Enabling Lightweight Performance Analysis of Complex Scientific Workflows with PerfFlowAspect

Aliza Lisan
University of Oregon
Eugene, OR, USA
alisan@uoregon.edu

Tapasya Patki
Lawrence Livermore National Laboratory
Livermore, CA, USA
patki1@llnl.gov

Stephanie Brink
Lawrence Livermore National Laboratory
Livermore, CA, USA
brink2@llnl.gov

Konstantinos Parasyris
Lawrence Livermore National Laboratory
Livermore, CA, USA
koparasy@gmail.com

Brian Gunnarson
Lawrence Livermore National Laboratory
Livermore, OR, USA
gunnarson1@llnl.gov

Giorgis Georgakoudis
Lawrence Livermore National Laboratory
Livermore, CA, USA
georgakoudis1@llnl.gov

Hank Childs
University of Oregon
Eugene, OR, USA
hank@uoregon.edu

Abstract

Scientific workflows are becoming increasingly complex, including utilization of multiple binaries, multiple clusters, and wide ranges of input parameters. This complexity will likely increase with the emergence of Artificial Intelligence and Machine Learning (AI/ML) simulations as well as the convergence of High-Performance Computing (HPC) and Cloud technologies. Existing performance tools target traditional bulk-synchronous parallel single-binary applications. Such tools do not integrate easily with upcoming complex workflows, for reasons including steep user-level learning curves, source code readability issues, and lack of support for composability and native visualization of traces.

We introduce PerfFlowAspect, a lightweight, open-source, and aspect-oriented performance analysis tool for complex scientific workflows. PerfFlowAspect is easy-to-use and can provide both coarse and fine-grained performance results ranging from traditional to complex multi-cluster, multi-binary workflows. We demonstrate its usability with two uses cases: (1) Merlin ICECap: an ML-workflow based on a scalable framework designed for multi-cluster simulations, and (2) Autonomous Multi-Scale (AMS): an application to integrate ML surrogate models in HPC physics simulations. Our findings demonstrate that PerfFlowAspect is effective in supporting complex, multi-cluster and multi-binary workflows in a user-friendly manner with minimal overhead (0.9% on average).

SSDBM 2025, Columbus, OH, USA

ACM ISBN 979-8-4007-1462-7/25/06
https://doi.org/10.1145/3733723.3733734

CCS Concepts

• **Computing methodologies** → **Massively parallel and high-performance simulations**; **Simulation evaluation**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; *Distributed architectures.*

Keywords

HPC, Scientific Workflows, Performance Analysis

ACM Reference Format:
Aliza Lisan, Tapasya Patki, Stephanie Brink, Konstantinos Parasyris, Brian Gunnarson, Giorgis Georgakoudis, and Hank Childs. 2025. Enabling Lightweight Performance Analysis of Complex Scientific Workflows with PerfFlowAspect. In *The International Conference on Scalable Scientific Data Management 2025 (SSDBM 2025), June 23–25, 2025, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3733723.3733734

1 Introduction

Large-scale simulations on high-performance computing (HPC) systems are essential in many scientific domains. As HPC systems are often very costly, it is critical that the simulation codes operate as efficiently as possible. This can be difficult to achieve due to sophisticated underlying hardware architectures and the difficulty in optimizing simulation codes at scale. Specialized tools, called performance analysis tools, help with identifying performance bottlenecks and inform scientists how well their simulations make use of HPC resources. These tools commonly operate by instrumenting simulation codes to generate performance data as they run. The data is analyzed afterwards, in order to present scientists with results in the form of tables, traces, graphs, and visualizations.

Traditionally, these performance analysis tools have operated on a single simulation code, which is comprised of a single binary and is typically a bulk-synchronous parallel program (for example, one that uses the Message-Passing Interface, or MPI). Modern HPC simulation codes, however, are diverging significantly from

these traditional approaches. These modern codes now explore a wide range of parameters, utilize distributed data setups, and leverage advanced statistical or AI/ML models. They accomplish this by defining a *scientific workflow* [12–14, 23], either by utilizing a single-binary code with diverse parameters for optimization (ensembles), or by connecting distinct components (multiple binaries) with specified data dependencies and coordination points. In some cases, these components can execute across different HPC clusters (and potential cloud setups), resulting in multi-binary and multi-cluster workflows.

The premise of this work is that performance analysis tools should adopt different approaches for these upcoming complex workflows. In a traditional setting, a simulation code may be instrumented (with source-code annotations) with hundreds of pairwise region markers (begin and end markers) in order to pinpoint fine-grained performance issues. For scientific workflows, however, it is desirable to limit the amount of instrumentation, rather than "pollute" many code bases. Workflow code bases are inherently more complex than single-binary applications, and as a result, preserving code readability, avoiding instrumentation errors, and reducing the burden on the user become key requirements.

The presence of multiple binaries and multiple HPC systems also implies that performance analysis tools will likely face more diverse technologies (underlying architectures, programming languages, etc.). For example, a workflow that combines physics simulations and machine learning may need to deal with C/C++, CUDA, and Python. Further, the sorts of analyses performed may be more coarse-grained, in terms of understanding where time is being spent in the overall workflow, as opposed to identifying a loop or an `if-else` block that is a bottleneck in one of the function calls in one of the constituent programs. Finally, an ideal tool should be able to compose traces from multiple binaries and present these to the user in a format that is easy to understand, analyze and visualize.

This paper introduces PerfFlowAspect [33, 37], a light-weight performance analysis tool for complex workflows. The key contributions of this work are:

- An approach for collecting performance data from multi-binary and multi-cluster scientific workflows running on heterogeneous HPC infrastructures, with minimal overhead (0.9% on average).
- Demonstration of PerfFlowAspect's effectiveness using two distinct, complex, and multi-language workflows: (1) Merlin ICECap: an ML-workflow based on a scalable framework designed for multi-cluster simulations, and (2) Autonomous Multi-Scale (AMS): an application to integrate ML surrogate models in HPC physics simulations.

PerfFlowAspect supports C/C++, CUDA, and Python programs, is user-friendly and preserves code readability by requiring minimal instrumentation code. It generates trace files with performance data (timing data, processor, and memory usage, etc.), which can be collated and visualized using open-source tools like Perfetto [20] and Hatchet [5, 9].

The rest of the paper is organized as follows. Section 2 discusses the limitations of existing performance analysis tools and lays down the requirements that are necessary for upcoming scientific workflows. Section 3 presents the design and implementation of PerfFlowAspect. Section 4 demonstrates how PerfFlowAspect can be utilized for the analysis of Merlin ICECap, a multi-binary and multi-cluster workflow. Section 5 shows how PerfFlowAspect can flexibly perform both coarse and fine grained analysis of a complex single-binary application, AMS. Section 6 presents overhead results and Section 7 further discuss benefits and limitations. Section 8 concludes the paper.

2 Background and Motivation

2.1 Upcoming Complex Scientific Workflows

The exascale era and the emergence of AI/ML techniques has changed the nature of scientific simulation codes. As opposed to the traditional bulk synchronous parallel applications, emerging scientific workflows now rely on distinct connected components and explore a wide range of parameters. When multiple coordinated components are involved in a workflow, each individual component is typically an independent program (binary) that performs a designated task, such as an MPI computation program, a specialized GPU kernel, a data staging process, or a Python script. While these components often depend on one another, they need not execute on the same HPC cluster, and could even execute across facilities or in converged HPC and cloud setups. Data can be passed between such components by using a shared file system or with message queues [3, 54, 55].

Further, these modern workflows are expected to have some components that may leverage specialized hardware such as GPUs or AI accelerators (e.g. Cerebras or SambaNova architectures). Understanding the performance bottlenecks as well as the resource utilization in such cases is crucial for the efficient use of such expensive hardware. Some examples of such complex workflows include Multiscale Machine-Learned Modeling Infrastructure (MuMMI) [15], American Heart Association Molecule Screening (AHA MoleS) [3], Autonomous MultiScale Library (AMS) [46], ICECap [49], CyberShake [21], Galactic [51], and GUIDE [17].

2.2 Existing HPC Performance Analysis Tools

Several performance analysis tools have been developed for HPC applications over the years. These performance tools typically fall into two key categories: (1) tools based on *sampling*, where event-based or time-based sampling is used to collect measurements, and, (2) tools based on *annotations*, where the application source code is instrumented with measurement indicators. Sampling-based tools (such as HPCToolkit [1, 42], Open|SpeedShop [53], or `mpiprof`) provide the convenience of profiling applications and binaries without having to make any changes to the source code. This is especially helpful when the performance profiling team does not have the scientific domain expertise to instrument the source code, or when modifying the source code could lead to significant build challenges or errors, or when the source code itself is unavailable due to access control concerns. These tools, however, run into the limitation that they collect data on the entire application as opposed to just its critical path. This can generate a lot more data than needed, making the analysis of the application's regions of interest difficult. Another limitation of such tools is related to the frequency of sampling,

which is typically constant through the entire application's execution. Determining the optimal sampling frequency is challenging: a high sampling rate can result in high logging overhead, collection of unneeded data and input-output (I/O) issues, while a low sampling rate can miss out on critical information and application behavior. As many applications tend to have phases (such as data staging, communication or compute phases), the optimal sampling frequency varies and cannot be customized at the level of the entire application.

As a result, annotation-based tools, such as TAU [52], Score-P [26], Timemory [40], and Caliper [7, 8], can be more advantageous for HPC users, as they are more customizable and have lower overheads. They allow users to focus on and profile the regions of interest on their application's critical path, making it possible to capture the behavior of different phases flexibly. Perfevent [31], a header only C++ library, also provides region-level data by simplifying interaction with Linux's performance monitoring capabilities. However, these tools and libraries do require domain expertise and access to the source code when it comes to the application being profiled. Online and continuous monitoring frameworks have also been proposed in the past [2, 16, 41], but these focus more on the system-level telemetry as opposed to HPC application critical paths. A comparative study of HPC performance tools can be found in [11], and discussion on optimizing them from an I/O perspective can be found in [18].

2.3 Tools Referenced in this Paper

In this paper, we make use of a few of the aforementioned tools, which we describe here. Caliper is a performance analysis tool that provides tracing and profiling services for traditional single-binary HPC applications. Users can annotate source code or use configuration files to utilize these services. Caliper uses JSON or custom file formats as output [7, 8].

Hatchet [5, 9] is a Python-based performance data analysis tool. It stores numerical and categorical information from profiling data along with the caller-callee relationships, allowing for pairwise comparison for scalability analysis.

Perfetto is an open-source stack for instrumentation and performance analysis [20]. It provides an interactive trace visualizer for querying hours-long traces, and supports multiple tracing formats, including the Chrome Tracing Format [19].

2.4 Limitations of Existing Tools

Existing tools discussed above lack several key features that are needed for scientific workflows. Several of them pollute the application's code base by requiring region begin and end markers, reducing the code's readability and making the annotations error prone. Some of them do not support analysis in multiple programming languages. Further, the trace formats are often not human-readable or easily visualizable with commonly available tools.

As an example, we present the annotation APIs from two widely used tools, TAU and Caliper. Listing 1 shows TAU's Python instrumentation API [44], which is necessary when trying to not automatically collect data from *all* functions in the user's code base. This example shows two timers, `x` and `y`, that track two areas of interest. Each timer needs to be managed separately by the user. Listing 2 shows the annotation of a C function `foo` with Caliper [38]. Here, each exit point has to be marked with `CALI_MARK_FUNCTION_END`. While both TAU and Caliper are powerful tools in terms of fine-grained performance analysis, these examples highlight how tedious and error-prone instrumenting regions of interest with these tools can be, especially with large code bases spanning different programming languages and multiple binaries. While annotations for C++-only code are simpler, the common usage pattern is with pair-wise begin-end markers. Debugging warnings and errors arising from annotation mismatches can be challenging. These examples also depict how such annotations can "pollute" the user's code base and greatly limit readability.

```
1   import pytau
2   from time import sleep
3
4   x = pytau.profileTimer("A Sleep for excl 5 secs")
5   y = pytau.profileTimer("B Sleep for excl 2 secs")
6   pytau.start(x)
7   print "Sleeping for 5 secs ..."
8   sleep(5)
9   pytau.start(y)
10  print "Sleeping for 2 secs ..."
11  sleep(2)
12  pytau.stop(y)
13  pytau.dbDump()
14  pytau.stop(x)
```

Listing 1: Example of manual instrumentation of Python code with TAU.

```
1   #include <caliper/cali.h>
2
3   int foo(int r){
4     CALI_MARK_FUNCTION_BEGIN;  /* Exports "function=foo". */
5     if (r > 0) {
6      CALI_MARK_FUNCTION_END; /* Mark each exit point.*/
7      return r-1;
8     }
9     CALI_MARK_FUNCTION_END;   /* Mark each exit point.*/
10    return r;
11  }
```

Listing 2: Example of manual instrumentation of C code with Caliper.

Caliper generates *.cali* files for the annotated regions in the code. TAU generates trace files in `OTF2`, `.trc`, or `.edf` formats. None of these file formats are readable or understandable by domain scientists or users. Any errors or inconsistencies in these files are thus challenging to detect with ease. The content in the *.cali* files needs to be visualized using a tool such as Hatchet. Caliper has in-built tools such as `cali-query` with different options to process *.cali* files and print them on the console in a relatively human-readable format. This output is not interactive and often requires minimizing text on the terminal to make it understandable, especially for complex applications. TAU's `OTF2` can be read by Vampir [25] or C and Python `OTF2` libraries, while the other two formats can be read by Jumpshot [56]. `.cali` and TAU trace files are also not composable across different binaries or programs. For example, Listing 3 shows a snapshot of a '.cali' file for the listing shown in Listing 2, which had 84 total lines.

```
1   __rec=node,id=12,attr=10,data=64,parent=3
2   __rec=node,id=13,attr=8,data=attribute.alias,parent=12
3   __rec=node,id=14,attr=8,data=attribute.unit,parent=12
4   __rec=node,id=45,attr=14,data=sec,parent=5
```

Listing 3: Few lines of the *.cali* file for the example in Listing 2

Table 1: Aspect-Oriented Concepts in Performance Analysis.

Concept	Definition	As Applied to Perf. Monitoring
Advice	Type of action	Measuring duration, memory usage, CPU usage, etc.
Join Points	*Candidate* locations where advice could be applied	Every line of code or every function
Pointcuts	Locations where advice is applied	Code locations that are instrumented with performance monitoring
Aspect	Combination of advice and pointcut	Applying performance monitoring code at a code location
Weaving	The mechanism to realize an advice at a pointcut	The mechanism to monitor performance for instrumented code

2.5 Requirements for Scientific Workflow Performance Analysis

These limitations illuminate the desired features of an effective performance analysis tool for scientific workflows below. First, considering that workflows consisting of multiple programs have (collectively) multiple source code components, the tool should preserve the readability of the complex code base, should be minimally intrusive in terms of number of annotations, and should also minimize introducing any annotation related errors. It should be user-friendly without a steep learning curve, and preferably have a human-readable trace file format that allows users to easily understand and validate the data. Second, given the emergence of complex AI/ML workflows that rely on both C/C++ and Python interactions, the tool should be able to support instrumentation and measurement across multiple languages with ease. Third, a tool that profiles scientific workflows effectively needs to support disparate data sources and generate trace files that are easily composable across different components (multiple binaries) and across the heterogeneous systems (multiple HPC clusters). Fourth, given the sheer amount of data that a workflow can generate, the tool should be able to provide both a coarse-grained, high-level analysis as well as a finer-grained analysis with hardware performance counters if and when desired. Finally, the tool should be able to generate trace files that can be visualized easily with existing and commonly-available visualization tools, such as Perfetto.

3 PerfFlowAspect: Design and Implementation

We introduce PerfFlowAspect, an open-source and user-friendly performance analysis tool designed for complex scientific workflows. PerfFlowAspect's design drew inspiration from Aspect Oriented Programming (AOP) [24], [6], which is a paradigm that increases modularity by separating out cross-cutting concerns, such as logging, error handling, or, in our case, performance monitoring. The benefits of AOP include separation of concerns, reduced code duplication, improved maintainability, and enhanced scalability. The concepts of AOP map to performance monitoring as shown in Table 1.

PerfFlowAspect's design is in response to the requirements discussed in Section 2. The remainder of this section describes PerfFlowAspect's implementation. AOP's influence can be clearly spotted in two of these requirements: (1) minimizing instrumentation at pointcuts (Requirement 1) and (2) using decorators and Low-Level Virtual Machine (LLVM) to realizing aspects in when weaving (Requirement 2).

Requirement 1: User-friendliness and minimal intrusion to source code. Our pointcut style includes the typical AOP practice of instrumentation only being required once at the beginning of a function. This style is referred to as an "around" pointcut and requires only a *single line* of annotation to capture an entire function's characteristics. This contrasts with the typical begin/end style in existing performance tools, where instrumentation is added in pairs. In particular, obviating "end" instrumentation and requiring only a single line of annotation makes the tool easy to learn for users, especially for domain scientists who do not have a background in computer science. It also creates a potential benefit in reducing user error during instrumentation. Further, this improves code readability by reducing code pollution. In addition to "around," our instrumentation also allows for "before" and "after" pointcuts (*i.e.,* the state of something before entering or after exiting a function). Finally, the types of advices obtained (timing, utilization measurements, and hardware performance counters) can be set dynamically via environment variables.

```
1  import PerfFlowAspect
2  import PerfFlowAspect.aspect
3
4  @PerfFlowAspect.aspect.critical_path("around")
5  def foo(msg):
6      do_work_here()
7      return 1 if msg == "hello" else 0
8
9  def main():
10     foo("hello")
```

Listing 4: Python code with PerfFlowAspect annotated functions.

```
1  {"name": "foo", "cat": "__main__", "pid": 26356, "tid": 26356,
       "ts": 1712881071887511.8, "ph": "X", "dur": 3043.0},
```

Listing 5: PerfFlowAspect generated trace file with compact logging.

Listings 4 and 5 show simple examples of PerfFlowAspect with Python. Users annotate the critical path or functions of interest within their application with a single line of code above the function, as shown on line 4 in Listing 4. C/C++ annotations are similar, and use the `__attribute__` specifier[1]. Whenever an annotated function is called in the program, performance data is recorded by the associated *advice* (see Table 1) in a trace file, as shown in Listing 5. PerfFlowAspect generates one trace file per process for C/C++ codes and a single trace file for processes in Python.

In this example, only timing data is collected, which is represented by `dur` in the trace file in microseconds. The trace also records the name of the annotated function (`name`), the calling function (`cat`), the process and thread identifiers (`pid` and `tid`), the timestamp (`ts`), and the verbosity (`ph`). Users can also choose to

[1] `__attribute__((annotate("@critical_path(pointcut='around')")))`

measure the processor and memory usage for each annotated function by setting environment variables. These are implemented with `getrusage()` in C/C++ and `psutil` in Python. Logging can be verbose (marked with `"ph": "B"`) and (`"ph": "E"`) or compact (`"ph": "X"`).

Requirement 2: Support for multiple languages. Performance data collection is currently supported for Python and C/C++. PerfFlowAspect uses decorators for the Python implemenation. Decorators act as a wrapper around a function by extending the behavior of the wrapped function without actually modifying it, enabling collection of performance data.

The C/C++ implementation uses a LLVM transformation pass [29]. It has three main modules: `parser`, `weaver` and `runtime`. The `parser` module parses the annotation with the `__attribute__` specifier and identifies locations of pointcuts and their type (around, before, after). This lexical analysis is done using `lex` and `yacc`. The `weaver` module collects performance data using a transformation pass. It supports both the Legacy Pass Manager and the New Pass Manager in LLVM [39], allowing portability ranging from `clang-10.0` through `clang-18.0`. In the case of the Legacy Pass Manager, a *FunctionPass* executes on each function in the program and overrides two virtual methods (`doInitialization` and `runOnFunction`). Functions that are annotated by the user in their source code are first identified in `doInitialization`. Then, additional measurement and logging code to collect timing and resource usage data is inserted before, after or around these user-annotated functions with the `runtime` module. This is accomplished by first either identifying the entry block for the function (when inserting before) or by iterating through the basic blocks of the function to find a return or resume instruction (when inserting after), and then utilizing the `getOrInsertFunction` to *weave* in the measurement code. The rationale for the New Pass Manager is very similar, except that it utilizes a *ModulePass*, and the associated `runOnModule` and `run` functions in a `PreservedAnalysis` class. The user's code base is built with the Clang compiler, and linked to the PerfFlowAspect library with standard build systems such as CMake. The use of an LLVM pass allows further extensions to PerfFlowAspect to support languages and tools such as Fortran and Caliper, respectively.

Requirement 3: Support for disparate data sources. PerfFlowAspect supports disparate data sources by having each component generate data in a way that the data can be collated afterwards to create a unified view. Trace files are generated in the Chrome Tracing Format (CTF) [19], which support composability and flexibility with their structure.

Requirement 4: Support for coarse-grained and fine-grained data. PerfFlowAspect provides native support for coarse-grained performance data, *i.e.,* time stamps, processor and memory usage for user-annotated functions. For fine-grained data, such as low-level hardware performance counters, PerfFlowAspect has an optional mode to include Caliper [7, 8] which it implements via an LLVM pass. Caliper's data can be collected through exisiting single line annotations without any additional instrumentation. From the user's perspective, this greatly improves the usability of Caliper itself by removing the need for tedious annotations. This also enables users to start with coarse-grained analysis and then apply the more cumbersome Caliper instrumentation after PerfFlowAspect identifies specific functions of interest.

Requirement 5: Interactive visualization of data. The CTF format can be interactively visualized in Perfetto [20]. Users can zoom in and out of timeline views based on the level of detail they need. They can click on individual functions to visualize timing, processor and memory usage, and get process- and thread-level information. Fine-grained data obtained through Caliper can be visualized with Hatchet [5, 9].

4 Evaluating PerfFlowAspect on Multi-Binary, Multi-Cluster Workflows

We begin this subsection by describing a complex multi-binary scientific workflow, which is based on the structure of a restricted-access workflow, called ICECap [49]. The ICECap workflow is based on the Merlin framework [47, 48], which is an open-source framework designed for large-scale ML workflows. Merlin can coordinate workflows across multiple HPC clusters when needed. It does so by using a persistent external queue server that lives outside the HPC systems but can talk directly to *workers* residing on the nodes of different HPC clusters. Merlin leverages some key open-source HPC and cloud computing technologies. Maestro [30] is used to define workflow task dependencies, and Celery [45] is used to translate those into concrete tasks, which are then configured using a RabbitMQ [55] and Redis [50] backend. This allows Merlin to scale to a very large numbers of workers, enabling large-scale simulations with very little overhead.

ICECap is a ML-assisted scientific workflow which demonstrates the optimization of a 17-parameter National Ignition Facility experiment [49]. The ICECap workflow involves six steps using three distinct programs (binaries) and two libraries (code bases). On the restricted side, one of these steps in ICECap runs a HYDRA [28] multi-physics simulation code. In order to run the ICECap workflow as close as possible to the original, only the HYDRA component was replaced by the Cloverleaf [22, 32] application. Cloverleaf is a proxy application that uses compressible Euler equations for computing 2D hydrodynamics.

Before the ICECap workflow starts running, 8 samples are generated as input by `Spellbook` [36], a Python utility library for Merlin. The first step sets up the database for the workflow (`setup-kosh`). ICECap uses Kosh [35] as its database on top of Sina [43]. This step is implemented as a Python program. The second step (`run-cloverleaf`) consists of three sub-tasks. The first sub-task creates an input deck for the simulation using the samples that were generated in the beginning. The second sub-task runs the simulation and the third collects the results. Each sub-task consists of a separate program and in our case, we ran the Cloverleaf simulation. `Spellbook` is used for the creation of the input deck, while a Python script is used for collection of results. Cloverleaf is implemented in C/C++ with MPI, and we leverage both CPU and GPU versions to demonstrate diversity in computation times of the samples. The third step pushes the collected data from the previous step to the Kosh database (`push-to-db`). This step is implemented as a Python program. The fourth step trains a machine learning model (`learner`), and the fifth step optimizes that model (`optimizer`). Both `learner` and `optimizer` use the DeepOpt library, which is written in Python. DeepOpt is used for performing Bayesian optimization, leveraging the powerful capabilities of BoTorch [4]. Its key feature is the ability

to use neural networks as surrogate functions during the optimization process, allowing Bayesian optimization to work smoothly even on large datasets and in many dimensions. The optimization is performed by running a single Bayesian optimization step, proposing a set of candidate points aimed at improving the value of the objective function, which is the output of the learned model. The final step focuses on performing another iteration to further optimize results. It launches set number of iterations (repeating steps 1-5) (`iterate`). When `iterate` is set to zero, only steps 1 through 5 are executed. This step does not involve a distinct program.

4.1 Experimental Setup

We ran the ICECap experiments on Lawrence Livermore National Laboratory's (LLNL) Corona and Ruby machines. Each Corona node contains a 2.0 GHz AMD Rome CPU with 48 cores and eight AMD 8xMI50 GPUs. Ruby nodes have a 2.2 GHz Intel Xeon CLX-8276L v4 CPU with 56 cores.

As described earlier, we used the Cloverleaf application for the second step in the workflow. Cloverleaf requires the selection of an input deck. The input deck specifies the size of the grid, initial parameters for the simulation, and the number of iterations. Our Cloverleaf experiments used an input deck that placed multiple high-energy spheres in a volume. Both CPU (MPI) and GPU (HIP) variants were leveraged. This deck sets up a $36500 \times 36500 \times 36500$ grid with 175 iterations for the GPU version, and a $30720 \times 30720 \times 30720$ grid with 150 iterations for the CPU version.

A Merlin workflow is specified with a *YAML* file that defines the steps (binaries), dependencies, required number of nodes, clusters, and input details. In our setup, Corona and Ruby have a shared file system. Merlin utilizes an on-premises cloud service, LaunchIT, deployed at LLNL to provide persistent services. The services used in this paper include Redis, RabbitMQ, and MariaDB. Merlin is connected to the RabbitMQ broker hosted on LaunchIT and contains a Celery queue. The *YAML* specification file also includes details to the shared file system paths to facilitate data sharing between the two clusters (e.g. `OUTPUT_PATH`). The workflow is invoked with the `merlin run` command, and Merlin reads the specified `OUTPUT_PATH` and hands it over to the Celery tasks it creates, which are then sent to queues on the RabbitMQ broker. Once the tasks are in the queues, they can be pulled from either cluster, provided Merlin is configured to connect to the same RabbitMQ instance.

Our domain scientists were interested in comparing the performance differences between a single-cluster and a multi-cluster execution of a complex workflow. To achieve this, we used two YAML files for our experiments. As a baseline, the first YAML file runs the entire ICECap workflow on Corona across nine nodes. The Cloverleaf step is executed with four samples running the CPU-only MPI version on four nodes, with 48 tasks per node. The remaining four samples were running the GPU variant on four nodes, each with eight tasks executing across the eight MI50 GPUs. In the second experiment, the YAML file specified that execution should occur on both Corona and Ruby, with an allocation of four nodes on Corona and five nodes on Ruby. Here, the eight samples were distributed equally across the two clusters. The four nodes on Ruby ran the Cloverleaf CPU-only version with 48 tasks on each node, and the four nodes on Corona ran the Cloverleaf-GPU

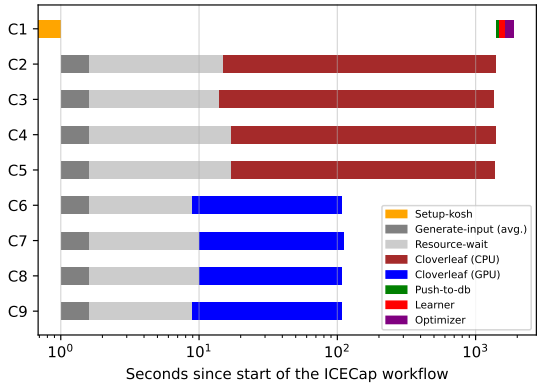

Figure 1: Gantt chart for the single-cluster experiment on Corona. Cloverleaf CPU and GPU samples run on nodes C2-C9, and the remaining steps run on node C1.

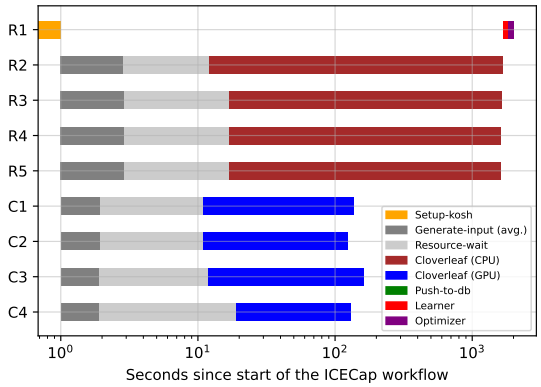

Figure 2: Gantt chart for the multi-cluster experiment on Ruby and Corona. Cloverleaf CPU runs on R2-R5, and Cloverleaf GPU runs on C1-C4. Remaining steps run on R1.

version across four nodes, with eight tasks on each node. The rest of the steps ran on Ruby.

In order for PerfFlowAspect to collect performance data, all of the constituent programs in the workflow must be instrumented. We did this for each of the three programs, Cloverleaf, and two libraries, meaning each would output its own trace during execution. A total of 93 single-line annotations were added. For some of the figures, we also wrote scripts that processed the data produced by PerfFlowAspect to produce custom visualizations using Matplotlib and Seaborn.

4.2 Holistic Performance Results

Figure 1 shows a Gantt chart of the high-level timeline view of the ICECap workflow running on a single cluster, Corona. The y-axis here indicates the cluster name and associated node number, representing the nine-node allocation on Corona. This view allows us to understand which step from the workflow was running on which node. The x-axis represents time in log scale.

Each bar for the `run-cloverleaf` step is divided into three parts. The first part (dark grey) is the average time taken to generate

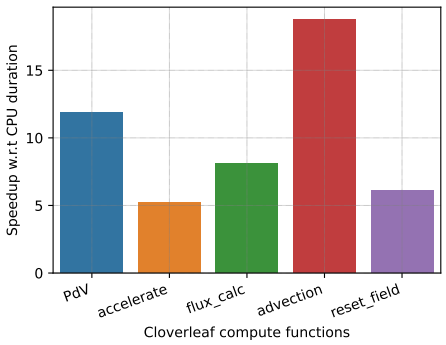

Figure 3: Speedup of Cloverleaf compute functions on the GPU with respect to the CPU.

inputs, measured using `/usr/bin/time`. The second part (light grey) is the wait time for node allocation once the simulation job is submitted by Merlin. The last part (brown or blue) is the duration of the Cloverleaf application, measured with PerfFlowAspect. We observe here that Cloverleaf is indeed on the workflow's critical path. It executes the eight samples across eight nodes (`C2-C9`). Four of these use the CPU version and four utilize GPUs. The remaining steps of the workflow all execute on a single node (`C1`), and their timing information is also collected with PerfFlowAspect.

The end-to-end turnaround time for the workflow for the single cluster run was 1,854.3 seconds. We note that out of the 72 MI50 GPUs allocated for the workflow (nine nodes, with eight GPUs each), only 32 (across the four nodes of Cloverleaf-GPU simulation) were utilized. This left more than half of the allocated GPUs idle, resulting in inefficient use of resources.

The Gantt chart in Figure 2 shows the high-level view of the ICECap workflow across Ruby and Corona, demonstrating a multi-cluster scenario. Here, the end-to-end turnaround time was 1,982.6 seconds, which is 6.9% slower. This was because the Cloverleaf CPU simulation step ran slower on Ruby (average of 1,636.7 seconds versus 1,379.1 seconds on Corona in the single-cluster experiment), despite running the same number of tasks (48) and Ruby having a higher CPU frequency. We attribute this to processor performance differences in the memory usage and caching. The `learner` step on R, however, was faster (175.1 seconds on Ruby as opposed to 262.7 seconds on Corona), offsetting some of this slowdown. It is important to note here that, despite the 6.9% slowdown, no resources (GPUs) were left unused, demonstrating better usage of overall resources at the site. High resource utilization of expensive hardware is preferred, making the multi-cluster approach favorable.

4.3 Per-Step Performance Results

4.3.1 Detailed Cloverleaf Results. Figure 3 shows the function-level speedup comparison of selected GPU kernels with respect to their CPU counterparts from a single iteration of Cloverleaf on Corona with data collected from PerfFlowAspect. Here, we observe that while some kernels, such as `PdV` and `advection`, result in reasonable speedups on the GPU, others, such as `accelerate` do not see a similar speedup. Such results enable our domain scientists to identify scaling bottlenecks at the kernel-level, allowing them to optimize relevant portions of their simulation code.

4.3.2 Detailed DeepOpt Optimizer Results. Figure 4 shows a snapshot of the Perfetto visualization of the trace file for the DeepOpt optimizer step. This interactive visualization is zoomed in to show only one iteration of the optimizer step. Such interactive function-level data helps users dig deeper into a single step's performance and identify functions with performance bottlenecks, as well as understand the call trees in the codebase. PerfFlowAspect users can also view processor and memory usage in Perfetto.

5 Fine-Grained Analysis of a Complex Single-Binary Application

This section demonstrates PerfFlowAspect applied to a complex single-binary application. It uses the Autonomous MultiScale (AMS) [46] library, which integrates ML-based surrogates into large-scale physics simulations. The goal of AMS is to reduce computation time while ensuring accuracy. Surrogate models are used when permissible as they execute faster, and slower physics-based models (IdealGas) are used to ensure reliable results. AMS is illustrated in Figure 5.

Each timestep has prologue and epilogue steps, both of which are not computationally expensive. AMS operates on the physics step, which is computationally expensive, with a fixed percentage of total samples, based on the input configuration, assigned to the ML-based surrogate model and the remainder assigned to the physics model. Results are then gathered and used for fine tuning and training the ML-model for the next iteration.

5.1 Experimental Setup

Our experiments were divided into three phases. The first two phases considered 486 experiments spanning a variety of AMS input configurations, with **Phase 1** focused on coarse-grained analysis and **Phase 2** focused on fine-grained analysis. **Phase 3**, which demonstrates visualization capabilities, considered fewer AMS input configurations (48), but considered a variety of concurrencies (5) for a total of 240 tests.

The AMS input configurations cover three key parameters: `threshold`, `physics_complexity`, and `number_of_elements`. The `threshold` controls the tradeoff between the physics module and machine learning model. Our experiments considered eight distinct values: 0.0 (only Ideal Gas), 0.1, 0.2, 0.4, 0.6, 0.8, 0.9, 1.0 (only ML surrogate). The `physics_complexity` parameter *artificially* adds complexity to the physics computation by using additional iterations and loops around the IdealGas module. For this parameter, we considered nine distinct values from 10 to 90. Finally, the `number_of_elements` sets the total number of samples. Here, there were six distinct values, from $20K$ elements to $120K$ elements. Also, as a baseline, we included a program that consists of only the physics computation and without any machine learning. This program is referred to as `no-AMS` in this text. Note that `no-AMS` is not the same as AMS with threshold 0.0 — threshold 0.0 does training but does not use the resulting model, whereas `no-AMS` does not perform training. In all, **Phase 1** and **2** are considering 432 combinations from the three factors ($6 \times 9 \times 8$) plus an extra 54 from `no-AMS` (6×9).

We ran all experiments on the Lassen cluster at LLNL. Each Lassen node contains an IBM Power9 CPU with 44 cores and four Nvidia Volta GPUs, but no GPUs were used for our results. Phase 1 and 2 experiments are single-node, and Phase-3 experiments are

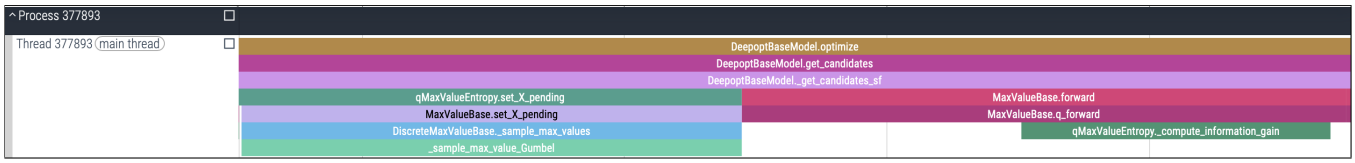

Figure 4: Zoomed-in Perfetto visualization of the DeepOpt optimizer step trace file generated by PerfFlowAspect.

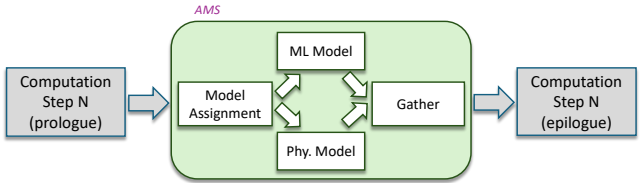

Figure 5: Single time-step depicting AMS in a simulation.

multi-node. All experiments were run as "sweeps," executing back-to-back on the same set of nodes.

5.2 Phase 1: Coarse-Grained Analysis of AMS

Figure 6 contains nine graphs corresponding to the nine values of physics complexity, depicting the performance of AMS's `main` function with respect to `no-AMS`. We note that while speedups can get as high as 2.5X, there are many configurations that run slower when using ML. We learn from this data that, counterintuitively, the Ideal Gas module is not always on the *critical path* of the application, and is computationally simple at low physics complexity.

Speedups can be observed for higher threshold values, especially when the complexity of the computation increases. Focusing on physics complexity, speedups only start occurring at a value of 30. The highest physics complexity (90) shows a generally positive story about using ML, although some configurations are still faster without it. The main reason for this is that the ML provides relatively uniform performance, while the performance of the Ideal Gas module depends on the underlying physics complexity. Furthermore, tasks outside of just the ML model, such as data loading and storing, are included in the timing of the `main` function.

Our domain scientist collaborators were specifically interested in timing and resource usage of the `evaluate` function within AMS. Figure 7 plots the speedup for the `evaluate` function. This function depicts the time taken for the evaluation of samples, where based on the threshold, a certain percentage would leverage the ML-model (`Surrogate::evaluate`) and others would leverage the physics model (`IdealGas::Eval`). As a result, this is the combined duration of the Ideal Gas and the ML-models within an AMS time-step, excluding the overhead of data loads and stores. The data here shows that the `evaluate` coming from a combination of ML and the Ideal Gas modules is considerably faster than using the Ideal Gas module alone, which we measure with `no-AMS`. Moreover, the speedups observed in Figure 7 for physics complexity of 30, 60 and 90 do not translate to the whole application, as can be seen in Figure 6. This verifies our earlier observation that the physics module is not always on the critical path.

Figure 8 shows both memory usage and CPU usage for various functions executed during a specific AMS experiment, obtained using the same annotation from PerfFlowAspect.

5.3 Phase 2: Fine-Grained Analysis of AMS

Figure 9 demonstrates how fine-grained data, such as hardware performance counters, can be obtained from Caliper using PerfFlowAspect's existing annotations. The figure shows L3 data and instruction cache counters for the `main` function in AMS. We believe this is a significant benefit over Caliper's native instrumentation approach, which can be tedious and error-prone. That said, this balance has some limitations, as PerfFlowAspect currently focuses on function-level data. In the future, we plan to extend PerfFlowAspect to support finer-grained annotations (*e.g.,* loop-level analysis).

5.4 Phase 3: Perfetto and Hatchet Visualization

PerfFlowAspect enables interactive trace file visualizations with Perfetto. We presented an example of this in Section 4 and omit this for AMS due to space considerations.

Figure 10 demonstrates how low-level data obtained through PerfFlowAspect's integration with Caliper can be visualized. This is a visualization of AMS `.cali` files and shows its performance across five different concurrencies — 2, 4, 8, 16, and 32 nodes, with four MPI ranks per node. This data shows that AMS does not scale well with the increase in MPI ranks and number of nodes for the physics model that was used.

6 Runtime Overhead

Table 2 shows experimental results for PerfFlowAspect's run-time overhead for Cloverleaf, AMS, and a synthetic Python benchmark. Each experiment was run five times. Experimental details for the three codes are as follows:

- **Cloverleaf** consisted of two sets of experiments, each run on both the CPU and the GPU. The first set varied the number of task counts with a fixed input mesh size (15360^3 for the CPU, $12K^3$ for the GPU) and step count (20 for the CPU, 175 on the GPU). The second set varied the input mesh size, ranging from $1K^3$ to $25K^3$ on the CPU and $15K^3$ to 35^3 on the GPU, with fixed number of tasks and step counts (150 for the CPU, 175 on the GPU). These experiments consisted of 23 different configurations and were run with "verbose" logging.
- **AMS** experiments had 12 configurations, with four threshold values and three complexity values. For each experiment, there were four types of measurement modes: runtime with only Caliper, with only PerfFlowAspect, with PerfFlowAspect with Caliper support, and with neither PerfFlowAspect nor Caliper as a baseline. The AMS team had previously added Caliper instrumentation to their code, which facilitated overhead comparison. These experiments were run with "compact" logging.

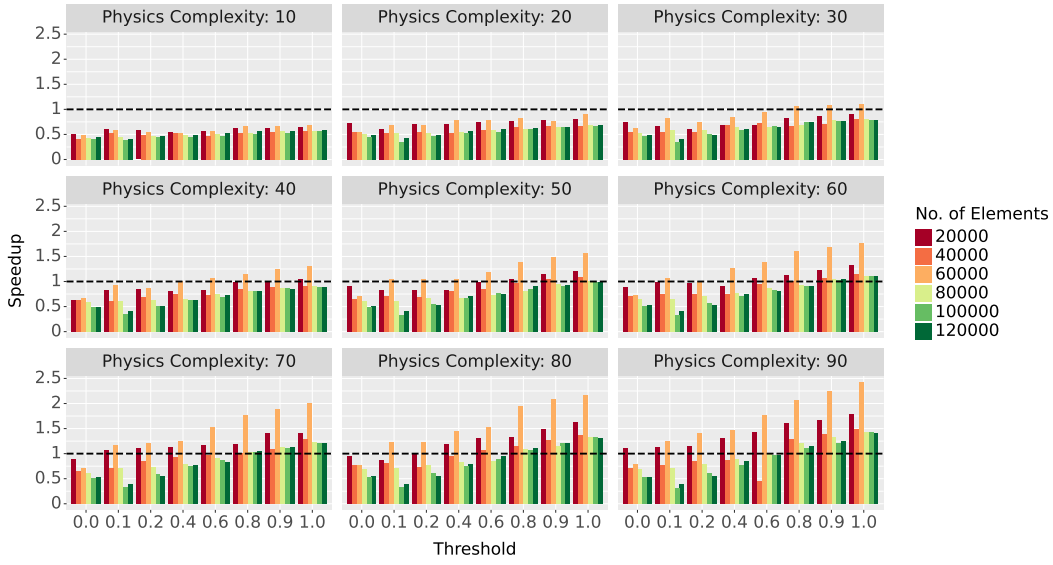

Figure 6: Nine sub-figures representing different physics complexity values. Each shows groups of six colored bars sharing the same threshold, which increases left to right, indicating greater AMS ML model usage over the Ideal Gas module. Bar heights show relative speedup vs. a `no-AMS` baseline. Across 432 values (each compared to one of 54 `no-AMS` runs), results reveal that ML speedups mostly occur as both threshold and physics complexity get larger. This suggests that physics computation is not always on the application's critical path.

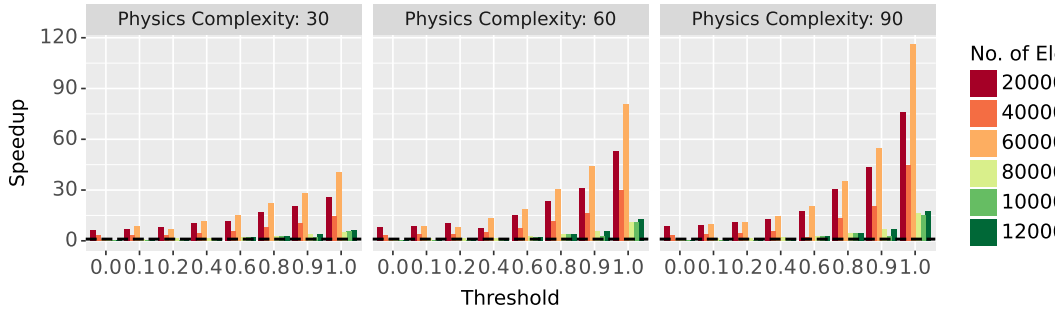

Figure 7: Speedup for the combined duration of Ideal Gas and ML model w.r.t. `no-AMS` Ideal Gas for physics complexities of 30, 60, and 90.

Table 2: Overhead of PerfFlowAspect for Cloverleaf, AMS and a synthetic Python benchmark.

Code	Experiment	Baseline	Architecture	Language	Minimum Relative Runtime	Maximum Relative Runtime	Average Relative Runtime
Cloverleaf	PerfFlowAspect	No PerfFlowAspect	CPU	C/C++	97.8%	106.9%	100.2%
Cloverleaf	PerfFlowAspect	No PerfFlowAspect	GPU	C/C++	96.9%	104.6%	100.3%
AMS	PerfFlowAspect, no Caliper	No PerfFlowAspect, No Caliper	CPU	C++	100.4%	107.8%	103.0%
AMS	Caliper, no PerfFlowAspect	No PerfFlowAspect, No Caliper	CPU	C++	99.0%	103.3%	101.0%
AMS	PerfFlowAspect + Caliper	No PerfFlowAspect, No Caliper	CPU	C++	101.9%	117.1%	107.3%
Python Benchmark	PerfFlowAspect	No PerfFlowAspect	CPU	Python	98.3%	101.1%	99.9%

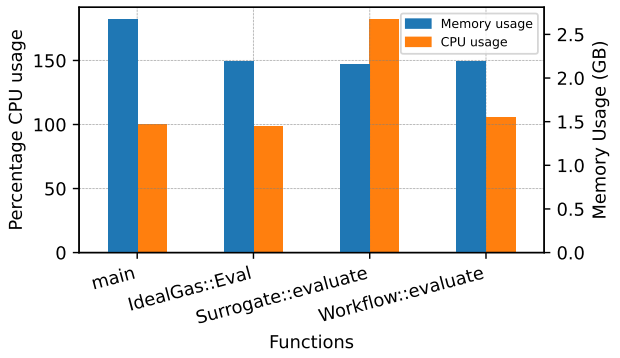

Figure 8: CPU usage (%) and memory usage (GBs) for 4 functions in AMS with threshold of 0.8, physics complexity of 80, and $100K$ elements. The left y-axis is for CPU usage while the right y-axis is for memory usage.

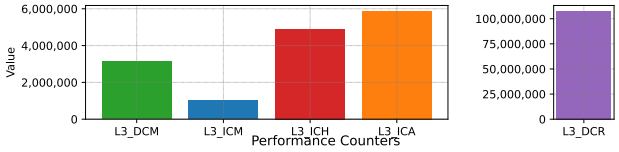

Figure 9: Integration with Caliper enables collection of low-level performance counters. The plot shows the maximum values for five counters for `main`, with a threshold of 0.8, physics complexity of 80, and $100K$ elements.

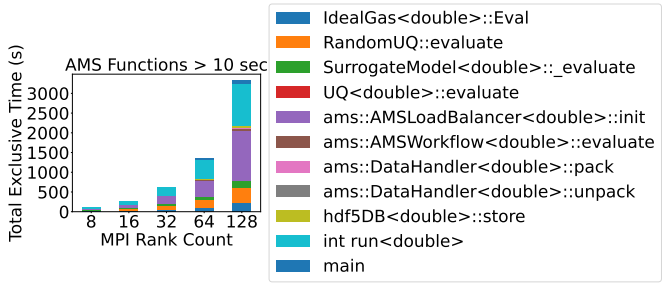

Figure 10: Hatchet comparison of AMS scalability across different MPI rank counts. This plot only includes functions that ran 10 seconds or more. The y-axis is the total exclusive time in seconds for all the functions for each task count.

- **The synthetic Python benchmark** consists of a multi-process program with multiple functions. One of the functions, with a mathematical computation within a spin loop, is called repeatedly in each process. This benchmark was used to perform experiments with five different process counts ranging from 1 to 48, resulting in five different configurations, with each "verbose" and "compact" logging.

Overall, Table 2 shows that PerfFlowAspect's overhead is small (0.9% on average), across codes, architectures, and languages. In some cases, experiments with PerfFlowAspect even ran faster than experiments without PerfFlowAspect, demonstrating that the variability in runtime was generally larger than the overhead. The most significant overheads (7.3% on average) were for the AMS experiments with both PerfFlowAspect and Caliper enabled. Other AMS experiments showed that PerfFlowAspect with "compact" mode logging has an overhead of 3% on average, which is slightly higher than that of Caliper (1% average). Note that PerfFlowAspect's "compact" mode with C/C++ has a higher overhead than its verbose counterpart. Overhead in the "verbose" mode with C/C++ is negligible, as shown in the Cloverleaf data. For the Python experiments, the overhead was negligible for both the "verbose" and the "compact" logging modes.

Our average overhead of 0.9% compares favorably with overheads documented in other works for Caliper, TAU, HPCToolkit, and Score-P [11, 27, 34]. The lowest documented overhead among these tools is on the order of 1-3% while the highest overhead doubles the duration of the application. Overall, the runtime overhead when using all these performance tools is similar, except for the case of Score-P for some of the applications. Of course, the amount of overhead is correlated with the amount of performance measurements taken, instrumentation techniques, and the implementation details of each tool. As PerfFlowAspect is taking fewer measurements, with instrumentation along the critical path, it is not surprising that our overhead is lower. That said, we still conclude that overhead is not problematic for our tool.

7 Discussion: Instrumentation Effort and Other Considerations

Merlin and AMS demonstrations show the capability of PerfFlowAspect in different ways. With Merlin, PerfFlowAspect is able to generate traces from the disparate programs (Python and C/C++), architectures (CPUs and GPUs), and HPC clusters (Ruby and Corona) in a manner that can be reconstituted for holistic analysis (Section 4.2). With AMS, PerfFlowAspect was able to successfully perform both coarse- and fine-grained analysis with a single set of annotations.

The Merlin workflow shows the benefit of PerfFlowAspect's approach for minimizing instrumentation efforts. As Merlin is made up of many distinct programs, the additional work required to instrument them is magnified and is error prone. PerfFlowAspect minimizes the per-program instrumentation and maintains code readability, making PerfFlowAspect's lightweight approach more attractive in these settings. In the ICECap workflow, the instrumentation code was less than 100 lines of code across all its programs.

The AMS application was different, as it involved ensemble runs of the same binary. Another key difference is that the AMS team had previously added Caliper support to their code, which enabled comparison between PerfFlowAspect's instrumentation with their efforts. We added 57 PerfFlowAspect single-line annotations around 57 AMS functions. For Caliper, 59 pairwise region annotations (with `CALI_MARK_BEGIN` and `CALI_MARK_END` markers) were added. However, these annotations covered only a subset of the functions we instrumented with PerfFlowAspect. Lastly, an AMS function with 18 lines of code had 10 lines of Caliper instrumentation code added (as opposed to a single line of PerfFlowAspect instrumentation).

With its integration with Caliper, PerfFlowAspect benefits from access to low-level hardware performance counter data that Caliper

provides. Utilizing PerfFlowAspect's single-line annotations to obtain Caliper data makes fine-grained analysis with Caliper easier, establishing a synergistic relationship between the two tools.

Different performance analysis scenarios may require different tools based on the user's need. Tools such as Caliper and TAU are widely used and well-suited for fine-grained analysis of traditional single binary applications. That said, their match for the complex multi-cluster, multi-binary workflow use case is not as strong. In the context of a user who is not a developer of an application, but rather combining multiple applications into a workflow, traditional performance tools can have a steep learning curve, and can be error-prone, cumbersome, and/or hard to debug. Further, the resulting profiles and trace files are hard to understand from a user's perspective. In response, our goal is to introduce a capable and much friendlier tool for users that enables analysis for their complex multi-cluster and multi-binary workflows.

8 Conclusions and Future Work

We introduced PerfFlowAspect, which has several key properties that are necessary for performance analysis of modern-day complex scientific workflows. We demonstrated its effectiveness with the analysis of the multi-binary and multi-cluster Merlin ICECap workflow as well as its ability for coarse- and fine-grained analysis with the AMS workflow. PerfFlowAspect is designed for reducing the overall burden on the user. It enables easier instrumentation for complex scientific workflows, preserves code readability and reduces annotation-related errors. It also generates composable and easy-to-understand trace files that support interactive visualization natively. PerfFlowAspect accomplishes this with a minimal overhead of 0.9% on average. It does not sacrifice the benefits that previous performance analysis tools have introduced, and instead, can also help improve the usability of tools such as Caliper.

Future research directions will involve identifying techniques to further lower PerfFlowAspect's overhead, with methods such as compression and aggregation for large trace files. Developing advanced call-graph visualization techniques and integrating with tools such as Thicket [10] is another direction for research. Some engineering challenges remain as well, which include designing general-purpose data parsing and graphing scripts, adding support for Fortran-based codes, and collection of additional performance data, such as power, I/O and network counters. Moreover, more thorough comparative studies may help demonstrate PerfFlowAspect's advantages over existing tools. Finally, while our demonstrations were on a traditional HPC cluster, PerfFlowAspect can also run in a Cloud setting. That said, we plan to explore scientific workflows that combine HPC and Cloud in the future.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 24-SI-005 (LLNL-CONF-863532).

References

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. 2014. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 154–165. doi:10.1109/SC.2014.18

[3] Dong H. Ahn, Xiaohua Zhang, Jeffrey Mast, Stephen Herbein, Francesco Di Natale, Dan Kirshner, Sam Ade Jacobs, Ian Karlin, Daniel J. Milroy, Bronis De Supinski, Brian Van Essen, Jonathan Allen, and Felice C. Lightstone. 2022. Scalable Composition and Analysis Techniques for Massive Scientific Workflows. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. 32–43. doi:10.1109/eScience55777.2022.00018

[4] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*. http://arxiv.org/abs/1910.06403

[5] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 20, 21 pages. doi:10.1145/3295500.3356219

[6] Walter Binder, Danilo Ansaloni, Alex Villazón, and Philippe Moret. 2011. Flexible and efficient profiling with aspect-oriented programming. *Concurr. Comput.: Pract. Exper.* 23, 15, 1749–1773. doi:10.1002/cpe.1760

[7] David Boehme, Pascal Aschwanden, Olga Pearce, Kenneth Weiss, and Matthew LeGendre. 2021. Ubiquitous Performance Analysis. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 431–449.

[8] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 47, 11 pages.

[9] Stephanie Brink, Ian Lumsden, Connor Scully-Allison, Katy Williams, Olga Pearce, Todd Gamblin, Michela Taufer, Katherine E. Isaacs, and Abhinav Bhatele. 2020. Usability and Performance Improvements in Hatchet. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. 49–58. doi:10.1109/HUSTProtools51951.2020.00013

[10] Stephanie Brink, Michael McKinsey, David Boehme, Connor Scully-Allison, Ian Lumsden, Daryl Hawkins, Treece Burgess, Vanessa Lama, Jakob Lüttgau, Katherine E. Isaacs, Michela Taufer, and Olga Pearce. 2023. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 281–293. doi:10.1145/3588195.3592989

[11] Onur Cankur and Abhinav Bhatele. 2022. Comparative Evaluation of Call Graph Generation by Profiling Tools. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham, 213–232.

[12] Rafael Ferreira da Silva. 2022. An Overview of AI Workflows for HPC Systems. https://www.nitrd.gov/nitrdgroups/images/2023/MAGIC-Rafael-Ferreira-da-Silva-05032023.pdf

[13] da Silva et al. 2021. A Community Roadmap for Scientific Workflows Research and Development. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE. doi:10.1109/works54523.2021.00016

[14] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. 2018. The future of scientific workflows. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 159–175. doi:10.1177/1094342017704893 arXiv:https://doi.org/10.1177/1094342017704893

[15] Francesco Di Natale, Harsh Bhatia, Timothy S. Carpenter, Chris Neale, Sara Kokkila-Schumacher, Tomas Oppelstrup, Liam Stanton, Xiaohua Zhang, Shiv Sundram, Thomas R. W. Scogland, Gautham Dharuman, Michael P. Surh, Yue Yang, Claudia Misale, Lars Schneidenbach, Carlos Costa, Changhoan Kim, Bruce D'Amora, Sandrasegaram Gnanakaran, Dwight V. Nissley, Fred Streitz, Felice C. Lightstone, Peer-Timo Bremer, James N. Glosli, and Helgi I. Ingólfsson. 2019. A massively parallel infrastructure for adaptive multiscale simulations: modeling RAS initiation pathway for cancer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 57, 16 pages. doi:10.1145/3295500.3356197

[16] Isaac Dooley, Chee Wai Lee, and Laxmikant V. Kale. 2009. Continuous performance monitoring for large-scale parallel applications. In *2009 International Conference on High Performance Computing (HiPC)*. 445–452. doi:10.1109/HIPC.2009.5433181

[17] Thomas et al. 2023. Computationally restoring the potency of a clinical antibody against SARS-CoV-2 Omicron subvariants. *bioRxiv* (2023). doi:10.1101/2022.10.21.513237 arXiv:https://www.biorxiv.org/content/early/2023/04/24/2022.10.21.513237.full.pdf

[18] Ke Fan, Suraj Kesavan, Steve Petruzza, and Sidharth Kumar. 2024. TinyProf : Towards Continuous Performance Introspection through Scalable Parallel I/O. In *International Supercomputing Conference HPC (ISC-HPC)*.

[19] Inc. Google. 2016. Chrome Trace Event Format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview#heading=h.yr4qxyxotyw

[20] Inc. Google. 2024. Perfetto: System profiling, application tracing and trace analysis. https://perfetto.dev/

[21] Robert Graves, Thomas Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, David Okaya, Patrick Small, and Karan Vahi. 2010. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics* 168 (03 2010), 367–381. doi:10.1007/s00024-010-0161-6

[22] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. 2012. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 465–471. doi:10.1109/SC.Companion.2012.66

[23] Shantenu Jha, Vincent R. Pascuzzi, and Matteo Turilli. 2022. AI-coupled HPC Workflows. arXiv:2208.11745 [cs.DC]

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[25] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155.

[26] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[27] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[28] Steven H. Langer, Ian Karlin, and Marty M. Marinak. 2014. Performance Characteristics of HYDRA - a Multi-Physics simulation code from Lawrence Livermore National Laboratory. (1 2014). doi:10.2172/1116974

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[30] Lawrence Livermore National Laboratory. 2017. Maestro Workflow Conductor. https://github.com/LLNL/maestrowf

[31] Viktor Leis. 2025. PerfEvent: A header-only C++ wrapper for Linux' perf event API. https://github.com/viktorleis/perfevent

[32] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2021. On measuring the maturity of SYCL implementations by tracking historical performance improvements. In *Proceedings of the 9th International Workshop on OpenCL* (Munich, Germany) *(IWOCL '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 13 pages. doi:10.1145/3456669.3456701

[33] Aliza Lisan, Tapasya Patki, Stephanie Brink, Zhiwei Yang, Spencer Greene, Konstantinos Parasyris, Shubbhi Taneja, and Hank Childs. 2024. PerfFlowAspect: A User-Friendly Performance Tool for Scientific Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC24)*. https://sc24.supercomputing.org/proceedings/poster/poster_files/post223s2-file3.pdf Poster.

[34] Xu Liu and John Mellor-Crummey. 2014. A tool to analyze the performance of multithreaded programs on NUMA architectures. *SIGPLAN Not.* 49, 8 (Feb. 2014), 259–272. doi:10.1145/2692916.2555271

[35] LLNL. 2020. kosh. https://github.com/LLNL/kosh.

[36] LLNL. 2020. merlin-spellbook. https://github.com/LLNL/merlin-spellbook.

[37] LLNL. 2023. PerfFlowAspect. https://github.com/flux-framework/PerfFlowAspect.

[38] LLNL. 2024. Caliper 2.11.0-dev documentation: Annotation API reference. https://software.llnl.gov/Caliper/AnnotationAPI.html

[39] LLVM Compiler Infrastructure. 2024. Status of the New and Legacy Pass Managers. https://llvm.org/docs/NewPassManager.html#status-of-the-new-and-legacy-pass-managers

[40] Jonathan R. Madsen, Muaaz G. Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. 2020. Timemory: Modular Performance Analysis for HPC. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 434–452.

[41] A.D. Malony, S. Shende, and R. Bell. 2004. Online performance observation of large-scale parallel applications. In *Parallel Computing*, G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter (Eds.). Advances in Parallel Computing, Vol. 13. North-Holland, 761–768. doi:10.1016/S0927-5452(04)80093-5

[42] John Mellor-Crummey. 2003. HPCToolkit: Multi-platform tools for profile-based performance analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*.

[43] Mohamed F. Mokbel, Xiaopeing Xiong, and Walid G. Aref. 2004. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 623–634. doi:10.1145/1007568.1007638

[44] University of Oregon. 2024. 1.7. Running a Python application with TAU: Chapter 1. TAU instrumentation Options. https://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01s07.html

[45] Open Source Community. 2024. Celery: Distributed Task Queue. https://docs.celeryq.dev/en/stable/index.html

[46] Konstantinos Parasyris, Loïc Pottier, Daniel J. Milroy, Harsh Bhatia, Tapasya Patki, Stephanie Brink, Thomas Stitt, Daniel E. Laney, Robert C. Blake, Jae-Seung Yeom, Peer-Timo Bremer, and Charles Doutriaux. 2023. Autonomous MultiScale Library. https://github.com/LLNL/AMS

[47] J. Luc Peterson, Rushil Anirudh, Kevin Athey, Benjamin Bay, Peer-Timo Bremer, Vic Castillo, Francesco Di Natale, David Fox, Jim A. Gaffney, David Hysom, Sam Ade Jacobs, Bhavya Kailkhura, Joe Koning, Bogdan Kustowski, Steven H. Langer, Peter B. Robinson, Jessica Semler, Brian K. Spears, Jayaraman J. Thiagarajan, Brian Van Essen, and Jae-Seung Yeom. 2019. Merlin: Enabling Machine Learning-Ready HPC Ensembles. *CoRR* abs/1912.02892 (2019).

[48] J. Luc Peterson, Ben Bay, Joe Koning, Peter Robinson, Jessica Semler, Jeremy White, Rushil Anirudh, Kevin Athey, Peer-Timo Bremer, Francesco Di Natale, David Fox, Jim A. Gaffney, Sam A. Jacobs, Bhavya Kailkhura, Bogdan Kustowski, Steven Langer, Brian Spears, Jayaraman Thiagarajan, Brian Van Essen, and Jae-Seung Yeom. 2022. Enabling machine learning-ready HPC ensembles with Merlin. *Future Gener. Comput. Syst.* 131, C (jun 2022), 255–268. doi:10.1016/j.future.2022.01.024

[49] J. Luc Peterson, Tim Bender, Robert Blake, Nai-Yuan Chiang, M. Giselle Fernández-Godino, Bryan Garcia, Andrew Gillette, Brian Gunnarson, Cooper Hansen, Judy Hill, Kelli Humbird, Bogdan Kustowski, Irene Kim, Joe Koning, Eugene Kur, Steve Langer, Ryan Lee, Katie Lewis, Alister Maguire, Jose Milovich, Yamen Mubarka, Renee Olson, Jay Salmonson, Chris Schroeder, Brian Spears, Jayaraman Thiagarajan, Ryan Tran, Jingyi Wang, and Chris Weber. 2024. Toward digital design at the exascale: An overview of project ICECap. *Physics of Plasmas* 31, 6 (6 2024). doi:10.1063/5.0205054

[50] Redis Labs. 2024. Redis: Remote Dictionary Server. https://github.com/redis/redis

[51] Mats Rynge, Gideon Juve, Jamie Kinney, John Good, Bruce Berriman, Ann Merrihew, and Ewa Deelman. 2013. Producing an infrared multiwavelength galactic plane atlas using montage, pegasus and amazon web services. In *23rd Annual Astronomical Data Analysis Software and Systems, ADASS, Conference*.

[52] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (may 2006), 287–311. doi:10.1177/1094342006064482

[53] The Krell Institute and Open Source Community. 2024. Open|SpeedShop. https://github.com/OpenSpeedShop/openspeedshop

[54] Nicholas Tyler, Robert Knop, Deborah Bard, and Peter Nugent. 2022. Cross-Facility Workflows: Case Studies with Active Experiments. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*. 68–75. doi:10.1109/WORKS56498.2022.00014

[55] Anthony Wood. 2016. *Rabbit MQ: For Starters*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[56] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. 1999. Toward Scalable Performance Visualization with Jumpshot. *Int. J. High Perform. Comput. Appl.* 13, 3 (Aug. 1999), 277–288. doi:10.1177/109434209901300310