgrowth in parallel profiles. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, SC 2019. Association for Computing Machinery (2019). https://doi.org/10.1145/3295500.3356219
10. Böhme, D., Beckingsale, D., Schulz, M.: Flexible data aggregation for performance profiling. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 419–428 (2017). https://doi.org/10.1109/CLUSTER.2017.34

11. Böhme, D., et al.: Caliper: performance introspection for HPC software stacks. In: Supercomputing 2016 (SC 2016). Salt Lake City (2016). lLNL-CONF-699263
12. Brunst, H., Hoppe, H.C., Nagel, W.E., Winkler, M.: Performance optimization for large scale computing: the scalable VAMPIR approach. In: Proceedings of the 2001 International Conference on Computational Science (ICCS 2001), San Francisco, pp. 751–760 (2001)
13. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
14. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency Comput. Pract. Experience **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.1556, http://apps.fz-juelich.de/jsc-pubsystem/pub-webpages/general/get_attach.php?pubid=142
15. Huck, K.A., Malony, A.D.: PerfExplorer: A performance data mining framework for large-scale parallel computing. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. SC 2005. IEEE Computer Society (2005)
16. Huck, K.A., Malony, A.D., Bell, R., Morris, A.: Design and implementation of a parallel performance data management framework. In: 2005 International Conference on Parallel Processing (ICPP 2005), pp. 473–482. IEEE (2005)
17. Huck, K.A., Malony, A.D., Shende, S., Morris, A.: Knowledge support and automation for performance analysis with perfexplorer 2.0. Sci. Program. **16**(2–3), 123–134 (2008)
18. Karavanic, K.L., et al.: Integrating database technology with comparison-based parallel performance diagnosis: the perftrack performance experiment management tool. In: Supercomputing 2005. Proceedings of the ACM/IEEE SC 2005 Conference, p. 39 (2005). https://doi.org/10.1109/SC.2005.36
19. Karavanic, K.L., Miller, B.P.: Experiment management support for performance tuning. In: SC 1997: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, p. 8. IEEE (1997)
20. Karlin, I., et al.: LULESH programming model and performance ports overview. Technical Report LLNL-TR-608824 (2012)
21. Knapp, R.L., et al.: PerfTrack: scalable application performance diagnosis for linux clusters. In: 8th LCI International Conference on High-Performance Clustered Computing, pp. 15–17. Citeseer (2007)
22. Knüpfer, T., et al.: Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-31476-6_7
23. Madsen, J.R., et al.: TiMemory: modular performance analysis for HPC. In: Sadayappan, P., Chamberlain, B.L., Juckeland, G., Ltaief, H. (eds.) ISC High Performance 2020. LNCS, vol. 12151, pp. 434–452. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50743-5_22
24. Mellor-Crummey, J., Fowler, R., Marin, G.: HPCView: a tool for top-down analysis of node performance. J. Supercomputing **23**, 81–101 (2002)
25. Mi, H., Wang, H., Cai, H., Zhou, Y., Lyu, M.R., Chen, Z.: P-tracer: path-based performance profiling in cloud computing systems. In: 2012 IEEE 36th Annual Computer Software and Applications Conference, pp. 509–514 (2012)
26. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: a portable interface to hardware performance counters. In: Proceedings Department of Defense HPCMP User Group Conference (1999)

27. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and Occam Developments, pp. 17–31 (1995)
28. Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., Hundt, R.: Google-wide profiling: a continuous profiling infrastructure for data centers. IEEE Micro **30**(4), 65–79 (2010)
29. Rosinski, J.M.: GPTL-general purpose timing library (2016)
30. Shende, S., Malony, A.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–331 (2006)
31. Skinner, D.: Performance monitoring of parallel scientific applications (2005). https://doi.org/10.2172/881368, https://www.osti.gov/biblio/881368
32. The Open|SpeedShop Team: Open|SpeedShop for Linux. http://www.openspeedshop.org