Caliper: Performance Introspection for HPC Software Stacks

David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer,
Alfredo Gimenez, Matthew LeGendre, Olga Pearce and Martin Schulz
Lawrence Livermore National Laboratory,
Livermore, California 94551

***Abstract*—Many performance engineering tasks, from long-term performance monitoring to post-mortem analysis and online tuning, require efficient runtime methods for introspection and performance data collection. To understand interactions between components in increasingly modular HPC software, performance introspection hooks must be integrated into runtime systems, libraries, and application codes across the software stack. This requires an interoperable, cross-stack, general-purpose approach to performance data collection, which neither application-specific performance measurement nor traditional profile or trace analysis tools provide. With Caliper, we have developed a general abstraction layer to provide performance data collection as a service to applications, runtime systems, libraries, and tools. Individual software components connect to Caliper in independent data producer, data consumer, and measurement control roles, which allows them to share performance data across software stack boundaries. We demonstrate Caliper's performance analysis capbilities with two case studies of production scenarios.**

***Keywords*-Performance analysis; High performance computing; Computer performance; Software tools; Software performance; Software reusability; Parallel processing.**

I. Motivation

Increasing on-node complexity, coupled with thread- and task-parallel runtimes, will make understanding the performance of exascale applications more difficult. Performance problems in modern systems can be caused by many different factors across many different levels in the software stack, from the operating and runtime system to libraries and application code. To diagnose problems, we need introspection capabilities in the form of annotation APIs that allow application and system developers to expose semantic information from their software components. We must also couple this semantic information with comprehensive performance measurement techniques. Finally, to understand the relationships between application code, runtime systems, the memory hierarchy, and the network, we must be able to configure tools to correlate data across software layers.

A wide range of sophisticated profiling and tracing tools can provide such information. Many of them also provide their own APIs to collect application and runtime information. However, these interfaces are tool-focused and require users to add tool-specific actions (like start/stop timer) to their codes, instead of allowing developers to express generalizable semantics of their code in a reusable way. Further, once added, the instrumented code is then typically limited to recording tool-specific profile or trace information, and can no longer be used for other purposes. This discourages or even prevents users from leaving annotations in the code. As a consequence, most pre-installed HPC production software today is not amenable to stack-wide profiling or tracing. Developers who wish to use instrumentation-based tools not only have to instrument their own code, but possibly the entire software stack all the way to the operating system.

In order to overcome this gap, we need a new instrumentation approach that separates the concerns of a) software developers, who expose application/library/runtime semantics through annotations; b) tool developers, who provide measurements to be correlated with application information; and c) tool users, who decide what information is collected, correlated and/or filtered, based on the specific analysis use case, available measurements, and application state. Such an approach must emphasize ease of use for the developer, incurring minimal overhead when used during production runs, and it must be composable across the entire software stack.

In this paper, we present *Caliper*, a library that addresses these requirements. Caliper is a cross-stack, general-purpose introspection framework that explicitly separates the concerns of software developers, tool developers, and users. On the surface, Caliper provides a simple annotation API for application developers, similar to timer libraries found in many codes. In fact, we can wrap Caliper calls inside existing timers. Under the hood, though, Caliper combines information provided through its API from all software layers into a single program context, and it offers interfaces for tool developers to extract this information and augment it with measurements. For this purpose, Caliper includes basic measurement services, but it also offers an interface for existing tools to exploit the stack-wide information. Finally, Caliper users can define a *policy* that links mechanisms to instrumentation points at runtime. By separating annotations, measurement mechanisms and policy, instrumentation annotations can permanently remain in production codes to be used for a wide range of performance engineering applications, from post-mortem analysis to runtime adaption. Moreover, software teams can add instrumentation in different software components independent of each other, allowing correlation of information across software layers. Components are loosely coupled; modifying annotations does not require re-working measurement services, or vice versa.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

Overall, our contributions through Caliper include:

- A novel performance analysis methodology that decouples the roles of application developers, tool users, and tool developers by separating these concerns through separate interfaces:
  - A low-overhead annotation interface that allows developers to convey application information through a flexible data model
  - A robust and efficient interface to provide collected information to tools
  - A flexible mechanism to specify data correlation and filtering policies
- A mechanism to combine the annotation information and performance data across software-stack boundaries by creating a stack-wide context
- An efficient way of representing combined context information in a *generalized context tree*

In two case studies, we demonstrate how Caliper allows application developers to diagnose performance interactions between separate components in a large multi-physics code, and how tool developers can build upon Caliper to access application context information at runtime. Both case studies demonstrate the previously unavailable composability and flexibility Caliper brings to the performance analysis process.

II. The Caliper Approach: Introspection as a Service

Caliper is a universal abstraction layer for general-purpose, cross-stack performance introspection. It is built around the principle of *separating mechanism and policy*: Caliper provides data-collection *mechanisms* (e.g. an instrumentation API, interrupt-based sampling timers, hardware-performance counter access, call-stack unwinding, or trace recording) as independent building blocks, which can be combined into a specific performance engineering solution by a user-provided data-collection *policy* at runtime.

A. Usage Model

Caliper is aimed at HPC software developers, tool developers, and performance engineers alike. Software developers instrument their software components with Caliper annotation commands to mark and describe high-level abstractions, such as kernel names, rank/thread identifiers, timesteps, iteration numbers, and so on. At runtime, the annotations provide context information to performance engineering applications, in terms of abstractions defined by the software developer. Caliper annotations are independent of each other, and developers can add them incrementally anywhere in the software stack. In the long term, we expect these annotations to be kept and maintained permanently in the target components by the component's developers.

Internally, the annotations add or remove entries on a virtual *blackboard*. The combined blackboard contents provide a holistic view of a program's execution context as described by the annotations, across all software stack layers. In addition, the annotations serve as event hooks that can trigger additional user-defined actions at runtime, e.g. for timing a specific code region.

Performance engineers use Caliper as a platform for building measurement analysis tools. In addition to the global context information generated by the source-code annotations, Caliper puts a variety of methodologies at the engineer's disposal, including interrupt-based timers, trace and profile recording, and timer or hardware-counter measurements. Each of these methodologies is an independent building block. Performance engineers can create runtime configuration profiles that combine the building blocks as needed for specific performance engineering use cases.

B. Basic Concepts

Caliper is centered around three basic abstractions: *attributes*, a *blackboard*, and *snapshots*. *Attributes* represent individual data elements that can be accessed through a unique key. The *blackboard* is a global buffer that collects attributes from independent sources. Blackboard updates also serve as event hooks that can be configured to trigger additional actions. Finally, *snapshots* represent measurement events. When a snapshot is triggered, Caliper creates a snapshot record that contains the current blackboard contents, and tells connected measurement providers to take measurements (e.g., a timestamp) and add them to the snapshot record.

Figure 1 illustrates the relationships between those components. Applications, libraries, runtime systems, and tools connect to Caliper in one or more different roles. As *data producers*, they create event hooks and update attributes on the blackboard, or provide measurement attributes for snapshot records. As *data consumers*, they can read the blackboard contents or process snapshot records. They can also perform control tasks, such as triggering snapshots. In this regard, the blackboard and snapshot concepts fulfill two purposes:

- they transparently combine attributes provided by different data producers across the software stack, and
- they enable separation between data producer, data consumer, *and* measurement control roles.

Caliper acts as the glue that connects independent components with similar or different roles across different software stack layers.

III. Data Model

Caliper provides a generic data model that can capture a wide range of information, including user-defined, application-specific concepts. Capturing arbitrary introspection information and storing it so it can be both effectively analyzed at runtime and efficiently streamed to disk is key to making Caliper a viable and useful tool for large-scale performance engineering. To this end, we have developed a data model that is both flexible and efficient. In contrast to traditional, single-purpose performance profile or trace data formats, our data model describes the data's layout and structure rather than its content or semantics. Our model facilitates general-purpose data storage and access similar to non-relational databases, but is optimized for storing the contextual information needed for typical performance engineering use cases.

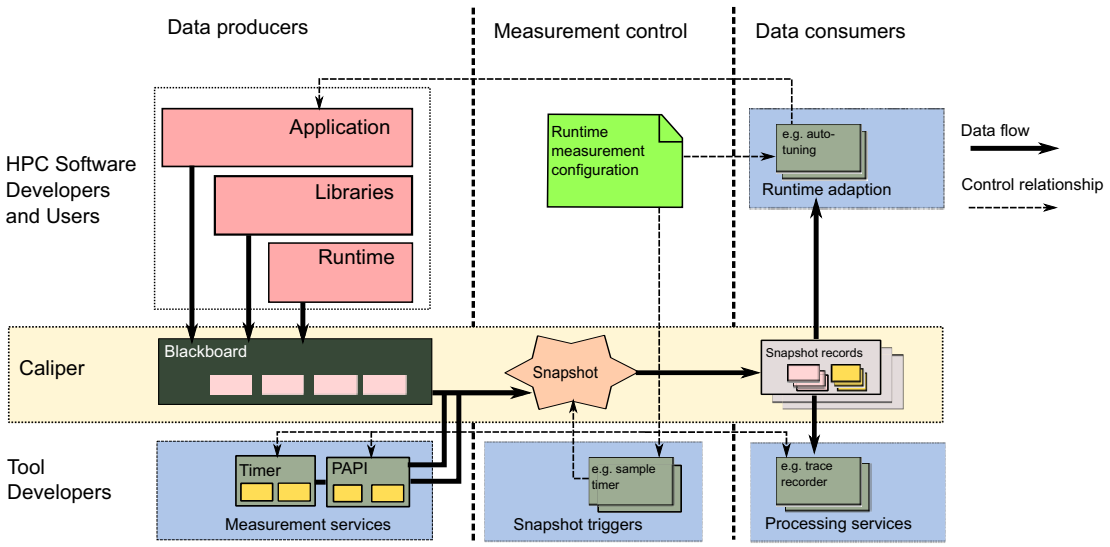

Figure 1. Caliper abstractions enable separation of concerns for performance introspection: Software developers publish program state information on the Caliper blackboard (left column). Tool developers implement data collection, measurement control, and data-processing mechanisms (bottom layer). Based on a user-provided configuration profile, Caliper activates snapshots to connect the individual introspection building blocks at runtime (middle layer). Snapshot records with the combined program context and measurement information can be processed at runtime or stored for post-mortem analysis (right column).

A. Attributes

The basic elements in our data model are *attributes* in the form of key-value pairs. Attribute keys serve as identifiers for attributes: in addition to a unique name, they store the attribute's data type, as well as optional additional metadata (e.g., the measurement unit) associated with the attribute. In additon to integers, floating-point numbers and strings, Caliper also supports user-defined datatypes for attributes.

B. Generalized Context Tree

While the conceptual representation of attributes as key-value pairs is flexible and generic, a naive implementation of blackboard buffers and snapshot records using this representation directly could result in significant storage requirements. Therefore, internally, we encode attributes more efficiently. In particular, we exploit typical characteristics of performance *context* information, such as call paths or named kernels: (1) only a subset of the attributes changes between subsequent snapshots, and (2), due to the iterative nature of typical HPC codes, many attribute combinations likely occur repeatedly during the execution of a program (e.g., the same call paths are visited many times during the execution of a loop).

We use these characteristics to build up a *generalized context tree* at runtime and add to it whenever introspection information is updated. A snapshot can then be represented by a single tree node ID. Conceptually, call-path profilers use a similar concept by storing a call-path ID pointing to a node in the call tree, instead of storing the entire call path with each measurement. However, our approach merges different attributes into a single tree, such that the combined information composed from any number of attributes can be represented by a single tree node.

Figure 2 illustrates the concept. Here, a generalized context tree is built out of three attributes: *phase*, representing a user-defined phase hierarchy as a list; *state*, representing a "parallel" or "serial" execution state; and *iteration*, representing an iteration number. Each attribute can be updated independently; for example, the module that sets the *iteration* attribute does not need to know that the *phase* attribute exists. Caliper builds up the merged tree structure internally whenever the attributes are updated.

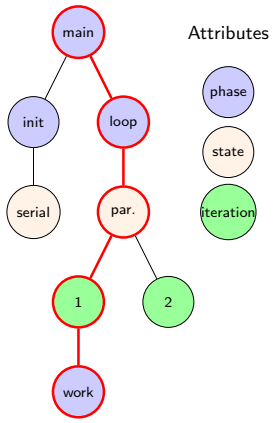

Figure 2. Efficient context information representation. A single pointer to the 'work' node represents the program state described by all nodes on the path to root, composed of three different, independent attributes.

The path from any tree node to the root represents the information composed of all attributes on the path. Thus, a single tree node reference is sufficient to encode an entire snapshot. In the example in Figure 2, the *work* node represents a snapshot that expands to *phase=main/loop/work*, *iteration=1*, and *state=parallel*. Note that the context tree also provides a straightforward way to encode hierarchical attributes, as

demonstrated by the *phase* attribute in the example. We use the flexible design of the generalized context tree to store attribute metadata in it as well, which allows us to encode all metadata in a single structure.

C. Snapshot Records

Snapshots create a list of attributes with the blackboard contents and measurement data at a specific point in the program's execution, which is stored in a *snapshot record*. In contrast to traditional HPC profiling and trace data formats, our data model does not place any restrictions on the kind, number or composition of different attributes in a record. While there is a semantic distinction between context and measurement attributes, we do not explicitly separate them in our data model: both are saved together in a snapshot as generic key-value pairs. For subsequent analysis, the contents of the record receives its meaning from the semantics of the attributes, which we presume to be known to the user who uses or created them.

By encoding of snapshots through context tree references, we reduce the space needed to store snapshot records compared to writing full key-value lists. This also provides a convenient and efficient mechanism to pass a snapshot to measurement tools without having to duplicate the actual introspection information. However, this also means that we need to store the context tree structure itself in order to reconstruct the information. Attributes whose values do change often or rarely reoccur in subsequent snapshots would needlessly increase the context tree size. For example, recording the global time can be very useful but its value never repeats. Therefore, we use a hybrid approach to store snapshot records: we write a context-tree node ID to encode attributes included in the tree, and we write explicit key-value pairs for the remaining attributes. A flag can be given to each attribute key via the Caliper API to decide whether the attribute should be included in the context tree or written explicitly.

IV. Design and Implementation

In the following, we discuss the design and implementation of the Caliper framework. Caliper consists of the source-code annotation API, a backend runtime component that manages blackboard buffers and the generalized context tree, add-on support services for control tasks (such as I/O and controlling snapshots), and additional data producer, measurement control, and data consumer services. The framework also includes libraries and basic post-processing tools for Caliper data streams. Caliper is written in standard C++ and has been tested on a variety of HPC platforms, including standard Linux clusters as well as BlueGene/Q and Cray XC systems.

A. Architecture

As shown in figure Figure 1, the blackboard and the snapshot mechanism are the basic methods to collect data from data producers and provide it to data consumers (including the option to write it out for offline analysis). Data producers and consumers interact with Caliper through annotation and control APIs, or by registering callback functions for certain events. The APIs allow them to

- create attribute keys,
- set, update, or remove attributes on the blackboard,
- query attributes on the blackboard,
- take snapshots, and
- register callback functions.

The callback interface provides notifications about

- the creation of new attribute keys,
- updating or removing attributes on the blackboard,
- snapshot generation,
- snapshot processing,
- I/O requests.

With these interfaces, Caliper enables functional and physical separation of data producers and consumers, as both only interact with the Caliper runtime and not directly with each other.

B. Blackboard Buffer Management

Caliper's runtime component manages the blackboard buffers which store the current values of attributes, and maintains the process-global generalized context tree. This runtime component lives in a single shared object instance per process that is created automatically on the first invocation of the Caliper API.

Caliper maintains separate blackboard buffers for each thread within the same process. To do that, attributes have a *scope* parameter, which is either *thread* or *process*. For runtimes which support lightweight tasks, a *task* scope is also available. The scope can be set at the creation of the attribute. By default, attributes receive the *thread* scope. Updates of thread-scope attributes go into the thread's local blackboard buffer. Consequently, when two different threads update a thread-scope attribute, each thread updates its own copy. In contrast, when different threads update a process-scope attribute, they overwrite each other's values. The blackboard buffer management is completely transparent to the user; she only needs to specify the scope.

Caliper does not explicitly address cross-process data integration. Thereby, we avoid dependencies to specific parallel runtime systems or programming environments, which would be needed to communicate across processes. Snapshots are solely managed on a per-process basis, and it is up to analysis tools to manage snapshot records from multiple processes. Snapshot records are compatible between processes of the same application, since attribute names match. Further, tools can use Caliper to store information about the parallel structure of an application, e.g. by adding process ranks or other task identifiers as attributes to the context of each process, thread or task. This enables analysis tools to provide comprehensive analysis operations across information gathered from large scale parallel applications.

C. Snapshots

A snapshot saves the states of all attributes (i.e., the blackboard contents) at a specific point in the execution

of the target program. Snapshots can be triggered via the `push_snapshot` or `pull_snapshot` API calls. The `pull_snapshot` call returns the snapshot record to the caller, while the `push_snapshot` call forwards the snapshot to other modules through the callback notification interface. Snapshots can be triggered asynchronously at any time, independent of blackboard updates. However, it is possible to trigger a snapshot for some or all blackboard updates by registering a callback function to do so, enabling event-driven data collection workflows. Caliper provides an `event` service module for this purpose.

In addition to the blackboard contents, Caliper can add *transient* attributes to a snapshot. This is done through a callback method that is invoked when a snapshot is triggered, and is used to add measurements such as timestamps to a snapshot.

Snapshots are thread-local; that is, they capture the contents of a single thread and process blackboard buffer. Thus, a snapshot record contains all process-scope attributes, and the thread-scope attributes from the triggering thread. The task of combining data from multiple threads or tasks for analysis purposes is left to the analysis tool.

D. Services and runtime configuration

Much of Caliper's functionality is provided by modules called services, each of which performing a specific task. Services can perform data producer, consumer, or measurement control roles (e.g., triggering snapshots).

In general, services are independent of each other, but they can be combined to form complex processing chains. For example, an event-driven snapshot trace of an annotated target program can be collected by enabling the `event` service (a measurement control service which triggers a snapshot when the blackboard is updated), the `trace` service (a data consumer service that mainatins Caliper's snapshot record output buffer), and the `recorder` service (writes output buffers to disk). Users specify the services that should be enabled via a configuration file or an environment variable before running the target application. By default, no services are active. The following setting loads the event-trace configuration described above:

```
CALI_SERVICES_ENABLE=event:recorder:trace
```

The functionality can be enhanced or modified by adding or replacing services. For example, adding the `timestamp` or `callpath` service adds timestamps or call stack information to the snapshots, respectively. Similarly, the `event` service can be replaced with a `sample` service for triggering snapshots based on timer interrupts instead of attribute updates. The two services could even be combined to create a hybrid event-driven and sampled trace. Many of the services also have their own configuration options to customize their behavior further. Caliper provides pre-defined configuration profiles for common tasks, which users can adapt to their needs. They can also create their own configuration profiles. Overall, the ability to add and remove services as needed gives users great flexibility in creating customized data collection solutions without having to modify the target application or the instrumentation system.

E. Annotation Interface

Caliper provides easy-to-use high-level C, Fortran, and C++ interfaces for source-code annotations. It provides functions to create and update attributes via begin/end annotations to outline temporal or spacial regions in the code or execution, or by directly setting a value. As a typical use case, consider a simple legacy profiling interface with `TimerStart` and `TimerStop` calls created by an application developer for basic performance monitoring. We can easily wrap Caliper annotations within this interface:

```
const struct Timer {
  const char* name;
  // ...
} timers[] = { { "outer_phase", /* ... */ },
               /* ... */ };

void TimerStart(int id) {
  // ...
  cali::Annotation(timers[id].name).begin();
}

void TimerStop(int id) {
  // ...
  cali::Annotation(timers[id].name).end();
}
```

Here, we create a `cali::Annotation` object to access an attribute with the given name (which is taken from the legacy Timer struct in the example) on the Caliper blackboard. In the example, we use the `begin()` method to add a Caliper attribute with the given timer name in `TimerStart`, and the `end()` method to remove it in `TimerStop`. Note that by default, the annotations only update the blackboard, but do not take time measurements. However, we can replicate the original timekeeping functionality by providing a runtime configuration profile that takes snapshots connected with a timestamp source for each timer attribute update. The specific functionality of the profiling interface can now be configured at runtime.

We can also use the annotation interface to export other kinds of execution context information as attributes, for example, the iteration number in a loop:

```
cali::Annotation iter_ann("iteration");

for (int i = 0; i < MAX; ++i) {
  iter_ann.set(i);
  // ...
}
```

Here, we create the `iter_ann` annotation object for the "iteration" attribute in advance. This way, we avoid a having to look up the attribute by name within the loop. We then use the `set()` method at the begin of each iteration to update the iteration information on the Caliper blackboard. In contrast to `begin()`, which would add a new entry, `set()` overwrites the current value on the blackboard. Caliper automatically combines the iteration counter information with the information from all other annotations within the program.

	Cab	rzAlastor
Nodes:		
CPU	Dual 8-core Xeon E5-2670@2.6GHz	Dual 10-core Xeon E5-2680@2.8Ghz
RAM	32GiB	48GiB
Interconnect	InfiniBand DDR	InfiniBand DDR
Software:		
OS	Linux 2.6.32	Linux 2.6.32
Compiler	Intel C++ 16.0.150	GCC 4.9.2

Table I
LLNL CLUSTER CONFIGURATIONS

F. Data Streaming and Processing

A performance experiment typically produces a large, distributed set of snapshot records. Caliper's I/O services write the snapshot records, together with the metadata information (i.e., the generalized context tree structure), into a data stream.

In a distributed-memory environment, Caliper currently produces one context stream per process. Because snapshots are self-contained and have no inherent ordering, multiple data streams naturally lend themselves to parallel processing. In particular, search and filter operations are embarrassingly parallel problems. Moreover, it is possible to perform aggregation operations on one or more attributes across all snapshot records in a stream. As a result, data streams in our model can cover a range of possible performance experiments, from runtime traces to profiles.

V. PERFORMANCE EVALUATION

To evaluate the performance impact of Caliper annotations in a real-world scenario, we compare annotated and non-annotated versions of the LULESH [1] hydrodynamics proxy application. Specifically, we use a modified version of LULESH that uses the RAJA C++ library [2], which decouples the loops in the application from the loop execution parameters to enable static tuning of the loop execution policy. We discuss RAJA in more detail in our case study in Section VI-B.

For our performance evaluation, we added Caliper annotations in the RAJA library and in the LULESH application code to collect one attribute for each RAJA loop invocation, and three attributes for each LULESH main loop iteration. Our experiments ran on the Cab cluster at LLNL (see Table I). We ran annotated and non-annotated versions of LULESH in its default configuration (problem size 30, run to completion) with 16 OpenMP threads on a single cluster node, and we report the wall-clock execution time of its main loop (excluding job setup and I/O). We compare the following Caliper runtime configuration profiles:

- Replacing Caliper with a no-op stub library ("Stub"),
- Performing blackboard updates only, without taking snapshots ("Blackboard"),
- Recording a snapshot trace (one snapshot per blackboard update) without performance measurements ("Snapshot"),
- Recording a snapshot trace with timestamps ("w/ Timestamp")

To mitigate the impact of external effects (e.g., OS noise) on our results, we ran each configuration 5 times, and report the fastest time for each configuration. The run-to-run variation was less than 0.3 seconds (or 3% of the execution time) for each configuration. In the Caliper-enabled configurations, Caliper performs 1,123,087 blackboard updates and snapshots. Snapshots are pushed into Caliper's trace buffer, but not written to disk. In a typical run, data is written to disk when the program exits to minimize perturbation.

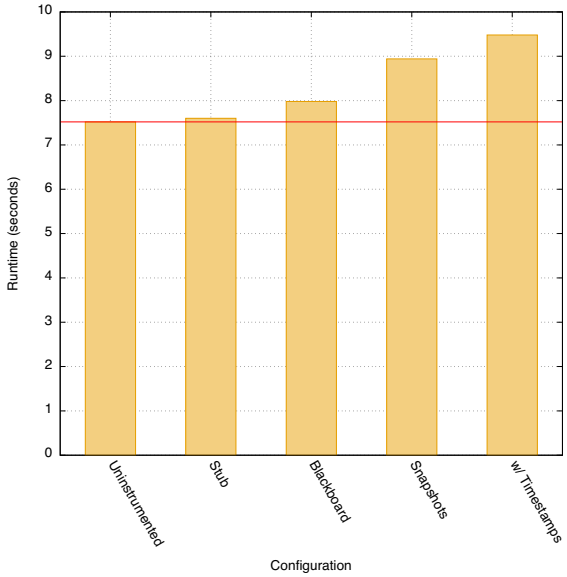

Figure 3. Runtime of the annotated LULESH proxy application for different Caliper runtime configurations. The red line highlights the uninstrumented baseline version.

	Blackboard updates only	with Snapshots	Snapshots with Timestamps
Context tree and blackboards	24.2 KiB	37.2 KiB	38.2 KiB
Trace buffer	n/a	6.4 MiB	8.0 MiB

Table II
CALIPER MEMORY CONSUMPTION PER PROCESS IN THE ANNOTATED LULESH PROXY APPLICATION FOR DIFFERENT CALIPER RUNTIME CONFIGURATIONS.

Figure 3 shows the results. The stub library configuration shows a measurable but small (1% of the execution time) effect of adding additional function calls in the RAJA loop invocations. When Caliper performs blackboard updates but no snapshots, the main loop's execution time increases by 0.46 seconds compared to the uninstrumented version, indicating an average cost of about 0.4 microseconds per blackboard update. Snapshots without additional measurement take about 0.85 microseconds each on average; when adding timestamps, they take 1.3 microseconds. Table II shows Caliper's memory consumption during the Caliper-enabled runs. With less than 40 KiB in each configuration, the memory requirements for the generalized context tree and blackboard buffers are small. Memory requirements for Caliper's trace buffers depend on the snaphot granularity and size. Caliper compresses snapshot records internally to minimize trace buffer sizes. In our LULESH example, Caliper used 8 MiB of memory to store

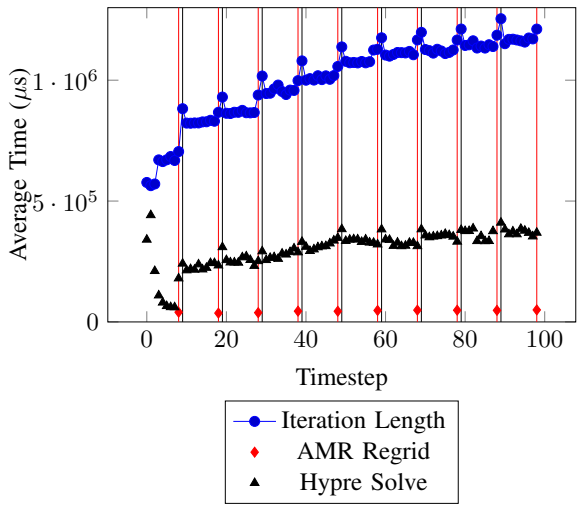

Figure 4. By relating data from independent Caliper context annotations in the HYPRE and SAMRAI libraries, we learn that regrid phases in SAMRAI cause re-setup of HYPRE matrices in the following time step and a longer timestep length as a result.

1,123,087 timestamped snapshots – 7.5 bytes per snapshot on average.

The separation of mechanism and policy allows great flexibility in balancing measurement overheads with application insight. In contrast to traditional introspection approaches, instrumentation and measurement costs, e.g. for taking timestamps or saving snapshot records, occur only when needed. For all but the most fine-grained semantic abstractions (e.g., individual iterations of inner loops), blackboard updates themselves are sufficiently efficient to keep Caliper annotations permanently enabled.

VI. Case Studies

In the following, we demonstrate two Caliper use cases. First, we show how Caliper was used to explain performance behavior in a large multi-physics code by correlating attributes from different libraries. In the second example, we show how we collect Caliper attributes to create predictive performance models, which are then used together with attributes collected online to select optimal settings at runtime.

A. Using Caliper to Instrument a Large Parallel Multi-Physics Code

Here, we study a large radiation hydrodynamics code capable of running small serial to large massively parallel simulations. It is used primarily in munitions modeling and inertial confinement fusion simulations.

This code relies on several numeric libraries, including a structured AMR library SAMRAI [3] and a linear solver library HYPRE [4]. We use Caliper to instrument the individual components and the application itself. While each component is instrumented separately[1], Caliper's shared context allows us to look at how the components impact each other. For example, the ability to annotate the regrid phases in SAMRAI will enable the application developers to anticipate the update of data structures needed for HYPRE along with increased timestep length of the application. This goes beyond classical performance analysis, which provides simple metrics like time spent in a given function, and is a step towards true algorithmic performance debugging. Moreover, any instrumentation in a commonly shared library, such as the widely used HYPRE package, can be reused for studying other simulation applications built on top of the same library. This enables enhanced tool support in applications that have no direct use of Caliper in their main code and without any further efforts on the side of the user. Further, existing Caliper instrumentation does not interfere with new Caliper instrumentation, making it easy to add and remove annotations as application developers see fit. Such instrumentation reusability and composability is key for large software.

To demonstrate these capabilities, we annotated the main application and its libraries. In the main application, we collect the following attributes:

1) Application phase ("main", "loop");
2) MPI rank;
3) Main loop iteration number ("iteration").

Because the HYPRE library is written in C, we used Caliper's C wrapper layer to annotate HYPRE. Here, we collect:

1) Execution phase (hypre_phase, "vcycle" and "loop");
2) The vcyle level ("vcycle_level").

In SAMRAI, we collect:

1) Execution phase (amr_phase, "regrid" and "loop");
2) The regrid level "regrid_level".

We then examined a simulation run that simulates a simplified inertial confinement fusion capsule. All experiments ran on the rzAlastor cluster at LLNL (see Table I). We ran each experiment ten times and averaged the results.

We enabled Caliper's timestamp service to automatically obtain phase duration information. With this configuration, Caliper recorded about 92,000 snapshots per process. In our configuration, we observe between 1 and 5% measurement overhead with Caliper snapshots enabled.

Table III shows an excerpt from the snapshot record stream produced by Caliper. We selected records from a single simulation iteration on a single process; therefore, we omitted the iteration number and MPI_Rank attributes in the table as they are identical in all of the selected records. A line represents a single snapshot record and each record contains all context attributes set by the instrumented components, as well as the duration of the phase in microseconds as measured by the timestamp service. Note that some attributes are not necessarily active in all of the snapshots.

The combined information from all instrumented components allows us to study the influence of parameters of some component on parameters or execution time of other components. For example, we can extract whether the refinement of the AMR grid in SAMRAI impacts the execution time of HYPRE. Using a simple Python script, we parse and

[1]In practice the development team for each library as well as for the application itself would insert Caliper annotation unknowingly from each other, enabling true modularity.

amr_phase=regrid				duration=36
amr_phase=regrid/loop				duration=76019
amr_phase=regrid/loop	regrid_level=1			duration=40
amr_phase=regrid/loop				duration=6
amr_phase=regrid				duration=361714
amr_phase=regrid			phase=main/loop	duration=143816
amr_phase=regrid		hypre_phase=vcycle	phase=main/loop	duration=20
amr_phase=regrid		hypre_phase=vcycle/loop	phase=main/loop	duration=435
amr_phase=regrid	vcycle_level=1	hypre_phase=vcycle/loop	phase=main/loop	duration=358
amr_phase=regrid	vcycle_level=2	hypre_phase=vcycle/loop	phase=main/loop	duration=146
amr_phase=regrid	vcycle_level=3	hypre_phase=vcycle/loop	phase=main/loop	duration=64
amr_phase=regrid	vcycle_level=4	hypre_phase=vcycle/loop	phase=main/loop	duration=39
amr_phase=regrid	vcycle_level=5	hypre_phase=vcycle/loop	phase=main/loop	duration=18
amr_phase=regrid	vcycle_level=4	hypre_phase=vcycle/loop	phase=main/loop	duration=29
amr_phase=regrid	vcycle_level=3	hypre_phase=vcycle/loop	phase=main/loop	duration=60
amr_phase=regrid	vcycle_level=2	hypre_phase=vcycle/loop	phase=main/loop	duration=120
amr_phase=regrid	vcycle_level=1	hypre_phase=vcycle/loop	phase=main/loop	duration=234
amr_phase=regrid	vcycle_level=0	hypre_phase=vcycle/loop	phase=main/loop	duration=243
amr_phase=regrid		hypre_phase=vcycle/loop	phase=main/loop	duration=8
amr_phase=regrid		hypre_phase=vcycle	phase=main/loop	duration=8
amr_phase=regrid			phase=main/loop	duration=286

Table III
SAMPLE OUTPUT OF CALIPER INSTRUMENTATION IN OUR HYDRODYNAMICS APPLICATION, HYPRE AND SAMRAI. ROWS REPRESENT INDIVIDUAL SNAPSHOTS, COLUMNS REPRESENT THE COLLECTED ATTRIBUTES.

cross-compare data from raw context streams as presented in Table III. Figure 4 shows that the timesteps in which the SAMRAI library regrids (shown by red markers) are always followed by re-setup of data structures for HYPRE and therefore longer HYPRE solve time (black markers) in the immediately following timesteps. The outliers in the early timesteps are attributed to cold caches. Both of those operations increase the runtime of the application, which is clearly correlated with the total timestep length of the application. We are able to gather these correlations because we have instrumented SAMRAI, HYPRE, and the main application, and are able to see the information from all three in the same context.

The experience of instrumenting this large hydrodynamics application and its components with Caliper annotations was straightforward and will enable us to study correlations between parameters in different components and their influence on other components' runtime and accuracy. From the software engineering perspective, the ability to instrument a large code incrementally is crucial. Additionally, composability of context annotations means that there is no need to disable subsets of the instrumentation in order to look for features in another part of the code, enabling us to leave the instrumentation in place.

B. Using Caliper for Runtime Tuning with Supervised Machine Learning

Complex scientific applications exhibit dynamic and data-dependent behavior. This behavior determines the parameter selections for tuning the applications runtime. Existing tuning approaches are applied statically, or assume code and data changes slowly, interleaving tuning phases with application execution. However, these slow changes are rarely the case in production scientific applications, where data can vary dramatically each time control passes over a loop. To obtain the best performance, we use lightweight decision models to dynamically tune application parameters at runtime on a loop-by-loop basis in response to changing application data. To achieve this, we require cross-stack data collection and furthermore, runtime access to that data, both of which are provided by Caliper.

To dynamically adjust the application execution configuration at runtime we use RAJA, a C++ library that provides flexible parallel execution methods over defined indices using a combination of template execution methods and lambda functions [2]. RAJA is a programming framework that decouples the loops in the application from the loop execution parameters, enabling static tuning of the policy used to perform the loop iterations. For example, the following application loop:

```
for (int i=begin; i < end; ++i) {
  sigxx[i] = sigyy[i] = sigzz[i] = - p(i) - q(i);
};
```

is transformed into a RAJA loop:

```
RAJA::forall<exec_policy>(IndexSet, [=](int i) {
  sigxx[i] = sigyy[i] = sigzz[i] = - p(i) - q(i);
});
```

We focus on tuning the execution policy, that is, how each loop is mapped to hardware by the RAJA library. Whilst typically most thread-safe loops in an application should be run using threads, in some cases, such as when the workload is small, the overhead of threads means it's actually faster to run in serial. Our model is designed to identify these cases and execute them accordingly.

Our tuning model is built using a decision tree algorithm that infers a function from a training set to a set of target labels. The training set is a subset of the data collected at

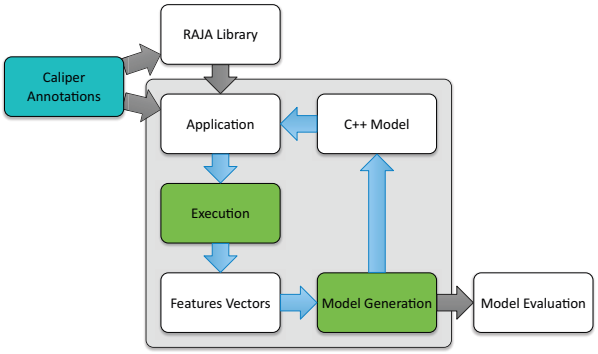

Figure 5. Workflow for automatically generating models to tune application parameters based on Caliper annotations.

runtime by Caliper annotations, consisting of a number of samples, each sample describes one invocation of some loop in the application execution. Each sample is labelled with the best parameter value, that is, the one with the lowest observed execution time. The correct label for each sample is determined by inspection. We assign to each sample a label based on the execution policy that provided the lowest runtime for that particular lambda invocation. The samples not used as part of the training set are reserved for model evaluation. The classifier built during the training step can then be used to predict the output of a test set of samples without any labels. The labels predicted by the classifier for the test set can be used to evaluate the accuracy of the learned function. We then convert the decision trees learned by our model generator into C++ code, which can be compiled and then used at runtime. Figure 5 shows our integrated workflow for collecting features and generating models using Caliper. There are two aspects of the workflow that are enabled by the cross-application flexibility provided by Caliper: data collection, and online data access.

1) Data Collection: A key challenge for generating models using supervised learning is collecting features that may impact application performance, such that they can be used to construct a model for predicting which parameter value is the best for a given loop. To collect data for model generation, we compile the application with an instrumented version of the RAJA library. Caliper annotations are inserted within the RAJA template execution method (see Listing 1). The design of RAJA naturally decouples the application specific loop body from the platform, architecture, and programming model specific loop execution. This means that the features for each loop can be recorded without modifying the application code.

We also measure application level features, such as the current timestep and the global problem size using Caliper annotations inserted into the application source. Caliper allows us to combine all features across the application stack, generating a feature vector for each loop invocation. Table IV describes some of the features we collect. Using Caliper to automatically collect and merge features from all software components, we are able to include features deemed important by the application developers in our model. This functionality is essential in capturing application and domain specific data

Listing 1. Caliper annotations at the RAJA library level.

```
template <typename EXEC_POLICY_T,
          typename LOOP_BODY>
RAJA_INLINE
void forall(const IndexSet& iset,
            LOOP_BODY loop_body)
{
  // Begin Caliper annotations, e.g.
  cali::Annotation("num_indices")
    .set(iset.getLength());
  forall(EXEC_POLICY_T(),
         iset,
         loop_body);
  // End Caliper annotations
}
```

Feature	Description
callpath.address	Call stack of current function.
func	Address of lambda function.
func_size	Total number of instructions in loop body.
index_type	Type of RAJA IndexSet.
num_indices	Total number of indices in IndexSet.
num_segments	Number of segments in the IndexSet.
problem_size	Global problem size (application specific).
stride	Stride of IndexSet segments.
timestep	Current timestep (application specific).

Table IV
FEATURES GATHERED USING RAJA AND CALIPER.

that might be deemed unimportant by a computer scientist, but that has a large impact on application performance. Whilst features such as the global problem size and the current timestep can be considered application specific, since they are accessed using a string identifier (e.g. "timestep"), we can use a common naming scheme to allow cross-application comparison of these attributes.

2) On-line Data Querying for Runtime Adaption: Our modeling workflow generates C++ code that implements the decision process found in the models. This code can be compiled with the application and used to dynamically alter execution policy at runtime. As input, the model requires a sample which contains values for all the features used when training the model. Using Caliper's on-line query interface, we are able to access these feature values from across the application stack within our generated code, and use these values to evaluate the model. A key advantage of using Caliper is that annotations are re-used; the annotations used to record features and create the model are used when the model is being evaluated online.

3) Results: Whilst this work is still in its preliminary stages, by using Caliper's online query interface and our generated data-dependent models, we are able to improve application in simple cases. Figure 6 shows the runtime at a range of processor counts on LLNL's Cab cluster (Table I) using one of our generated data-dependent models in the CleverLeaf mini-application [5]. At small problems sizes the default serial execution is sufficient, but at larger problem sizes, threading is enabled by the model to improve performance.

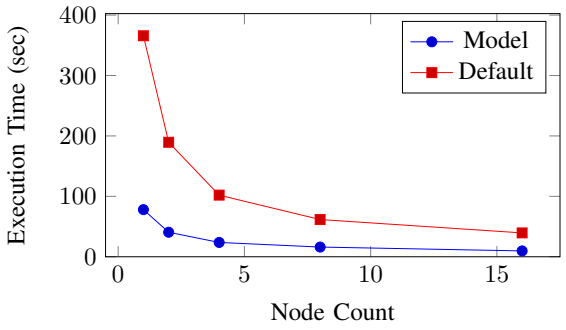

Figure 6. Runtime improvements using the dynamically auto-generated model on up to 256 cores.

VII. Related Work

A. Performance Introspection

The HPC community has developed a wealth of performance introspection and data collection frameworks that cover a wide range of methods, systems, and use cases. In general, performance introspection systems are based on a few basic techniques: instrumentation, i.e., inserting commands to observe a program section of interest into the target code; statistical sampling, which probes the program's state and collects measurement data at regular intervals, e.g. via a timer interrupt; call-stack unwinding, which provides the chain of function calls leading to a performance probe; and interfaces within the software stack that serve as hooks for measurement probes (e.g., the OpenMP tools interface [6]) or provide certain information (e.g., the PAPI [7] interface for accessing CPU performance counter data).

For basic performance monitoring tasks, many HPC applications use home-grown instrumentation solutions with simple time-measurement "calipers" around code sections of interest. On the other end of the spectrum frameworks like Score-P [8], TAU [9], HPCToolkit [10], or Open|SpeedShop [11] collect detailed per-thread execution profiles or traces for in-depth post-mortem analyses, such as automatic bottleneck detection [12], profile analysis [13], [14], or trace visualization [15].

Score-P and TAU primarily use instrumentation to collect measurement data, but also support sampling. In addition to source-to-source translation or compiler-based mechanisms to instrument user functions automatically, they also provide APIs for manual source-code annotation. However, their instrumentation approach creates tool-specific code, requiring users to recompile their code or potentially an entire software stack specifically for performance analysis. In contrast, Caliper's general-purpose annotations can be left in the source code and activated for performance analysis use cases as needed by a runtime configuration profile.

HPCToolkit and Open|SpeedShop use statistical sampling and call-stack unwinding to generate call-path profiles. While this approach enables some form of cross-stack analysis, the call path does not capture application semantics on a relatable level (e.g., kernel names) that can only be obtained through source-code annotations. The Caliper blackboard makes it possible to correlate samples with high-level program context.

B. Blackboard Systems

The general concept of a *blackboard* originated with the Hearsay-II speech understanding system [16], in which many disparate *knowledge sources* stored observations in a common global database. Blackboard systems later grew into a major sub-field of artificial intelligence [17], [18]. With the shift towards complex node architectures, blackboard-like designs have become popular for performance introspection tools.

The Autonomic Performance Environment for Exascale (APEX) [19] – originally developed as introspection interface for the HPX programming model – is a data-sharing layer for cross-stack performance analysis and runtime tuning. APEX shares Caliper's design philosophy regarding the blackboard and separation of concerns. However, the TAU-like data format underlying APEX limits its expressivity to start/stop regions, whereas Caliper's data model captures arbitrary kinds of context information. APEX also hard-wires some of its profiling activities (e.g. taking timestamps). The Resource-Centric Reflection Toolkit [20] (RCRToolkit) also uses a blackboard-like design to monitor parallel runtime systems. RCRToolkit's shared-memory blackboard is managed by a per-node daemon process. Clients communicate with the daemon using data in Google Protocol Buffers [21] with strict, pre-specified schemas.

In contrast to these tools, Caliper's snapshot mechanism provides a more flexible, runtime-configurable way to connect data producers and consumers. Its data model is unstructured and schema-less, which allows component designers and application developers more freedom to quickly extend and modify their annotations and measurements. Additionally, Caliper uses the *generalized context tree* to represent attribute values efficiently as node handles.

VIII. Conclusions

Faced with rising architecture and application complexity paired with ever growing system scales, the ability to understand and optimize application performance is more critical than ever before. While a wide range of performance analysis solutions exist that enable us to collect the performance data for this task, many of them require tool-specific annotation solutions tied to a particular data model that cannot be combined across tools and/or independently annotated software modules.

With Caliper, we address this combinatorial challenge by separating the concerns of software developers, tool developers, and users. With the Caliper annotation API, software developers can expose high-level application semantics for performance analysis in a general-purpose, reusable way – a crucial requirement for our long-term goal to integrate performance introspection across the entire HPC production software stack from the start.

Our two case studies already demonstrate the benefits of this approach: instrumenting a large radiation hydrodynamics code as well as its libraries shows how we can combine context information from independent annotations, and how this helped us to dissect performance information. In the RAJA use case, we demonstrate how Caliper annotations support runtime adaption based on supervised machine learning, a

novel performance engineering approach that requires both off-line and on-line access to context information. Combined, these case studies illustrate how flexible introspection offered by Caliper offers a new path towards effective and insightful performance analysis for complex HPC applications.

SOURCE CODE

Caliper is available at https://github.com/LLNL/Caliper. Documentation is available at software.llnl.gov/Caliper.

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.

LLNL release LLNL-CONF-699263.

REFERENCES

[1] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "Lulesh programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[2] R. D. Hornung and J. A. Keasler, "The RAJA Poratability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[3] B. T. N. Gunney, A. M. Wissink, and D. A. Hysom, " Parallel Clustering Algorithms for Structured AMR," *Journal of Parallel and Distributed Computing*, vol. 66, no. 11, pp. 1419–1430, 2006.

[4] R. Falgout, J. Jones, and U. Yang, "The Design and Implementation of HYPRE, a Library of Parallel High Performance Preconditioners," *Chapter in Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset and A. Tveito, eds.*, vol. 51, no. 4, pp. 267–294, 2006.

[5] D. A. Beckingsale, W. Gaudin, A. Herdman, and S. Jarvis, "Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units," in *Proceedings of the 44th International Conference on Parallel Processing*. IEEE, Aug. 2015, pp. 61–70.

[6] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: OpenMP tools application programming interfaces for performance analysis," in *Proc. of the 9th International Workshop on OpenMP (IWOMP), Canberra, Australia*, ser. LNCS, no. 8122. Berlin / Heidelberg: Springer, 2013, pp. 171–185.

[7] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP User Group Conference*, Jun. 1999.

[8] Knüpfer, Andreas and Rössel, Christian and Mey, Dieteran and Biersdorff, Scott and Diethelm, Kai and Eschweiler, Dominic and Geimer, Markus and Gerndt, Michael and Lorenz, Daniel and Malony, Allen and Nagel, Wolfgang E. and Oleynik, Yury and Philippen, Peter and Saviankou, Pavel and Schmidl, Dirk and Shende, Sameer and Tschüter, Ronny and Wagner, Michael and Wesarg, Bert and Wolf, Felix, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, Brunst, Holger and Müller, Matthias S. and Nagel, Wolfgang E. and Resch, Michael M., Ed. Springer Berlin Heidelberg, 2011, pp. 79–91.

[9] S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[11] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open|speedshop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[12] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010. [Online]. Available: http://apps.fz-juelich.de/jsc-pubsystem/pub-webpages/general/get_attach.php?pubid=142

[13] J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, pp. 81–101, 2002.

[14] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 41–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.55

[15] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[16] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty," *ACM Computing Surveys (CSUR)*, vol. 12, no. 2, pp. 213–253, 1980.

[17] H. P. Nii, "Blackboard application systems, blackboard systems and a knowledge engineering perspective," *AI magazine*, vol. 7, no. 3, p. 82, 1986.

[18] D. D. Corkill, "Blackboard systems," *AI expert*, vol. 6, no. 9, pp. 40–47, 1991.

[19] K. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An Autonomic Performance Environment for Exascale," *Supercomputing Frontiers and Innovations*, vol. 2, no. 3, 2015.

[20] A. Mandal, R. Fowler, and A. Porterfield, "System-wide introspection for accurate attribution of performance bottlenecks," in *Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, Venice, Italy, 06/2012 2012.

[21] K. Varda, "Google's data interchange format," Online, July 7 2008, https://developers.google.com/protocol-buffers/.