SYMBIOSYS: A Methodology for Performance Analysis of Composable HPC Data Services

Srinivasan Ramesh* Allen D. Malony* Philip Carns† Robert B. Ross†
Matthieu Dorier† Jerome Soumagne‡ Shane Snyder†
*University of Oregon {sramesh, malony}@cs.uoregon.edu
†Argonne National Laboratory {mdorier}@anl.gov, {carns, rross, ssnyder}@mcs.anl.gov
‡The HDF Group {jsoumagne}@hdfgroup.org

***Abstract*—Microservices are a powerful new way of building, customizing, and deploying distributed services owing to their flexibility and maintainability. Several large-scale distributed platforms have emerged to serve the growing needs of data-centric workloads and services in commercial computing. Concurrently, high-performance computing (HPC) systems and software are rapidly evolving to meet the demands of diversified applications and heterogeneity. The interplay of hardware factors, software configuration parameters, and the flexibility offered with a microservice architecture makes it nontrivial to estimate the optimal service instantiation for a given application workload. Further, this problem is exacerbated when considering that these services operate in a dynamic and heterogeneous HPC environment. An optimally integrated service can be vastly more performant than a haphazardly integrated one. Existing performance tools for HPC either fail to understand the request-response model of communication inherent to microservices or they operate within a narrow scope, limiting the insight that can be gleaned from employing them in isolation.**

We propose a methodology for integrated performance analysis of HPC microservices frameworks and applications called SYMBIOSYS. We describe its design and implementation within the context of the Mochi framework. This integration is achieved by combining distributed callpath profiling and tracing with a performance data exchange strategy that collects fine-grained, low-level metrics from the RPC communication library and network layers. The result is a portable, low-overhead performance analysis setup that provides a holistic profile of the dependencies among microservices and how they interact with the Mochi RPC software stack. Using HEPnOS, a production-quality Mochi data service, we demonstrate the low-overhead operation of SYMBIOSYS at scale and use it to identify the root causes of poorly performing service configurations.

***Index Terms*—microservices, storage, performance, tools**

I. Introduction

Storage systems on today's high-performance computing (HPC) platforms are complex and rapidly evolving because of the continuous adoption of new technologies in storage hardware, networking infrastructure, memory, and compute resources. On the application front, the traditional MPI-based parallelism is increasingly supplemented by large-scale task parallelism [1], [2]. The heterogeneity in hardware, diversified application mix, and execution environments coupled with increasing on-node parallelism complicates the task of managing and optimizing I/O performance and meeting the application's data needs. Composability is a useful development paradigm for this kind of complex environment; it allows distributed services to be incrementally developed and improved in a modular fashion. The Mochi project [3] is an example of an HPC framework that embodies this principle. It structures and catalyzes the development of customized HPC data services through the use of microservices that can be rapidly composed to meet application requirements.

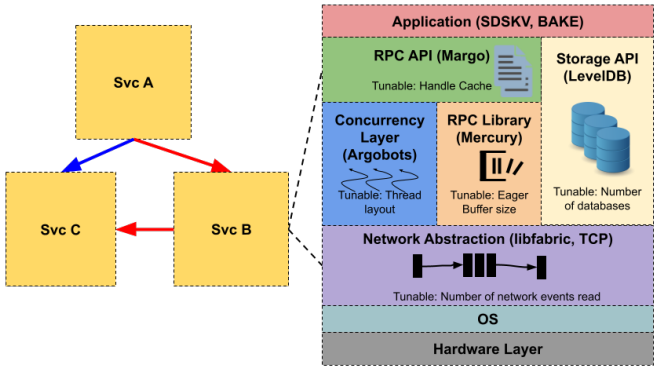


Fig. 1: Mochi: Interaction between Distributed Components and Software Stack

As evidenced by ongoing efforts in the software industry [4], this paradigm of using microservices to design distributed software has the advantage of being highly maintainable and flexible. However, this engineering approach complicates the task of performance analysis and optimization. Basic questions such as deciding on the optimal service configuration or detecting resource saturation are nontrivial in the context of microservices. The following questions, representing the key analysis and tuning activities with HPC microservices, motivate the need for an integrated performance infrastructure.

1) **What combinations of dependent microservice operations have the greatest impact on performance?** Conceptually, the dependencies among the microservices are represented as distributed callpaths through the system. By analyzing the structure of the callpath, the distribution of the call times, and the call counts, the developer can quickly track down the most resource-intensive callpath and the load distribution for that callpath. As opposed to monolithic architectures where callpaths are local to a process, generating callpaths for microservices is inherently difficult because these callpaths can span across multiple processes on different nodes. Microservices that make up a composed service are loosely coupled, work on potentially different scales, operate in a heterogeneous execution environment,

and are configured in myriad different ways. Figure 1 depicts a scenario where three microservices (A, B, and C) interact to generate two distinct callpaths in the system: A⟶B⟶C (shown in red) and A⟶C (shown in blue). The microservices A, B, and C can be located on the same process, on different processes within a node, or on entirely separate nodes depending on how the service is configured. Concerning the generation of callpaths, the microservice model breaks the assumption that most HPC performance tools make about the passage of control being limited to addresses within a process.

2) **How do resource utilization and time map to individual steps in a microservice operation?** Once the microservice dependencies have been identified, it is imperative to understand the relative contributions of various software components and events that make up the remote procedure call (RPC) on the client and the server. To tease out these details, we need a way to integrate various sources of performance data and timers gathered from different levels in the stack and fuse that data with the distributed callpaths as common reference points.
3) **What hardware and software resources are being saturated?** Identifying a poorly performing service configuration involves the ability to detect resource saturation. Resource saturation can occur on a hardware level or software level. For example, tasking frameworks that manage concurrency on the server place newly spawned tasks in internal queues. Observing a backlog of tasks on these queues is an indication that the tasking system is starved of compute resources. Correlating these resource saturation metrics with higher-level RPC callpath information can help narrow down the cause for the task pileup.
4) **Is there a better service configuration?** Ultimately, the goal of the performance analysis framework is to aid in the generation of a better, more optimal service configuration, if it exists. If resource saturation is detected, the performance data must be suffiently able to indicate what parameters could be changed in order to improve performance.

While existing tools excel at the performance analysis of a specific component or distributed architecture, they provide only a disjoint or partial profile of microservice performance at best. An integrated performance system that utilizes a combination of different performance instrumentation and measurement strategies is necessary to holistically evaluate microservice performance and answer these questions. In this paper, we present *SYMBIOSYS*, a system for capturing distributed callpath information, observing key timing and resource saturation metrics, and fusing this data in a meaningful way to glean insights into the behavior of distributed HPC microservices. The contributions of our research work are as follows:

- Design of distributed performance measurement and analysis architecture for microservice-based HPC environments
- Design of a performance data exchange framework for RDMA-based RPC communication libraries
- Development of a specific instantiation of the architecture tailored for Mochi data services
- Experiments to analyze the root cause of Mochi service performance inefficiencies on real-world HPC systems

Mochi has enabled the development of a wide variety of HPC data services. Examples of such services include FlameStore [3], a data service designed to support distributed deep learning workflows; GekkoFS [5], a scalable POSIX-like filesystem with relaxed semantics; and HEPnOS [3], a storage service designed for high energy physics simulations. We expect our performance framework to support this wide range of HPC service and execution environments that are enabled by Mochi. In Section §II we discuss related research. In Section §III we provide an overview of the Mochi project. This will set the stage for the SYMBIOSYS performance infrastructure proposed in Section §IV. We also describe the implementation of this framework in the context of Mochi. In Section §V we demonstrate its use. An overhead study of the performance framework is presented in Section §VI. In Section §VII we give concluding remarks and briefly discuss future work.

II. Related Work

We present a brief overview of the analysis activities that are central to HPC microservices and the effectiveness of different classes of tools to address these requirements.

A. HPC Performance Tools

HPC performance tools excel at the performance analysis of applications based on the distributed-memory parallel programming model. State-of-the-art tools such as TAU [6], ScoreP [7], CALIPER [8], and HPCToolkit [9] employ sophisticated sampling, automatic compiler instrumentation, manual instrumentation, and library interposition techniques to gather insights into application and communication library performance. Typically, they build on the presence of an MPI programming model to capture distributed performance information. These tools implicitly assume that control is not passed *between* applications. As a result, HPC performance tools cannot be directly utilized to observe distributed microservice callpaths. While some HPC tools are capable of working with user-level tasking frameworks such as Argobots (e.g., APEX [10]), almost all are constrained to measuring code performance within a node, with limited application in the generation of distributed callpaths.

B. Cloud-Based Tools for Microservices

Within the general distributed systems community and industry, several efforts have been made to design performance tools for microservice-based distributed services. Broadly speaking, they employ some form of metadata propagation to stitch together request trace events across processes to form a complete picture. Distributed request tracing is effective in detecting structural and empirical anomalies [11]. A comprehensive survey of the variety of tools available for distributed

tracing can be found in [12]. Dapper [13] from Google, OpenZipkin [14], and Jaeger [15] are the notable industry efforts at tracking requests and associated metadata through a hyperscale distributed setup. Our distributed tracing implementation is compatible with the trace format of these tools. Unlike these tools, however, SYMBIOSYS does not require additional processes on the node for staging performance data. Further, we find the need to extend their data model to support the generation and capture of a rich variety of performance data from across the stack.

C. Tools That Integrate Data Sources

A growing body of research employs techniques to exchange vital performance data between software layers. Notably, within the MPI community there are ongoing efforts [16], [17], [18] to expose internal MPI counters and events in order to gain a deeper insight into the distributed communication performance. The OpenMP community is also pursuing similar efforts [19], [20] to associate library-level performance data with higher-level tasks. The PAPI software-defined events [21] approach aims to standardize the exchange of software-level performance metrics across layers through *accessor functions*. Our performance data strategy is inspired by the efforts in the MPI community, whereby tool support is directly available in the communication library as opposed to employing an external component.

III. Background

The difficulty with enabling the analyses raised in Section I lies in understanding how to combine a variety of instrumentation and measurement techniques to present an integrated analysis and profile. Observing distributed callpaths, for example, involves tracking and forwarding RPC call ancestry across distributed microservices by employing some form of request metadata propagation. Attributing resource usage to individual steps within a microservice operation involves exchanging performance data across the software stack and orienting it around appropriate reference points. Identifying resource saturation requires the ability to correlate low-level performance metrics with high-level callpath information. While these techniques can be broadly applicable to any microservice environment, we focus on a proof of concept using the Mochi software stack. Section III provides a brief overview of the Mochi RPC execution model and describes the instrumentation points relevant to SYMBIOSYS performance analysis.

A. Mochi Overview

Mochi data services are built and composed by using the RPC as the fundamental communication method between processing elements, whether they are local or remote. A Mochi client (referred to interchangeably as an *origin* entity) contacts a service provider (referred to as a *target* entity). The Mochi ideology is to provide the tools and environment necessary to enable the rapid development of HPC data services. This goal is achieved by *composing* microservices to build higher-level, customized functionality. Examples of microservices include BAKE [22], a microservice for storing and retrieving object blobs; SDSKV, a microservice enabling RPC-based access to multiple key-value backends; and REMI, a microservice to enable the shifting of data between microservice instances. For a full description of the microservices and the composed services that are built on top of them, see [3].

B. Mochi Core Framework

Mochi's core components include Argobots [23], Mercury [24], Margo, Thallium, and SSG (Scalable Service Groups). For our study, we focus on the first three.

1) Argobots: Developed outside the scope of the Mochi project, Argobots is a user-level threading library designed for highly concurrent systems. Argobots decouples work to be done (user-level threads, or ULTs) from the hardware resources that perform that work (execution streams, or ESs).

2) Mercury: Mercury is an RPC framework designed for HPC environments. Mercury takes advantage of RDMA-enabled HPC networks for large data transfers and a callback-driven completion model for concurrency.

3) Margo: As illustrated in Figure 1, Margo is the common underlying layer for Mochi services to interact with RPCs and RPC handlers. Margo eases the burden of Mercury callback programming and Argobots concurrency management and presents a unified model that leverages both technologies. Margo operates in two modes: client and server. When used in server mode, Margo allows the registration of one or more *providers*. Essentially, a provider is an *instantiation* of a class that implements a given microservice API. Clients forward RPC requests to uniquely addressable microservice providers within each service process.

C. Mochi RPC Execution Model

We describe the events that occur during the generation and execution of a Mochi RPC call.

1) Request Generation: After the target provider address has been acquired, the origin entity generates an RPC request. The RPC request metadata is serialized inside Mercury and sent over to the target eagerly. In the case that the eager buffer overflows, Mercury employs an internal RDMA call to send the additional request metadata. Margo installs a callback with Mercury that is invoked when the response is available. These actions correspond to steps $\mathbf{t_1}$ to $\mathbf{t_3}$ in Figure 2.

2) Execution of the Request: When a target provider receives an RPC request, the main service provider execution stream (progress ES) creates a *new* ULT to service the request ($\mathbf{t_4}$). This request enters a pool of tasks waiting to run on the next available ES. When the ULT is assigned an ES to run on, it begins executing ($\mathbf{t_5}$) by first deserializing the input metadata ($\mathbf{t_6}$ to $\mathbf{t_7}$). The number of ESs available to the service provider is specified during the initialization phase. These ESs constantly dequeue ULTs from the various registered pools and execute them as they arrive. If there are no ULTs to execute, the ESs remain idle. The service provider transfers data from the origin through Mercury's bulk interface.

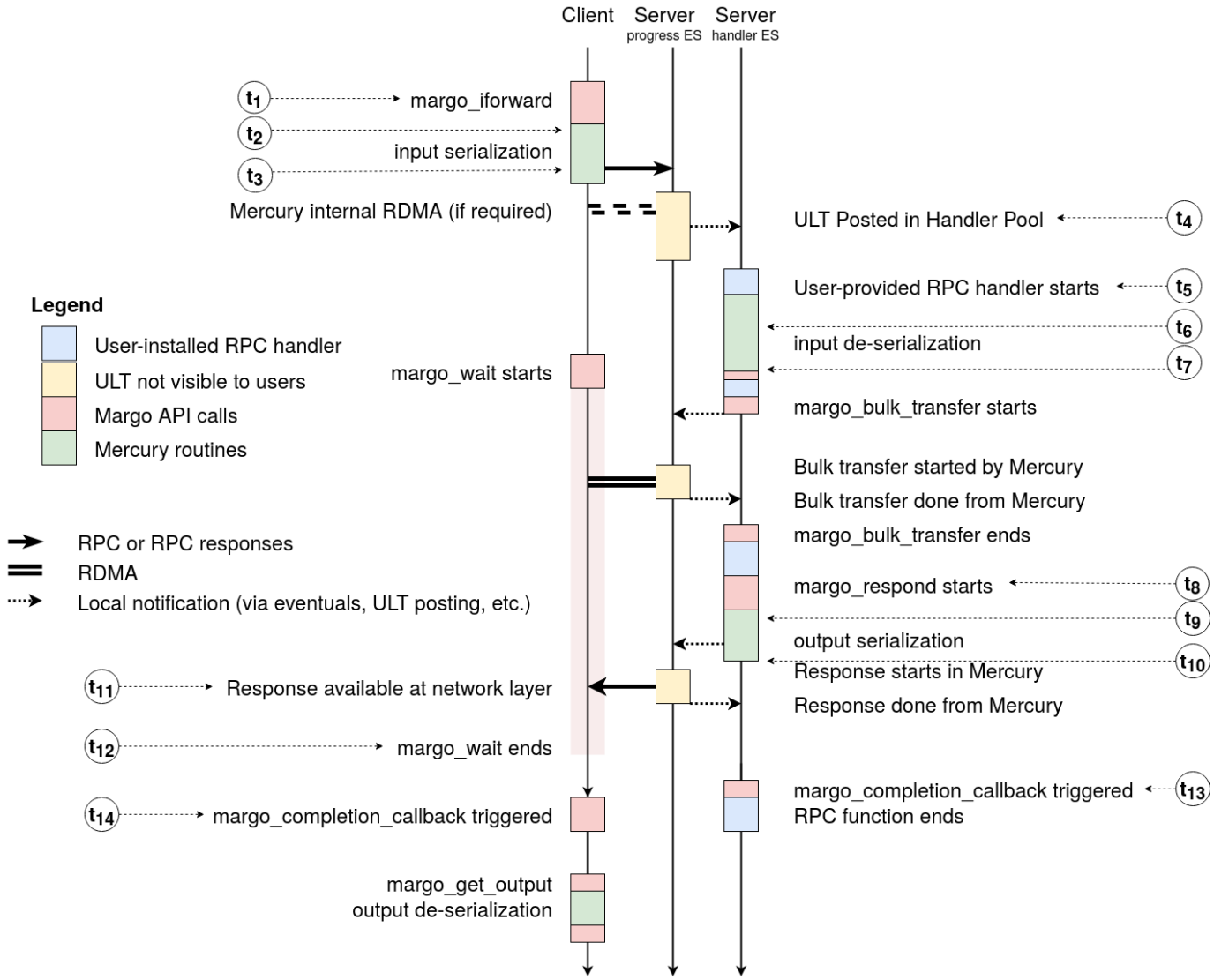


Fig. 2: Mochi RPC Execution Model

3) Issuance of a Response: The target provider generates a response ($\mathbf{t_8}$), and the output gets serialized inside Mercury ($\mathbf{t_9}$ to $\mathbf{t_{10}}$). The Margo library on the target registers a callback handler for the response. This callback handler is triggered by Mercury ($\mathbf{t_{13}}$) when the response has been sent to the origin.

4) Receipt of a Response: Once the response is available at the network layer on the origin ($\mathbf{t_{11}}$), at some later point in time the Mercury progress engine adds the completion callback for this request to the completion queue ($\mathbf{t_{12}}$). Then the callback for this request is triggered ($\mathbf{t_{14}}$).

IV. SYMBIOSYS

SYMBIOSYS is an *integrated* performance instrumentation, measurement, and analysis framework for HPC microservices. SYMBIOSYS can capture distributed callpath information; discover microservice request structure; and associate callpath information with software and hardware resources from the concurrency control, RPC library, and network layers depicted in Figure 1.

A. Distributed Callpath Profiling and Tracing

Common industry practice [13] to enable *portability* is to build these capabilities into the core communication libraries. Doing so has the dual advantage of operating seamlessly and not requiring end-user effort. *Margo* is the gateway to the core communication library (*Mercury*) and the runtime system (*Argobots*) managing execution on the compute node. Thus, Margo is the ideal software layer to host the performance measurement system.

1) Distributed Callpath Profiling: SYMBIOSYS tracks RPC callpath ancestries to present a callpath profile summary. This summary contains information about the total amount of time spent along different callpaths (or *callchains*) in the system. Each microservice instance keeps track of its callpath ancestry and forwards this information along the request path. This callpath information is maintained separately on the origin and target entities. Further, for every callpath, each origin entity making the call and each target entity servicing

the call are uniquely identified in the profile. During the invocation of an RPC call, the RPC call name is hashed into a 64-bit value and sent along with the RPC request. This 64-bit value denotes the *callpath ancestry* for the chain of RPCs. The Margo instance invoking the RPC stores this hash value inside Mercury at $\mathbf{t_1}$ (see Figure 2) and retrieves it from a callback argument at $\mathbf{t_{14}}$. At this point, the Margo instance measures the time it took for the RPC target to service the call. This is referred to as the *origin execution time*.

The delay between the receipt of the RPC call at $\mathbf{t_3}$ and the execution of the corresponding ULT at $\mathbf{t_4}$ is denoted as the *target ULT handler time* and is stored in a ULT-local key. The Margo instance receiving the RPC call at $\mathbf{t_3}$ unpacks the incoming RPC request and stores the 64-bit hash value in another key local to the ULT servicing that request. This is important because this ULT can make another RPC call as a side effect of the original one. If that is the case, the ULT needs to pass the callpath ancestry to downstream operations in order to maintain the correct chain of operations. The ULT first performs a 16-bit left shift of the 64-bit value representing callpath ancestry. It then hashes the name of the downstream call RPC and performs a logical `OR` operation such that the name of the downstream RPC call occupies the lowest 16 bits of the 64-bit value. The ULT then proceeds with making the RPC request. Currently, Margo can store RPC callpath lengths of up to four in the 64-bit hash value. When the ULT servicing the request on the target completes, it measures the time to service this request at $\mathbf{t_8}$. This is denoted as the *target ULT execution time*. The delay between the issuance of a response at $\mathbf{t_8}$ and the triggering of the corresponding completion callback at $\mathbf{t_{13}}$ is stored in a ULT-local key as the *target completion callback time*.

2) Distributed Request Tracing: While callpath profiling is useful for gathering a quick summary of service performance, information about individual requests is lost. Traces can span across multiple nodes (and processes) and contain rich performance information that can be analyzed for correlations of various performance metrics with time. The key idea lies in propagating request metadata (typically a unique request ID) through the system and then having a postprocessing system collect and stitch the individual trace events after execution has completed. Distributed tracing involves the generation of trace events at $\mathbf{t_1}$, $\mathbf{t_{14}}$ on the origin and $\mathbf{t_5}$, $\mathbf{t_8}$ on the target. The end-client (typically the user application) generates a globally unique *request ID* and propagates this ID along with a counter representing the *order* of the event in the individual trace. We implement Lamport's algorithm [25] to mitigate clock skew in the system. For every trace event generated, the current timestamp is stored along with a rich variety of performance data gathered from the RPC API, RPC library, and concurrency control layers.

B. Performance Data Exchange with the RPC Library

Associating higher-level callpaths with events from the RPC library and concurrency control layers is critical to form a complete picture of RPC performance. Typically, each software item in the RPC stack behaves like a black box, preventing the exchange of vital performance data that can aid in understanding performance and can present optimization opportunities. The MPI community has attempted to standardize the exchange of performance data through the MPI Tools Information Interface [16]. Our architecture for performance data exchange with the RPC library takes inspiration from these efforts.

TABLE I: Performance Variable Classes

PVAR Class	Description
STATE	Represents any one of a set of discrete states
COUNTER	Monotonically increasing value
TIMER	Interval event timer
LEVEL	Represents the utilization level of a resource
SIZE	Represents the size of a resource
HIGHWATERMARK	Highest recorded value
LOWWATERMARK	Lowest recorded value

1) Performance Variables: From the viewpoint of performance, several important events occur inside the Mercury communication library. We identify and implement several key performance variables (PVARs) in Mercury that capture these events. We introduce the concept of *PVAR classes* to represent the variety in the types of PVARs that can exist. For example, the PVAR class `STATE` is used to represent the current state of a particular Mercury resource or metric. Table I presents a list of PVAR classes currently available, and Table II lists some of the various PVARs that are currently implemented. The PVAR `num_posted_handles` represents a PVAR of the `STATE` class. Similarly, the PVAR class `COUNTER` represents a monotonically increasing value, and the PVAR classes `HIGHWATERMARK` and `LOWWATERMARK` denote the highest or lowest values recorded for a particular metric.

The other key concept we introduce is the notion of *PVAR bindings*. Many PVARs have a "global" scope and represent a counter or metric that has a wide temporal and spatial presence across the entire Mercury library. Such PVARs have a bind type `NO_OBJECT`. An example of a PVAR of this type is the `completion_queue_count` representing the current length of the Mercury completion queue. Other PVARs are short-lived and have a much narrower scope. Every RPC call is internally associated with a Mercury handle object. We introduce the bind type `HANDLE` to represent PVARs bound to internal Mercury handles. Once the particular RPC has completed, these PVARs go out of scope, and their values are lost forever. Examples of such PVARs include the timers representing the input and output serialization and deserialization times on the origin and the target.

2) Performance Tool Interface: We introduce a PVAR interface in Mercury to externally sample these Mercury PVARs. Briefly, the steps taken by an external tool to access and sample the PVARs are as follows:

- Initialize a PVAR session: Each tool querying the Mercury PVAR interface is assigned a unique `session_handle`.
- Query the interface for the supported PVARs: Once a session is initialized, the external tool queries the interface to gather

TABLE II: List of Available Performance Variables

PVAR Name	Description	PVAR Class	PVAR Binding
num_posted_handles	Number of currently posted RPC handles	LEVEL	NO_OBJECT
completion_queue_size	Number of events in Mercury's completion queue	STATE	NO_OBJECT
num_ofi_events_read	Number of OFI completion events last read	LEVEL	NO_OBJECT
num_rpcs_invoked	Number of RPCs invoked by instance	COUNTER	NO_OBJECT
internal_rdma_transfer_time	Time taken to transfer additional RPC metadata through RDMA	TIMER	HANDLE
input_serializaton_time	Time taken to serialize input on origin	TIMER	HANDLE
input_deserializaton_time	Time taken to de-serialize input on target	TIMER	HANDLE
origin_completion_callback_time	Delay between the arrival of RPC response and invocation of completion callback	TIMER	HANDLE

information about the number, type, binding, and count of all the PVARs exported.

- Allocate handles for PVARs: Once the relevant PVARs have been identified, the tool must allocate `pvar_handles` for the PVARs it wishes to read. The interface provides an API call for this purpose.
- Sample PVARs: At any point after the handle has been allocated, the external tool can sample (read) the value of the PVAR by providing the `pvar_handle` as input to the sampling API. If this PVAR is bound to a Mercury handle, the tool must provide the Mercury handle as input.
- Finalize the PVAR session: When the external tool is done sampling PVARs, it can free the allocated `pvar_handles` and finalize the PVAR session.

C. Integrating Data Sources

The microservice callpath measurements are used to orient and integrate performance data from the communication library. The Margo RPC API layer initializes a PVAR session with Mercury inside its initialization routine. At the same time, it also initializes all necessary PVAR handles. Figure 3 represents this interaction between the two layers of the Mochi stack. Margo samples the Argobots layer for the number of blocked and runnable tasks when generating a trace event. At these instrumentation points, it also samples memory usage and CPU utilization from the OS layer.

Although the OpenFabrics Network Interface [26] specification does not allow us to read the instantaneous number of events in its completion queue, we can gather a sense of the size of the completion queue by sampling the *number of actual events last read* in the form of the `num_ofi_events_read` Mercury PVAR at $\mathbf{t_{14}}$. When Mercury PVAR profiling is enabled, it samples the `num_ofi_events_read` PVAR and adds this data to the trace record. At the origin, Margo reads the PVARs holding the *origin callback completion time* and *input serialization time* when measuring at $\mathbf{t_{14}}$. Similarly, at the target, the PVARs representing the *target internal RDMA transfer time*, *input deserialization rime*, and *output serialization time* for the particular RPC call are sampled when measuring at $\mathbf{t_{13}}$. These trace events and profile data are consolidated, aggregated, and presented for visualization at the end of the execution.

V. Profiling Case Studies

In this section, we present scenarios describing the usage of SYMBIOSYS to aid with the key performance analysis activities discussed in Section I.

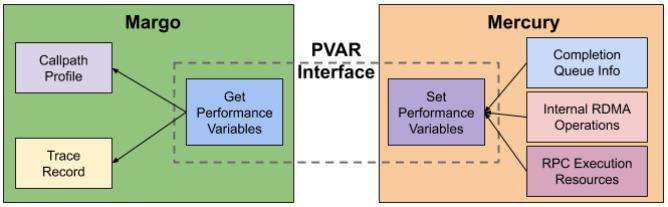


Fig. 3: PVAR Interface Between Margo and Mercury

TABLE III: Combining Instrumentation Strategies

Interval Name	Interval Start	Interval End	Instrumentation Strategy
Origin Execution Time	$\mathbf{t_1}$	$\mathbf{t_{14}}$	ULT-local key
Input Serialization Time	$\mathbf{t_2}$	$\mathbf{t_3}$	Mercury PVAR
Target Internal RDMA Transfer Time	$\mathbf{t_3}$	$\mathbf{t_4}$	Mercury PVAR
Target ULT Handler Time	$\mathbf{t_4}$	$\mathbf{t_5}$	ULT-local key
Input Deserialization Time	$\mathbf{t_6}$	$\mathbf{t_7}$	Mercury PVAR
Target ULT Execution Time (exclusive)	$\mathbf{t_5}$	$\mathbf{t_8}$	ULT-local key
Output Serialization Time	$\mathbf{t_9}$	$\mathbf{t_{10}}$	Mercury PVAR
Target ULT Completion Callback Time	$\mathbf{t_8}$	$\mathbf{t_{13}}$	ULT-local key
Origin Completion Callback Time	$\mathbf{t_{12}}$	$\mathbf{t_{14}}$	Mercury PVAR

A. Mobject: Identifying Dominant Microservice Dependencies

1) Background: Mobject [3] is a distributed object storage service that exposes a subset of the RADOS [27] API to support concurrent, noncontiguous writes of objects. Each Mobject *provider node* (service provider process) hosts three types of providers—a Mobject sequencer provider, a BAKE provider, and an SDSKV provider. The Mobject sequencer provider translates the RADOS operations into the underlying BAKE and SDSKV operations. BAKE is used to store object data through RDMA transfers between BAKE and client memory. SDSKV is used to store metadata information. We note that the Mobject provider is the client-facing provider and control always goes back to the Mobject provider after the BAKE and SDSKV operations complete. Figure 4 depicts this structure.

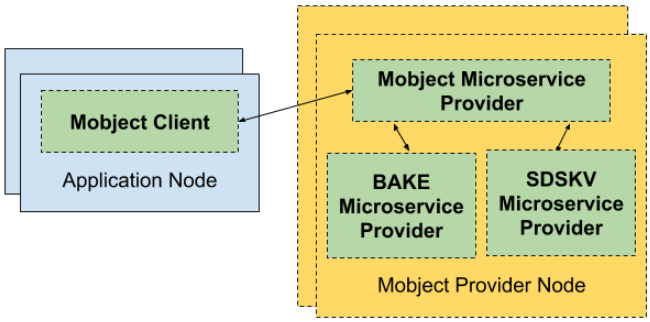


Fig. 4: Mobject Illustration

2) Identifying Dominant Microservice Dependencies: Identifying dominant microservice dependencies or callpaths is a crucial first step in performance analysis. It isolates resource-intensive portions of the workload at a high level and helps determine where to focus attention for further analysis and optimization. Recall that RPC callpaths can cross process boundaries. The SYMBIOSYS profile summary script ingests all the profiles and performs a global analysis to identify origin-target pairs for each callpath. The script summarizes and sorts callpaths by cumulative end-to-end request latency to identify the most dominant ones. For each of these dominant callpaths, the SYMBIOSYS profile summary script generates call count distributions for all the participating origin and target entities. These distributed callpaths are used as a pivot around which lower-level communication library data and tasking library queue information is oriented. The results from this profile summary can be used as a starting point for a more detailed performance study.

We employ a single Mobject service provider node and 10 ior [28] clients colocated on the same physical node. The ior benchmark has been modified to use Mobject for reading and writing objects. For this setup, Figure 6 depicts the top 5 most dominant callpaths by cumulative end-to-end request latency. mobject_read_op is the most expensive Mobject API operation overall. The profile suggests that the mobject_read_op⟶sdskv_list_keyvals_rpc is the dominant component of the top-level mobject_read_op API call. Note that for each of these callpaths, the breakdown of the individual steps for each callpath such as the *input serialization time*, *internal RDMA transfer time*, and *target handler time* is shown. For this application setup, these individual steps occupy a negligible time in the profile as compared with the time taken to execute the request on the target.

3) Discovering Individual Request Structure: Once the dominant callpaths have been identified, developers may be interested in tracing the path of individual requests to identify the exact microservice operations getting invoked as a result of a higher-level operation. This sort of analysis is especially useful for identifying root causes for performance anomalies resulting from structural abnormalities in service requests.

For the same ior and Mobject setup described previously, Figure 5 represents the trace visualization for a single invocation of the mobject_write_op callpath. It discovers 12 discrete SDSKV and BAKE microservice calls (e.g., mobject_write_op⟶sdskv_get_rpc, mobject_write_op⟶bake_persist_rpc) that make up the higher-level mobject_write_op request. Each of these 12 discrete microservice calls has its own profiling data, so the user can break down where time is spent and reason about request performance. Without this trace data, the internal structure of the request is completely opaque to the user. SYMBIOSYS enables this Gantt chart visualization through an adapter module that "stitches" the events with a common requestID from different processes into a Zipkin [14] JSON trace file.

B. Sonata: Mapping Resource Usage to Individual Steps

1) Background: Sonata is a microservice for remotely accessing and storing JSON objects. It is based on an UnQLite [29] database and offers the ability to remotely run analysis on the stored JSON objects through Jx9 scripts. While the BAKE microservice is optimized for large blobs of unstructured data and the SDKSV microservice is optimized for small key-value pairs, Sonata is instead optimized for document storage, especially if there is a need to perform complex, in-place queries on these documents.

2) Mapping Resource Usage to Individual Steps: The JSON document to be stored is transferred as RPC metadata. However, if Mercury's *eager buffer* overflows, the additional RPC metadata is transferred through an internal Mercury RDMA operation (between $\mathbf{t_3}$ and $\mathbf{t_4}$). With a large RPC metadata transfer, it is imperative to understand the contributions of (de)serialization and internal RDMA transfer operations to the RPC execution time. We execute a simple Sonata benchmark with one target and one origin entity on separate compute nodes. The benchmark repeatedly invokes the `sonata_store_multi_json` API call to store a fixed-length JSON record array in a set of batches. The *batch size* benchmark parameter determines the size of RPC metadata and the total number of RPC calls. Figure 7 depicts the breakdown of the cumulative RPC execution time on the target for a JSON record array of 50,000 entries and a batch size of 5,000. While the *target internal RDMA transfer time* is relatively low, the time to de-serialize the input accounts for 27% of the overall execution time on the target.

C. HEPnOS: Observing Resource Saturation and Identifying a Better Service Configuration

1) Background: HEPnOS [3] is a Mochi storage service designed for high energy physics experiments and simulations at Fermilab. Data in HEPnOS is arranged in a hierarchy of datasets, runs, subruns, and events. Events correspond to serialized C++ data objects. HEPnOS distributes both object data and metadata. Each HEPnOS service provider node hosts one BAKE provider to store object data and one SDSKV provider to store object metadata. Figure 8 depicts the structure of the HEPnOS service. Client processes (physics simulation) contact the BAKE and SDSKV providers directly through a C++ client API.

The production HEPnOS client application is a workflow comprising multiple steps. In this study, we focus on the "data-loader" step in which particle event data is loaded into the system. We chose this step for evaluation since it is the most mature and has the fewest external dependencies. The data-loader reads HDF5 files containing physics simulation event data from a conventional file system. It then writes this event data into the HEPnOS data service. A SYMBIOSYS profile of the data-loader client application suggests that sdskv_put_packed is the only dominant RPC callpath generated, regardless of scale. The sdskv_put_packed API is used to store a key-value list (event data in HEPnOS) to a backend database (`LevelDB`, `BerkeleyDB`, or `map`). This

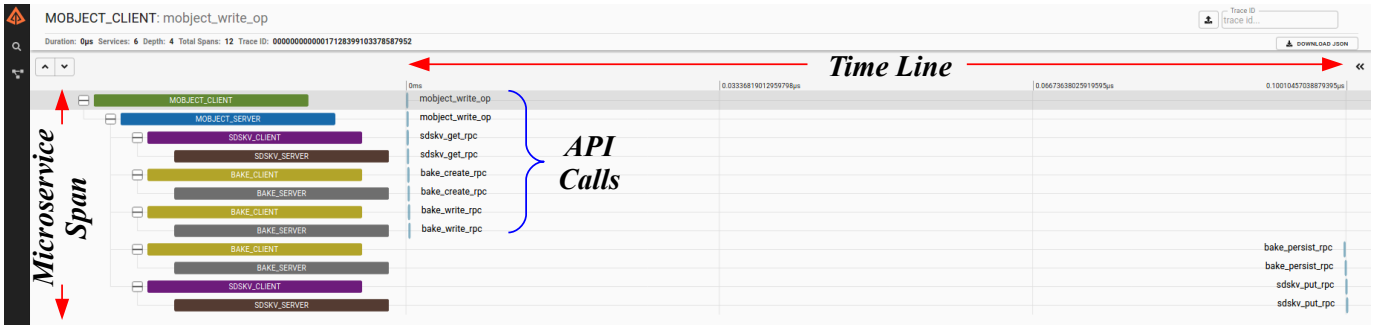


Fig. 5: ior + Mobject: OpenZipkin [14] Trace Visualization Depicting Discrete Steps for a Single mobject_write_op Request

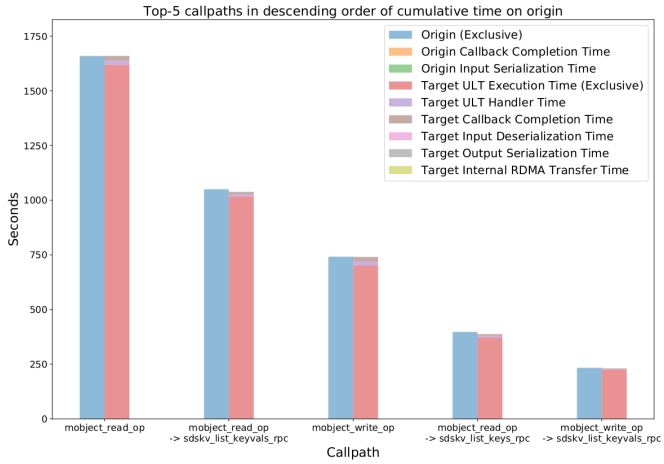


Fig. 6: ior + Mobject: Identifying the Dominant Callpaths

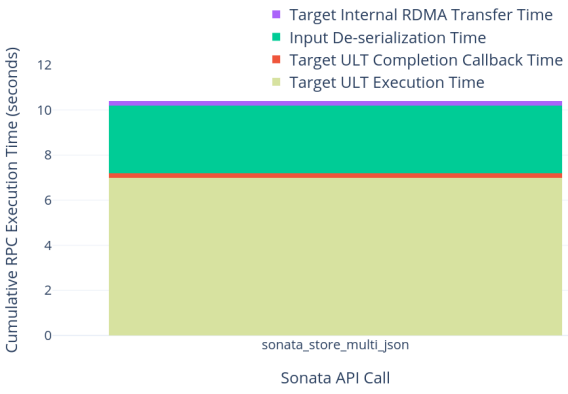


Fig. 7: Sonata: Mapping Execution Time to Individual Steps

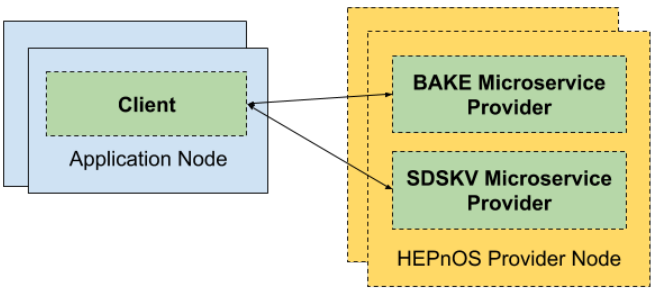


Fig. 8: HEPnOS Illustration

RPC call typically results in the target issuing a bulk data transfer (refer to Figure 2) to pull in the key-value content. The focus of our study is to analyze the performance of the RPC component of the HEPnOS data-loader application. We employ SYMBIOSYS to study the root causes of poorly performing HEPnOS configurations and to determine better service configurations to improve performance. Table IV enlists the various service configurations that are a part of this study.

2) Too Few Execution Streams: It is difficult to estimate beforehand the optimal number of target Argobots execution streams (ESs) required for a given workload. However, by observing delays in the execution pathway of the RPC call, one can determine when these resources saturate, thereby identifying a poorly performing service configuration. Recall that on the target, a new ULT is spawned at $\mathbf{t_4}$ for every incoming RPC request. We define the delay between events $\mathbf{t_4}$ and $\mathbf{t_5}$ in Figure 2 as the *target handler time*. This is the time spent by a newly spawned ULT in the Argobots handler pool before an ES is available to pick it up for execution. When the target is overloaded with RPC requests and lacks the execution resources to dispatch the corresponding ULTs promptly, the *target handler time* can contribute significantly to the overall request latency and worsen performance.

Figure 9 demonstrates that $\mathbf{C_1}$ suffers from a lack of execution resources on the target. Avoidable delays inside the Argobots handler pool (target handler time) account for 26.6% of the total RPC execution time.

A Better Service Configuration: $\mathbf{C_2}$ remediates this by adding 15 additional execution streams (threads). Overall cumulative RPC execution time improves by 53.3%, with the target handler time contributing 14% to the overall time.

3) Too Many Databases: Each target provider node employs several databases to parallelize the writing of HEPnOS event data. Specifically, in this study the target employs a `map` backend. For the sdskv_put_packed RPC, the origin implements a hashing scheme using the key and the total number of databases to identify the target database ID. Everything else being equal, the greater the number of target databases, the greater is the number of RPCs generated. Since the `map` backend is not capable of parallel insertions, employing too many target databases can create a flood of RPCs and cause write serialization during bursty behavior. Configuration $\mathbf{C_2}$

TABLE IV: HEPnOS: Service Configurations

Configuration	Total Clients; Clients Per Node	Total Servers; Servers Per Node	Batch Size	Threads (ESs)	Databases	Client Progress Thread?	OFI_max_events
$\mathbf{C_1}$	32; 16	4; 2	1024	5	32	✗	16
$\mathbf{C_2}$	32; 16	4; 2	1024	20	32	✗	16
$\mathbf{C_3}$	32; 16	4; 2	1024	20	8	✗	16
$\mathbf{C_4}$	2; 1	4; 2	1024	16	8	✗	16
$\mathbf{C_5}$	2; 1	4; 2	1	16	8	✗	16
$\mathbf{C_6}$	2; 1	4; 2	1	16	8	✗	64
$\mathbf{C_7}$	2; 1	4; 2	1	16	8	✓	64

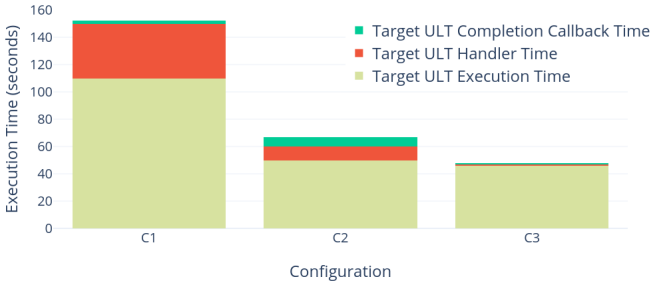
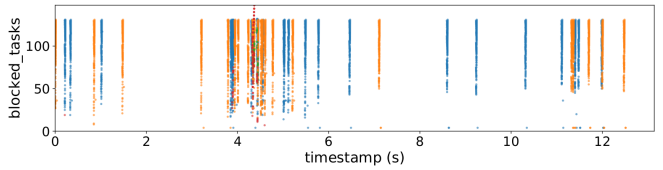
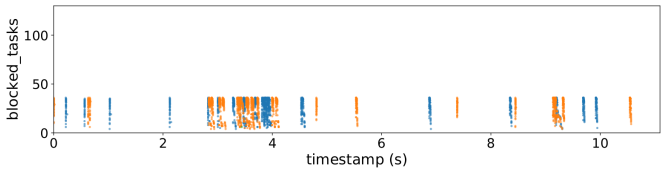


Fig. 9: HEPnOS: Cumulative Target RPC Execution Time for sdskv_put_packed



(a) C2



(b) C3

Fig. 10: HEPnOS: Sampling Blocked Tasks from Argobots for sdskv_put_packed

suffers from this problem. The x-axis in Figure 10 denotes the timestamp of when the request began execution on the target at $\mathbf{t_4}$ (Figure 2), and the y-axis represents the total number of blocked ULTs sampled from Argobots at this time. Each colored dot represents a single request. Requests executed at different targets are represented by different colors. Figure 10a depicts this serialization problem during bursty behavior with configuration $\mathbf{C_2}$. This pattern of vertical lines generated by requests that arrived at the same time but complete in quick succession (as opposed to simultaneously) is indicative of serialization on a backend resource.

A Better Service Configuration: Counterintuitively, RPC performance in this situation improves when reducing the number of databases. RPC performance in $\mathbf{C_3}$ is better than $\mathbf{C_2}$ by 28.5%. Figure 10b also demonstrates that the severity of serialization in $\mathbf{C_3}$ is much reduced as compared with $\mathbf{C_2}$. The reduced number of RPCs generated with $\mathbf{C_3}$ also has the effect of lowering the target handler time and the target completion callback time—the ULTs are being processed quickly without introducing unwanted delays.

4) Effect of a Low Batch Size on Client Progress: HEPnOS clients batch key-value pairs containing HEPnOS event data to improve RPC throughput when generating an sdskv_put_packed request. A batch size of 1,024 ($\mathbf{C_4}$) is roughly 475 times more performant than a batch size of 1 ($\mathbf{C_5}$). Figure 11 suggests that instrumentation from the RPC API and RPC library layers is insufficient to capture all components of the cumulative RPC execution time for $\mathbf{C_5}$ (the unaccounted portion is depicted by the blue color in Figure 11). We consider the question of identifying this gap in instrumentation. We also seek to improve RPC performance when low batch size is an inherent property of the application setup and cannot be controlled.

The HEPnOS data-loader client employs a Mercury *progress ULT* to progress RPC communications within the client process. This progress ULT has two important tasks: (1) read the OFI events containing notifications of RPC responses from the network abstraction layer and add the corresponding completion callbacks to the completion callback queue and (2) trigger completion callbacks from the completion callback queue. In order to prevent context switching overheads, by default, this progress ULT is executed within the context of the main Argobots execution stream that also executes the ULTs issuing RPC requests. When the batch size is low, the progress ULT can compete for CPU resources with the ULTs issuing RPC requests. As a result, the completion callback queue or the OFI event queue can clog up and introduce unwanted delays. Every time it is scheduled to execute, the progress ULT reads up to a maximum of `OFI_max_events` events from the OFI interface. `OFI_max_events` has a default

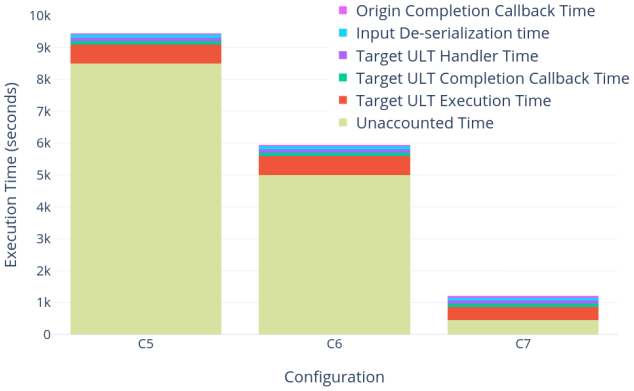


Fig. 11: HEPnOS: Unaccounted Component of RPC Execution

user-defined value of 16, set inside the Mercury library. Figure 12a depicts a sample of the `num_ofi_events_read` PVAR in configuration $\mathbf{C_4}$ where the batch size is optimal. In this configuration, the `OFI_max_events` threshold is never breached, implying that the OFI completion queue is emptied at regular intervals. Figure 12b depicts a sample of the same PVAR in configuration $\mathbf{C_5}$ where the batch size is low. Clearly the number of OFI events read consistently breaches the threshold value of 16, suggesting that the completion queue is backed up.

A Better Service Configuration: Increasing the `OFI_max_events` threshold from 16 to 64 with $\mathbf{C_6}$ improves RPC performance by over 40% while reducing the unaccounted time by 47%. The performance data gathered from various layers in the software stack indicate that employing a separate, dedicated execution stream for the client's progress thread is likely to improve performance. Figure 11 confirms that RPC performance for $\mathbf{C_7}$ improves by a further 75% and the unaccounted time reduces by a further 90% as compared with $\mathbf{C_6}$. From Figure 12d, we conclude that the OFI event queue is no longer backed up.

VI. Overhead Evaluation

We present the overheads in employing the SYMBIOSYS framework for performance measurement and analysis of the HEPnOS storage service. We pay special attention to the process of separating the overheads of adding instrumentation, making the performance measurement, and analyzing the generated data using the performance analysis scripts we have developed.

A. Setup

1) Hardware: All the experiments were conducted on the Theta [1] system at the Argonne Leadership Computing Facility (ALCF). Theta, a CrayXC40 system, hosts 4,393 Intel KNL compute nodes, each of which hosts 64 processing cores. We used the Intel KNL processors for all our experiments.

2) Software: All of our experiments were conducted using the HEPnOS storage service along with a data-loader client application setup. The Mochi components were installed using the Spack [30] package manager. We employed 128 nodes for our large-scale study. We used 32 HEPnOS service provider processes spread evenly over 16 nodes. Each service provider process was assigned 30 threads and 16 databases for storing HEPnOS events. We employed 224 data-loader clients spread over 112 nodes. The batch size was set to 8,192, and a separate client progress thread was not employed for our experiments.

B. Results

We measured the execution time of the data-loader application as the metric to compare the instrumentation and measurement overheads in SYMBIOSYS. We used the following terms to denote the various stages of the process:

- Baseline: This is the baseline execution time with instrumentation and measurement disabled.
- Stage 1: This is the execution time with instrumentation turned on while no measurements are being made. In SYMBIOSYS, this corresponds to the addition of RPC callpath and trace ID information in the RPC request.
- Stage 2: Callpath profiling, tracing, and system statistic sampling are enabled, but Mercury PVAR collection is disabled.
- Full Support: Callpath profiling, tracing, and system statistic sampling are enabled. Mercury PVAR collection is turned on, and the PVAR data is integrated on the fly with the callpath profiles.

[1]https://www.alcf.anl.gov/support-center/theta

TABLE V: HEPnOS: Analysis Overheads

Profile Summary (s)	Trace Summary (s)	System Statistics Summary (s)
35.1	481.1	73.4

Figure 13 depicts the overheads involved in enabling various stages of performance measurement using the HEPnOS setup. Each entry in the table is the average of 5 execution times. At large scale, the SYMBIOSYS tracing system collected a total of 1 million samples. Even at this scale, enabling profiling and tracing led to minimal overheads that were indistinguishable from the run-to-run variation in execution time. Table V presents the time taken to analyze the collected performance data and generate visual plots. The profile and system summary analysis scripts completed in a short amount of time, while the trace summary script took a longer time to run when applied to the large-scale performance data.

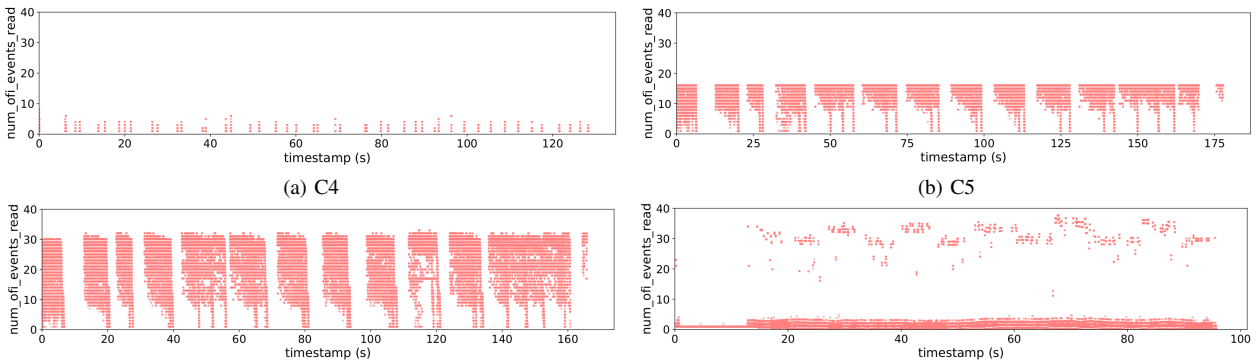
VII. Conclusion and Future Work

This paper presented the SYMBIOSYS integrated performance measurement and analysis infrastructure designed for HPC data services. We have enabled the effective and portable profiling and analysis of HPC microservices by tracking the RPC callpath ancestry. By integrating data from the RPC communication library through a data-exchange strategy, SYMBIOSYS was able to correctly attribute low-level events and resource usage levels with higher-level interactions between service entities. Further, we demonstrated that the performance infrastructure was capable of operating with a low overhead at scale.

Our future research will focus on in situ approaches for dynamic reconfigurability based on knowledge learned from the experimental results and application studies. We envision the creation of policy-driven mechanisms whereby rules governing response to poor performance behavior can be formulated and applied based on performance monitoring and models.

VIII. Acknowledgement

We thank Kevin Huck at the University of Oregon for his help in exploring the use of the APEX [10] monitoring framework for Mochi performance analysis. This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.



(a) C4 (b) C5

(c) C6 (d) C7

Fig. 12: HEPnOS: Sampling OFI Events Read from Network Abstraction Layer for sdskv_put_packed

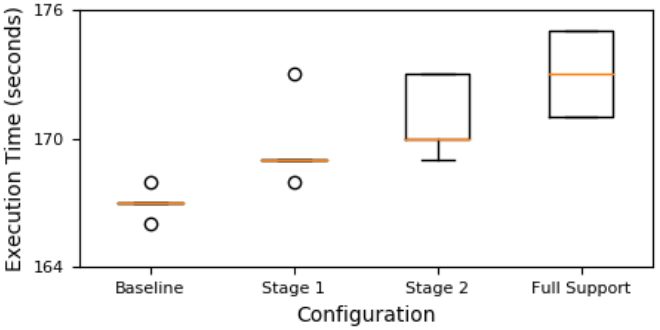


Fig. 13: HEPnOS: Measurement Overheads

REFERENCES

[1] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof *et al.*, "Candle/supervisor: A workflow framework for machine learning applied to cancer research," *BMC bioinformatics*, vol. 19, no. 18, p. 491, 2018.

[2] P. M. Kasson and S. Jha, "Adaptive ensemble simulations of biomolecules," *Current opinion in structural biology*, vol. 52, pp. 87–94, 2018.

[3] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms *et al.*, "Mochi: Composing data services for high-performance computing environments," *JCST*, vol. 35, no. 1, pp. 121–144, 2020.

[4] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[5] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes *et al.*, "Gekkofs-a temporary distributed file system for hpc applications," in *CLUSTER*. IEEE, 2018, pp. 319–324.

[6] S. S. Shende and A. D. Malony, "The tau parallel performance system," *IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.

[7] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for HPC*. Springer, 2012, pp. 79–91.

[8] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre *et al.*, "Caliper: performance introspection for hpc software stacks," in *SC*. IEEE Press, 2016, p. 47.

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

[10] K. Huck, S. Shende, A. Malony, H. Kaiser, A. Porterfield, R. Fowler *et al.*, "An early prototype of an autonomic performance environment for exascale," in *Proceedings of the 3rd Int. Workshop on Runtime and Operating Systems for Supercomputers*, 2013, pp. 1–8.

[11] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken *et al.*, "Diagnosing performance changes by comparing request flows." in *NSDI*, vol. 5, 2011, pp. 1–1.

[12] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? key design insights from years of practical experience," *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14*, 2014.

[13] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[14] "Openzipkin," http://zipkin.io.

[15] "Jaeger tracing," https://www.jaegertracing.io.

[16] M. P. Forum, "Mpi: A message-passing interface standard," 1994.

[17] T. Islam, K. Mohror, and M. Schulz, "Exploring the capabilities of the new mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 91–96.

[18] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. Panda, "Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau," *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[19] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich *et al.*, "Ompt: An openmp tools application programming interface for performance analysis," in *Int. Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[20] K. A. Huck, A. D. Malony, S. Shende, and D. W. Jacobsen, "Integrated measurement for cross-platform openmp performance analysis," in *Int. Workshop on OpenMP*. Springer, 2014, pp. 146–160.

[21] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "Papi software-defined events for in-depth performance analysis," *IJHPCA*, vol. 33, no. 6, pp. 1113–1127, 2019.

[22] P. Carns, J. Jenkins, C. D. Cranor, S. Atchley, S. Seo, S. Snyder *et al.*, "Enabling {NVM} for data-intensive scientific services," in *INFLOW*, 2016.

[23] S. Seo *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE TPDS*, 2017.

[24] J. Soumagne *et al.*, "Mercury: Enabling remote procedure call for high-performance computing," in *CLUSTER*, 2013.

[25] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[26] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 34–39.

[27] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *PDSW*, 2007, pp. 35–44.

[28] W. Loewe, T. McLarty, and C. Morrone, "Ior benchmark," 2012.

[29] G. Douglas and R. Lawrence, "Littled: a sql database for sensor nodes and embedded applications," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 827–832.

[30] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski *et al.*, "The spack package manager: bringing order to hpc software chaos," in *SC*. IEEE, 2015, pp. 1–12.