**ABSTRACT**

**Thicket:** Seeing the Performance Experiment Forest for the Individual Run Trees

Thicket is an open-source Python toolkit for Exploratory Data Analysis (EDA) of multi-run performance experiments. It enables an understanding of optimal performance configuration for large-scale application codes. Most performance tools focus on a single execution (e.g., single platform, single measurement tool, single scale). Thicket bridges the gap to convenient analysis in multi-dimensional, multi-scale, multi-architecture, and multi-tool performance datasets by providing an interface for interacting with the performance data.

Thicket has a modular structure composed of three components. The first component is a data structure for multi-dimensional performance data, which is composed automatically on the portable basis of call trees, and accommodates any subset of dimensions present in the dataset. The second is the metadata, enabling distinction and sub-selection of dimensions in performance data. The third is a dimensionality reduction mechanism, enabling analysis such as computing aggregated statistics on a given data dimension. Extensible mechanisms are available for applying analyses (e.g., top-down on Intel CPUs), data science techniques (e.g., K-means clustering from scikit-learn), modeling performance (e.g., Extra-P), and interactive visualization. We demonstrate the power and flexibility of Thicket through two case studies, first with the open-source RAJA Performance Suite on CPU and GPU clusters and another with a large physics simulation run on both a traditional HPC cluster and an AWS Parallel Cluster instance.

**KEYWORDS**

HPC, exploratory data analysis, performance analysis, parallel profile, multi-dimensional

**ACM Reference Format:**


1 INTRODUCTION

The rise of complexity in HPC simulations, software stacks, and heterogeneous architectures presents increased challenges for performance optimization. Applications often exhibit many parameters, a complex software ecosystem, heterogeneous hardware configurations, and sophisticated execution context, requiring multiple tools to collect and analyze performance data. Effectively managing and exploring this data is a significant bottleneck to finding actionable insights about the application’s performance slowdowns. To address these challenges, we present an open-source Python toolkit called Thicket. Our solution enables rapid exploration and hypothesis testing of the multi-dimensional, multi-scale, multi-architecture, and multi-tool performance data collected from performance experiments.

Code developers, users, and site operators experience challenges related to multi-dimensional performance analysis. Application developers have to select library and compiler options for every

---

*Also with Texas A&M University.*
platform on which their code is used. They deal with raw performance data collected at various system levels and often have to divert valuable developer time to keep pace with new technologies integrated into supercomputers. Application users run the codes on different computing platforms and settings, validating results with collaborators at other sites, or running on larger systems as their applications scale up. Site operators make optimization decisions by comparing the performance of available choices, i.e., by profiling the target application in a range of different configurations on different systems and potentially multiple complementary profiling tools. Analyzing the results of these ensembles requires considering multiple sources and data types. This includes raw performance data such as runtime per function, additional metrics collected by external tools such as cache misses, and metadata such as the build settings and execution context for individual performance profiles. Performing Exploratory Data Analysis (EDA) on all collected performance data can result in actionable insights and limit performance being left on the table. Still, there does not exist a comprehensive tool that supports the structured analysis of unstructured data. We design Thicket based on the experiences and insights of application code developers, users, and site operators to address these needs.

Thicket improves accessibility and lowers the barrier of entry to performance data exploration by providing a comprehensive interface to drive the decision-making process of application code developers, users, and site operators while allowing the preservation of important information about the execution context and other metadata. To achieve these goals, Thicket is designed as a Python data analysis toolkit. Its crucial element, the thicket object, unifies data collected from multiple tools across multiple executions so it can be easily manipulated, modeled, and visualized. Capabilities include manipulation operations of performance data (e.g., filtering, grouping, and querying), statistical operations and data modeling to derive data across dimensions and metadata with the support of popular external libraries such as scikit-learn and Extra-P; and visualization of data in an intuitive call tree context. The capabilities of Thicket are accessible through powerful Jupyter notebooks for interactive analyses and multi-variate visualizations. We demonstrate how Thicket supports a variety of performance analysis tasks through two case studies, one evaluating the RAJA Performance Suite across CPU and GPU configurations and one evaluating a multi-physics simulation across a traditional cluster and a cloud configuration.

The contributions of this paper are as follows:
- We present Thicket and its components (i.e., performance data, metadata, and aggregated statistics) designed to unify multi-run performance data for EDA;
- We describe the EDA capabilities in Thicket;
- We showcase Thicket on two use cases. First, we study the scalability of RAJA on CPU and GPU using performance data generated by Caliper by running the RAJA Performance Suite. Second, we study the performance of a multi-physics simulation on cloud (AWS) and HPC resources.

2 THICKET: UNIFYING PERFORMANCE DATA FOR EDA

Thicket fills the missing "last step" in the exploratory analysis workflow for multi-dimensional performance data, providing users with the tools needed to comprehensively study an ensemble of
jobs. Figure 1 illustrates integration of Thicket into a traditional performance analysis workflow. Thicket provides essential steps for studying performance data after conventional operations, such as linking code to monitoring tools, running applications, and gathering metrics into call tree profiles, are completed.

Running applications with the support of measurement tools (Step 1 in Figure 1) is traditionally the first step required to collect performance data. Users integrate a profiling library with their applications and target specific regions of interest by inserting instrumentation points into the code. There are many tools available for collecting the performance data, such as HPCToolkit [4, 28], TAU [35], and Score-P [23]. In our work, we use Caliper [9] to collect primary inclusive and exclusive timings per MPI rank, hardware performance counters on the CPU architectures, and GPU measurements on NVIDIA GPUs. We also use NVIDIA’s Nsight Compute (NCU) [3] tool to collect additional GPU metrics.

Generating performance profiles (Step 2 in Figure 1) leaves users with many unstructured performance datasets from the ensemble of executed jobs. Collected data may include quantitative metrics, such as the total time spent on each annotated source-code region, and qualitative metrics, such as hardware and compiler information. Caliper writes both metadata and performance data for each run in a call tree profile; the metadata includes application build settings and execution context, while the measured performance metrics are assigned to nodes in a call tree. The call tree represents the application structure, where each call tree node represents executed functions or nested source code regions. The call tree nodes define connectivity in the call tree, and each call tree node stores its children and parents. The performance metrics are the measurements collected at runtime using the profiling libraries, such as time durations or hardware-counter values. While call tree profiles have proven effective in structuring performance data of single runs, extracting knowledge from ensembles of profiles remains an arduous task.

Thicket uses the collected data to instantiate a thicket object composed of three components: (1) Performance data, (2) Metadata, and (3) Aggregated statistics (Step 3 in Figure 1). Performance data is a multi-dimensional table of measured and derived performance metrics (e.g., job runtimes and L2 miss rate) per node in the call tree profiles for each run of the application. Metadata is a structured collection of information about each run, including the application’s build settings and execution context (e.g., cluster name and time of the run). Aggregated statistics provides a summary of the performance metrics collected by several runs (e.g., variance in runtime across multiple runs, average L2 misses). Figure 2A shows an example of a code with four call sites (functions). Each function has a row of collected performance metrics (Figure 2B). In the example, the code is run twice, generating two profiles. This results in a thicket object with a performance data table containing two rows per function (Figure 2C), a metadata table with the metadata per profile (Figure 2D), and a set of aggregated statistics (Figure 2E) computed across the two profiles.

Thicket provides powerful analysis tools in a set of Jupyter notebooks (Step 4 in Figure 1) with built-in functionality and customized studies to aggregate, visualize, and export performance data from thicket objects. This enables users to explore the collected performance data and extract collective knowledge through exploratory performance analysis (EDA). Examples of EDA available in Thicket include examining and manipulating performance datasets; calculating statistics across experiments; applying external functions such as clustering or principal component analysis (PCA) from scikit-learn; or generating performance models with Extra-P.

### 3 ARCHITECTURE OF THE THICKET OBJECT

A thicket object is a flexible data model that enables the structured analysis of unstructured performance data. We designed its architecture by following a holistic approach that allows studying the different performance dimensions by linking the three object’s components — performance data, metadata, and aggregated statistics — through primary and foreign keys, as shown in the entity relationship diagram in Figure 3.

#### 3.1 Thicket Components

The performance data table is a multi-dimensional, multi-indexed structure with one or more rows of data associated with each node of the call tree, each row representing a different execution (i.e., profile index) of the associated call tree node. The (call tree node, profile index) pair shown in Figure 3 uniquely identifies each row in the performance data. The metadata table stores the applications’ metadata (i.e., application’s build settings and execution context); each row represents a single execution of the application and is identified by a unique profile index. The aggregated statistics table supports an order-reduction mechanism and stores processed applications’ performance. Each row of the aggregated statistics table holds data aggregated across all profiles associated with a particular call tree node. Thicket provides users with capabilities for computing common aggregated statistics on their performance data, such as mean and variance.

Figure 3 shows how the three components relate to each other through indices (i.e., primary and foreign keys). In the figure, the primary keys are in bold and are fixed. The values in the tables are populated dynamically based on the type of analysis. The metadata and aggregated statistics tables are relationally linked to the performance data table through profile index and call tree node.
Thicket relies on the observation that executions of a code using the same hierarchical index to create a new data dimension for a thicket single object. Users can compose multiple profiles into a thicket design isomorphism problem [15] to find intersections of the call trees. We consider their call trees and solve the graph isomorphism problem [15] to find intersections of the call trees. Thicket uses the call tree to join the profiles giving each profile its own profile index. A user can specify the profile index (e.g., using a study-relevant metadata column such as problem size) or let Thicket generate a unique hash value.

Figure 4 shows a small example of a performance data table containing performance data for five call tree nodes (call tree node names are in the left column; the call tree itself is not shown). In this example, the user has two profiles, one with a problem size of 1,048,576 and the other with a problem size of 4,194,304. Thicket joins the profiles on the call tree nodes, using the problem size as a secondary index, and the resulting thicket object contains two rows (i.e., one row per profile) per call tree node.

3.2 Multi-dimensional Performance Data Composition

Each code execution generates a profile with its metadata (i.e., build settings, execution context) and performance data for a set of metrics collected during the execution.

3.2.1 Composition of a set of profiles into a thicket object. After executing a set of runs, users might be interested in analyzing their data as a set of profiles to quickly inspect which simulation parameters are present in the set or to see which performance metrics were collected. The user calls the thicket constructor with the set of profiles. Thicket uses the call tree to join the profiles giving each profile its own profile index. A user can specify the profile index (e.g., using a study-relevant metadata column such as problem size) or let Thicket generate a unique hash value.

3.2.2 Hierarchical composition of multiple thicket objects. When the user has several thicket objects from profiles generated on different architectures or using different measurement tools, the user may be interested in comparing the performance metrics. The user calls the thicket constructor with the thicket objects as input. A new thicket object is constructed using the (call tree node, profile index) hierarchical index for values that exist in all input thicket objects. The new thicket object has an additional index generated for the columns.

Figure 4 shows an example of composing a CPU thicket object and a GPU thicket object. For each (call tree node, profile index) index present in both input thicket objects, the metrics from the CPU thicket object (i.e., time (exc), Reps, Retiring, and Backend bound) and the metrics from the GPU thicket object (i.e., time (GPU), gpu_compute_memory_throughput, gpu_dram_throughput, and sm_throughput) are composed with an additional index for the columns (i.e., CPU, GPU).

4 ENABLING EXPLORATORY DATA ANALYSIS (EDA)

Thicket is implemented in Python, providing easy integration with other open-source libraries, such as scikit-learn. We leverage Jupyter notebooks [21] to interface with Thicket as they offer an environment for portability and reproducibility of iterative workflows and in-situ visualization [20, 33, 34]. To enable Exploratory Data Analysis (EDA), Thicket provides a wide range of capabilities including manipulation of thicket objects (e.g., filtering, grouping, and querying), generation of statistics and performance models (e.g., built-in aggregated statistics, integration with external libraries), and visualization of data using built-in functions or interactively through Jupyter notebooks.

4.1 Manipulating the Thicket Objects

Thicket enables users to study the different dimensions of thicket objects through the manipulation of their performance data and the metadata components. We implement three processing capabilities to explore the different dimensions: (1) filtering, (2) grouping, and (3) call tree querying. Applying any of the three capabilities to one or more thicket objects creates one or more new thicket objects.

Figure 3: An entity relationship diagram describing the links among the components of a thicket object: (i) performance data, (ii) metadata, and (iii) aggregated statistics. The keys in bold are fixed and link the components of the thicket object. The values are examples of metrics collected during the execution.

Thicket relies on the observation that executions of a code using the same hierarchical index to create a new data dimension for a thicket single object. Users can compose multiple profiles into a thicket design isomorphism problem [15] to find intersections of the call trees. We consider their call trees and solve the graph isomorphism problem [15] to find intersections of the call trees. Thicket uses the call tree to join the profiles giving each profile its own profile index. A user can specify the profile index (e.g., using a study-relevant metadata column such as problem size) or let Thicket generate a unique hash value.

Figure 4 shows a small example of a performance data table containing performance data for five call tree nodes (call tree node names are in the left column; the call tree itself is not shown). In this example, the user has two profiles, one with a problem size of 1,048,576 and the other with a problem size of 4,194,304. Thicket joins the profiles on the call tree nodes, using the problem size as a secondary index, and the resulting thicket object contains two rows (i.e., one row per profile) per call tree node.

3.2 Multi-dimensional Performance Data Composition

Each code execution generates a profile with its metadata (i.e., build settings, execution context) and performance data for a set of metrics selected by the user. Possible metadata attributes include:

- Problem sizes and simulation parameters;
- Time series (i.e., states of the simulation over time);
- Amount of resources for strong or weak scaling studies (e.g., number of processors or threads);
- Types of architectures (e.g., CPU and GPU);
- Executables (e.g., different compilers or optimizations);
- Data collected by different tools/libraries (e.g., Caliper, NCU).

Thicket relies on the observation that executions of a code using different build settings or execution contexts (e.g., running on different hardware architectures or varying the problem sizes) typically yield similar or identical call trees, making the call trees an ideal basis for composing large ensembles of code runs. To compose similar runs, Thicket considers their call trees and solves the graph isomorphism problem [15] to find intersections of the call trees. We design Thicket to allow users to compose the performance data of multiple profiles. Users can compose multiple profiles into a single thicket object, or compose multiple thicket objects with the same hierarchical index to create a new data dimension for analysis.

Figure 3: An entity relationship diagram describing the links among the components of a thicket object: (i) performance data, (ii) metadata, and (iii) aggregated statistics. The keys in bold are fixed and link the components of the thicket object. The values are examples of metrics collected during the execution.
Figure 4: Example of multi-dimensional performance data for studying code performance for two problem sizes. CPU metrics are collected in profiles from the CPU, and GPU metrics are collected in profiles from the GPU. The user constructs one thicket object using the two CPU-generated profiles (one per problem size) and one thicket object using the two GPU-generated profiles. The user then hierarchically composes these thicket objects into a single thicket object with an additional column index (i.e., CPU, GPU).

Figure 5: Example metadata table in Thicket containing four RAJA Performance Suite profiles generated on two clusters.

4.1.1 Filtering. When studying the impact of build settings or execution context, it is not always feasible to aggregate across all loaded profiles, for example, when different problem sizes or compilers are considered. For this reason, Thicket makes it easy to filter the profiles based on specific criteria. After loading profiles into a thicket object, a common starting point is to obtain an overview by consulting the metadata table. The first column of the metadata table is a unique profile index followed by metadata providing additional details about the build settings and execution context. Figure 5 presents an example of a thicket metadata table from four application runs of the RAJA Performance Suite, one of our case studies discussed in more detail in Section 5.1. In our work, the application metadata has been collected with the performance metrics into a profile per run. Example metadata includes the compiler and RAJA versions used to build the application, the cluster and problem size used for the runs, the launch date of the run, and the user who ran the application.

4.1.2 Grouping. When performing a grouping of profiles, a user defines a list of metadata attributes (i.e., columns in the metadata table) to form a new thicket object. Specifically, this capability allows grouping profiles based on their unique values in a column or a unique combination of values in multiple columns. It generates a list of new thicket objects combining the unique values in the columns. Figure 7 shows an example of grouping profiles based on the unique combination of values from the compiler and problem size columns of the metadata table in Figure 5. With two unique values for the compiler (i.e., clang-9.0.0 and xlc-16.1.1.12) and two unique values for the problem size (i.e., 1,048,576, 4,194,304) in the metadata table in Figure 5, a total of four new thicket objects are created.

4.1.3 Querying the Call Tree. Thicket enables users to extract a set of paths from a call tree and its corresponding performance data by leveraging the Call Path Query Language in Hatchet [27]. To extract a call path, the user first creates a query describing the properties that a path must have. A query is defined as a sequence of query nodes, whereas a query node comprises a quantifier and a predicate. The quantifier defines how many actual nodes in a call tree path to match to a query node. The predicate defines what

![Figure 6: The resulting metadata table after a filter operation on the compiler column from Figure 5.](image-url)
Figure 7: The resulting metadata table from Figure 5 after the profiles have been grouped based on unique combinations of values from the compiler and problem size columns.

conditions must be satisfied for an actual node to match a query node. After creating a query, a user applies it to a call tree and its corresponding performance data, finding all the paths in the call tree that match the properties described by the query. The filtered call tree nodes are returned as a new thicket object. The example shown in Figure 8 queries for a call path containing nodes with .block_128 in the name.

4.2 Statistics and Modeling

Thicket enables users to make decisions and predictions based on their performance data by providing capabilities for computing aggregated statistics on their performance data and returning results in a form suitable for a wide range of advanced analysis and visualizations.

4.2.1 Built-in Aggregated Statistics. Aggregated statistics that summarize the distribution of the data is useful for quick introspection. Thicket generates an aggregated statistics table to store such information using extendable order-reduction mechanisms of common aggregation and statistics functions. The built-in functions in Thicket include variance, standard deviation, maximum and minimum, percentiles, correlation coefficient, mean, and median. Users can also filter the aggregated statistics table using a similar mechanism to the metadata table described in Section 4.1.1. The table at the top of Figure 9 shows an example in which a user has computed the standard deviation of the Retiring, Backend bound, and time (exc) columns that are in the corresponding performance data table.

4.2.2 Leveraging External Libraries: Scikit-Learn. In addition to supporting traditional performance analyses such as top-down, Thicket enables users to easily leverage their performance data in more general data science techniques. By building Thicket in Python, the performance data, metadata, and aggregated statistics can quickly be passed to other common Python data science

Figure 8: The call tree before (top) and after (bottom) applying the query language on a thicket object to find leaf nodes with the name ending in block_128. The call trees show the exclusive time for each node.
Figure 10: Results of K-Means clustering for the retiring and backend bound top-down metrics concerning speedup (relative to -O0) for "Stream" kernels from the RAJA Performance Suite. The frontend bound and bad speculation top-down metrics are omitted because they each represented less than 10% of the total execution time for all kernels.

Figure 11: Extra-P models of a MARBL function on RZTopaz (i.e., CTS) and AWS ParallelCluster. Red dots represent performance measurements of the M_solver->Mult function in MARBL. The blue line is a scaling function computed by Extra-P from the performance measurements.
We added a convenient high-level interface in Thicket (top), while the similar structure of the generated scalability models vague. It also helps communicate findings to others in the community. We provide static built-in visualizations across multiple facets, especially when the analysis questions are data, such as trends, outliers, or possible bad data, simultaneously.

Figure 11 shows measured times and the automatically generated scalability bottleneck and exclusive time. We dive deeper into exploring the outliers by looking at the the histogram plots for the nodes of interest (see insets).

4.2.3 Performance Modeling using Extra-P. Performance models are valuable tools for studying scalability expectations and limitations of parallel programs and algorithms. Thicket can generate performance models of ensemble data automatically using the Extra-P [2] library. Given an ensemble of performance profiles covering one or more modeling parameters as input, Extra-P derives an analytical performance model (i.e., a function) that best matches the input data [11]. A common use case is a scalability model where we use an ensemble of performance measurements taken at small numbers of MPI ranks (e.g., 1, 4, 8, 16, and 64 ranks) to generate an analytical scaling function which allows us to extrapolate performance at higher rank counts (e.g., 512 ranks). By generating such performance models in bulk for an entire set of code regions, developers can easily identify regions which might become scalability bottlenecks. Thicket is an ideal entry point for modeling studies with Extra-P since the thicket object holds all of the required input data (modeling parameters like number of MPI ranks, together with the associated performance data) in one place. We added a convenient high-level interface in Thicket that gives developers easy access to Extra-P’s modeling functionality.

Figure 11 shows the selected solver function in MARBL is faster on AWS ParallelCluster (bottom) than RZTopaz (top), while the similar structure of the generated scalability models suggests that the function scales similarly well on both systems.

4.3 Data Visualization

Data visualization is a key element of the EDA process. It leverages human visual perception to identify points of interest in the data, such as trends, outliers, or possible bad data, simultaneously across multiple facets, especially when the analysis questions are vague. It also helps communicate findings to others in the community. We provide static built-in visualizations in Thicket for quick overviews and more advanced, interactive visualizations through Jupyter notebook templates.

4.3.1 Built-In Visualization Functions. Thicket has a growing collection of commonly needed visualizations to quickly gain an overview of the data. Two visualization capabilities in Thicket are a heatmap and histogram to aid with the identification of outliers or determining relationships between variables by visualizing their value distributions.

Figure 12 shows a heatmap of the Retiring_std, Backend bound_std, and time (exc)_std metrics that are computed in the aggregated statistics shown in the top table of Figure 9. We show the heatmaps on two different sets of axes due to the scale of the metric values. The dark color indicates a node and metric pair with a high standard deviation. The histogram plots are used to show detailed distribution of the data for these nodes of interest, that is Polybench_GESUMMV and Lcals_HYDRO_1D. The visualizations rely on the matplotlib and seaborn libraries and support passing most of their standard parameters as additional keyword arguments to allow for user-specified customization.

4.3.2 Interactive Visualization through Jupyter Templates. While our suite of built-in visualizations enable quick analysis of the thicket data object, more in-depth visual analysis is supported by the multi-linked, interactive visualizations available through our Jupyter Notebook templates. These provide a graphical interface for directly manipulating the data and can be used with the built-in programmatic manipulation available through our Python API. Specifically, we provide two main linked visualizations. The tree + table visualization pairs the thicket call tree with a collection of metric charts, ordered to show trends in the data, enabling comparison across the whole ensemble of performance runs. The paired parallel coordinate and scatter plot visualization shows correlations among the numerous variables in the dataset. Analysts can interactively choose variables to focus on and re-order the visualization by variables of interest. We demonstrate these visualizations as part of our case study, with examples shown in Figure 14 and Figure 18.

5 CASE STUDIES

In this section, we introduce the applications and the environmental setup (including build settings and execution context) used to showcase the EDA capabilities of Thicket. We study the performance of two applications, the open-source RAJA Performance Suite [13, 25] on CPUs and GPUs, and MARBL, a multi-physics simulation on AWS and HPC architectures.

5.1 RAJA Performance Suite

The RAJA Performance Suite [13, 25] is designed to explore performance of loop-based computational kernels found in HPC applications. RAJA [6, 24] is a C++ library providing software abstraction to enable architecture portability of HPC applications. RAJA has two main goals: 1) enable application portability while minimizing changes to existing algorithms and programming styles and 2) achieve performance comparable to directly using, for example, OpenMP or CUDA.

We run the RAJA Performance Suite on two different node architectures in Quartz and Lassen. We used the OpenMP and CUDA...
variants on Quartz and Lassen, respectively. The details of each of
the node architectures are given below. Figure 13 summarizes the
RAJA Performance Suite experiments we performed and used in
the studies for this paper.

Quartz. Each compute node has 36 (2x18) Intel Xeon E5-2695 v4
hyper-threaded cores and 128GB of RAM. Our software develop-
ment environment for RAJA Performance Suite included both Clang
9.0.0 and GCC 8.3.1 to create two executables. We ran the OpenMP
variant of RAJA Performance Suite on a single node with 36 threads.

Lassen. Each compute node features two Power9 CPUs and four
Volta GPUs connected via three NVLINK2 connections. Each GPU
has 16GB of memory and the node has 256GB of system memory.
Our software development environment for the RAJA Performance
Suite included CUDA 11.1.1 for the GPU compiler and XL 16.1.1.12
for the CPU compiler.

5.1.1 Top-down visualization and analysis on Intel CPUs. Top-down
analysis [39] is a quick and practical performance analysis method
for out-of-order processors, using designated hardware perfor-
mance counters in a structured hierarchical approach to identify
dominant bottlenecks. The analysis breaks down the observed CPU
pipeline utilization into four broad categories: retiring, frontend
bound, backend bound, and bad speculation. Each category is hierar-
chically divided into more detailed sub-categories to narrow down
specific performance bottlenecks, however, in this work we only
focus on the top-level categories. Yasin [39] describes the detailed
top-down model and its derivation from specific performance coun-
ters for Intel Core architecture processors. Caliper has a built-in
module to collect the required performance counters and compute
the top-down metrics for annotated code regions, which we used
to collect top-down metrics for the RAJA Performance Suite.

Figure 14 shows a purpose-built top-down analysis visualization
in Thicket. This visualization leverages a tree + table para-
digm to relate measured performance data back to the nodes from
which they came. The top-down data is rendered as a series of
stacked bar charts, color coded by the associated top-down metric.
The metrics in a single bar are percentages of boundedness and
accordingly add up to 1. Each bar represents a single profile in
our ensemble and bars are grouped and sorted by an independent
variable of interest. Through this grouping and vertical alignment
with associated nodes, users can quickly see how the bounded-
ness of their program scales for particular nodes of interest. This
can inform users about function-level optimization opportunities.

In this example, we have 10 individual profiles for each config-
uration, and we group the bars by the RAJA Performance Suite
profile size ranging from 1,048,576 to 8,388,608. We identify that
the Apps_VOL3D kernel is more compute-bound than the other ker-
nels, as it has a higher percentage of retiring instructions. The
Apps_VOL3D kernel is 90% backend bound, while

Lcals_HYDRO_1D kernel is 90% backend bound, while

Stream_DOT kernel is heavily backend bound as the problem size increases. The Lcals_HYDRO_1D and Stream_DOT kernels are similarly backend bound, however they become more
backend bound as the problem size scales, indicating data sat-
uration.

5.1.2 Multi-Architecture Analysis. NVIDIA's Nsight Compute (NCU)
is a vendor-specific profiling tool to gather GPU performance met-
rics, such as kernel time and memory throughput. NCU profiling is
capable of producing hundreds of detailed metrics per GPU kernel,
which we append to the metrics from our CPU profiles to enhance
our analyses and provide more insights into GPU performance. We
leverage Caliper annotations in RAJA Performance Suite to enable
NCU to profile these kernels.

Figure 15 is an example of multi-dimensional performance data
in a thicket, demonstrating how Thicket composes metrics across
two different architectures — CPU and GPU — into a centralized,
digestible format. The table is comprised of four different profiles
that have been composed horizontally. The table also composes
the profile data side-by-side: (1) basic CPU metrics under the CPU
header, (2) top-down metrics from caliper's top-down analysis mod-
ule under the CPU top-down header, (3) time on the GPU under
the GPU header, and (4) GPU-specific metrics from NCU under the
GPU Nsight Compute header. Due to space constraints, we include
a small set of kernels and performance metrics of interest for our
analysis.

We use the composed performance data to compute the GPU to
CPU speedup of two RAJA Performance Suite kernels, Apps_VOL3D
and Lcals_HYDRO_1D, and store these values in a new speedup col-
umn under the Derived header in Figure 15. We focus on these two
kernels because they are different in their execution behaviors. The
speedup of Lcals_HYDRO_1D is not as big as that of Apps_VOL3D,
and we use the composed performance data to look into the avail-
able top-down and GPU metrics to understand why this occurred.
The Lcals_HYDRO_1D kernel is 90% backend bound, while Apps_VOL3D
is more evenly split between backend bound and retiring at 54% and
37%, respectively. The higher retiring percentage in the Apps_VOL3D
kernel indicates that it is more compute-heavy and has potential
for leveraging higher gains on the GPU. Thicket enabled more

<table>
<thead>
<tr>
<th>cluster</th>
<th>systype</th>
<th>build</th>
<th>problem size</th>
<th>compiler</th>
<th>compiler optimizations</th>
<th>omp num threads</th>
<th>cuda compiler</th>
<th>block sizes</th>
<th>RAJA variant</th>
<th>#profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>quartz</td>
<td>toss_3_x86_64_ib</td>
<td>[1048576, 2097152, 4194304, 8388608]</td>
<td>clang++-9.0.0 [00, -01, -02, -03]</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>Sequential</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>quartz</td>
<td>toss_3_x86_64_ib</td>
<td>[1048576, 2097152, 4194304, 8388608]</td>
<td>g++-8.3.1 [00, -01, -02, -03]</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>Sequential</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>quartz</td>
<td>toss_3_x86_64_ib</td>
<td>[1048576, 2097152, 4194304, 8388608]</td>
<td>clang++-9.0.0</td>
<td>-00</td>
<td>72</td>
<td>N/A</td>
<td>N/A</td>
<td>OpenMP</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>quartz</td>
<td>toss_3_x86_64_ib</td>
<td>[1048576, 2097152, 4194304, 8388608]</td>
<td>g++-8.3.1</td>
<td>-00</td>
<td>72</td>
<td>N/A</td>
<td>N/A</td>
<td>OpenMP</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>lassen</td>
<td>blues3_3_ppc64le_ib_p9</td>
<td>[1048576, 2097152, 4194304, 8388608]</td>
<td>XL++-7.1-16.1.1.12</td>
<td>-00</td>
<td>1</td>
<td>nvcc-11.2.152 [128, 256, 512, 1024]</td>
<td>CUDA</td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: Summarized view of the RAJA Performance Suite configurations used in this paper.
detailed performance analysis by providing a holistic performance data table containing a variety of metrics in a centralized structure.

5.2 MARBL

MARBL is a next-generation, massively-parallel, GPU-enabled multiphysics code under development at Lawrence Livermore National Laboratory. MARBL supports 1D, 2D, and 3D unstructured meshes with high-order numerical discretizations, Arbitrary Lagrangian-Eulerian (ALE) solution methods, and adaptive mesh refinement (AMR). MARBL was designed to simulate high-energy density physics (HEDP) experiments including high-explosive, magnetic, or laser driven experiments [5].

The MARBL team is collaborating with Amazon Web Services (AWS) to demonstrate the ability to run advanced multi-physics codes on cloud resources at scale. To that end, we reproduce a set of scaling studies in the AWS cloud that was originally performed on commodity clusters at LLNL and Sandia National Laboratory (SNL) [36]. In our study, we are focusing on a modestly-sized, 3D triple-point shock interaction benchmark problem, and compare results from the off-the-shelf cluster (RZTopaz) with an AWS cloud instance (AWS ParallelCluster). The details of each system are given below. Figure 16 summarizes the MARBL experiments we used in this study. We study six different configurations with 1 through 32 compute nodes and 36 to 1,152 MPI ranks. We run MARBL with 36 MPI ranks per node on both systems.

RZTopaz. RZTopaz is an Intel Xeon system with 36 (2x18) E5-2695 v4 cores and 128GB of RAM per node and Intel Omnipath interconnect.

AWS ParallelCluster. AWS ParallelCluster is an AWS instance with C5n.18xlarge compute nodes using the AWS Elastic Fabric Adapter (EFA) interconnect with placement groups. Each compute node has 36 (2x18) Intel Xeon Platinum 8124M cores and 192GB RAM.

5.2.1 Scaling Analysis. In our first case study with MARBL, we use Thicket’s statistical capability to perform a strong scaling study on this ensemble. Strong scaling is a common analysis for studying the performance of applications. It uses a fixed global number of unknowns and varies the number of compute nodes. Note that ideal scaling has a slope of -1.

Figure 17 shows the results of a strong scaling study of a 3D triple-point problem in MARBL. For this strong scaling study, we used up to 2,304 MPI ranks across 64 compute nodes on each cluster. Each data point in this study is the average of five MARBL runs. Shaded areas show the deviation from the average. We used Thicket to load in the ensemble performance data from our strong scaling study, and then used matplotlib to quickly generate a line plot. Our line plot shows that both Intel MPI and OpenMPI scale well up to 16 nodes on both RZTopaz and AWS ParallelCluster.

5.2.2 Interactive Visualization. Thicket provides custom notebook-embedded visualizations to explore relationships across multiple parameters in the metadata and performance data spaces of our performance ensembles. Figure 18 shows custom metadata visualizations with the cluster name, number of MPI ranks, walltime, and maximum elements per rank as variables. Like our top-down visualization, it was designed to be embedded in Jupyter notebooks to support the script-heavy, exploratory analysis workflows enabled by Thicket. This visualization provides two scatterplots and a parallel coordinate plot (PCP) to give users many different perspectives on their data.

The left scatterplot plots metadata values on the x-axis against a per-function measured metric on the y-axis. This plot enables users to relate independent variables stored as metadata against dependent variables stored as performance data. Specifically, the scatterplot here shows how the number of elements processed affects the measured runtime for the timeStepLoop function of our application.

The right scatterplot enables users to view relationships between two variables in the performance data. This plot aids in identifying
Figure 15: Multi-dimensional performance data of two kernels in RAJA Performance Suite run on a CPU (e.g., Quartz) and a GPU (e.g., Lassen) architecture. The data is assembled from multiple profiles. The time (exc), Bytes/Rep, and Flops/Rep columns are from a CPU run with no profiling overhead; the Retiring and Backend bound time columns are from a CPU run with top-down analysis, the time (gpu) is from a GPU run, and gpu__compute_memory_throughput, gpu__dram_throughput, sm__throughput, and sm__warps_active are collected by Nsight Compute. The speedup column is derived from $\frac{CPU, time(exc)}{GPU, time(gpu)}$.

Figure 16: Summarized view of MARBL configurations used in this paper.

Figure 17: Strong scaling results for MARBL 3D triple-point shock interaction.

interesting clusters of profiles and outliers based on multiple variables. It further enables users to relate interesting subsets of data back to the metadata via the PCP.

The PCP shows the relationships between many metadata variables stored in the thicket object. Each axis corresponds to a column in the metadata table and each colored line traces the metadata values associated with a single profile. The crayon icons to the left of each axis enable users to color lines and points based on the associated metadata. In our example shown in Figure 18, all the data is colored according to which architecture (arch) it was run on: RZTopaz or AWS ParallelCluster.

In Figure 18 we see a case study of how these components can provide insight into larger trends which occur in ensembles of performance profiles. By coloring profiles according to their architecture, a clear pattern emerges both in the scatter plots and the PCP. As the problem size increases, MARBL performs better on the AWS ParallelCluster than RZTopaz. This holds for both the individual function being examined here (timeStepLoop) as evidenced by the scatterplots as well as the overall runtime of the program evidenced by the walltime axis in the PCP.

This visualization also provides an easy way to verify the scalability of applications at a glance by showing trends between the number of MPI ranks and overall runtime. We see that the MARBL software scales well to increased parallelism because there is a lot of cross-over between the the walltime and mpi.world.size axes. This kind of criss-crossing structure generally indicates inverse correlations in PCPs. In this specific case, more MPI ranks are associated with lower runtimes, with the AWS ParallelCluster being consistently lower than RZTopaz.

6 RELATED WORK

There is a variety of community-driven and commercial HPC performance analysis tools covering a wide range of measurement methodologies and use cases. Comprehensive performance analysis frameworks like HPCToolkit [4, 28], Score-P [23], and TAU [35] collect detailed per-thread execution profiles or traces for in-depth analyses. These tools focus on recording performance data and conducting interactive analyses of individual program executions, but they typically do not provide specific capabilities for analyzing ensembles of runs.

LLNL’s ubiquitous performance analysis approach [8] provides a workflow for automatically collecting performance profiles and program metadata for a set of program executions, e.g. as part of a nightly testing setup. A web interface with specialized graphical tools for filtering and comparing large sets of runs allows users to explore the collected performance data. Thicket users can utilize elements of the ubiquitous performance analysis system, in particular Caliper [9] and Adiak [1] for recording performance profiles and run metadata, respectively. However, instead of a limited set of pre-built visualizations, Thicket provides a fully programmable framework for analyzing ensemble performance data.

Prior work in ensemble analysis exists with performance data management tools such as PerfDMF [16] and PerfTrack [18, 19, 22], which are designed to analyze and compare performance data from
different runs of an application. PerfDMF provides robust, interoperable components for performance data management. It serves as the SQL-based storage backend for PerfExplorer2 [17], a data mining framework with capabilities to correlate performance data and metadata for various types of ensemble analyses, such as scaling studies. The PerfTrack framework also uses a SQL database to store profile data from multiple experiments, and provides interfaces to the data store, a GUI for interactive analysis, and modules to automatically collect experiment metadata. Our Thicket framework builds upon many of the elements developed in these performance data management tools, but provides a programmable interface and integration with modern Python data analysis tools to enable flexible and customizable ensemble studies.

Hatchet [7, 10] is an open-source Python library capable of reading in profiling output from several tools, such as HPCToolkit and Caliper, and enables programmatic analyses on hierarchical performance profiles. Its functionality is limited to performing analysis on one or two profiles at a time with such use cases as computing load imbalance across nodes in a single run, or computing the speedup of a single core to many cores. Thicket uses Hatchet’s readers for loading in a single profile at a time, and provides additional tools for Exploratory Data Analysis of multi-run performance experiments.

An alternative to using Pandas dataframes as Thicket’s internal representation is Xarray [14]. Xarray offers a rich programming interface to interact with labeled multi-dimensional data that is leveraging Dask and Pandas internally. While Xarray already implements various aggregation and statistics operations it primarily targets non-sparse data leading to undesired data duplication for the performance data we would like to represent in Thicket.

The custom built visualizations for Thicket were informed by prior work in the visualization domain. Several works [30, 37, 38] describe the use of parallel coordinate plots for visualizing ensemble data produced from scientific simulations. We adapt their insights to the performance analysis domain. The top-down visualization was informed by Juniper [29] and Lineup [12]. Lineup explores techniques for organizing bar charts while Juniper introduces a tree+table design for multivariate visualization. A recent work on the intentional design of notebook embedded visualizations [34] informed our development and design of the top-down analysis visualization and the metadata visualization.

7 CONCLUSIONS

We present Thicket, an open-source Python toolkit for understanding and optimizing simulation performance on supercomputers. It provides an interface for interacting with the multi-dimensional, multi-scale, multi-architecture, and multi-tool performance data through a modular structure. Thicket enables Exploratory Data Analysis (EDA) of performance data, including interactive visualization, performance modeling, and data science techniques. We explore two case studies to highlight the capabilities of our toolkit, first with the open-source RAJA Performance Suite run on CPU and GPU clusters, and second with a large multi-physics code known as MARBL run on an HPC cluster and an AWS instance.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-844468). Additionally, Connor Scully-Allison and Katherine E. Isaacs were supported by the Department of Energy under DE-SC0022044.

The authors thank Matt Legendre, Magnus Strengert, Sergei Shudler, Holger Jones, and Todd Gamblin for their feedback on this work.

REFERENCES
