Towards Empirical Roofline Modeling of Distributed Data Services: Mapping the Boundaries of RPC Throughput

Philip Carns
carns@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Matthieu Dorier
mdorier@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Rob Latham
robl@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Shane Snyder
ssnyder@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Amal Gueroudji
agueroudji@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Seth Ockerman
sockerman@wisc.edu
University of Wisconsin-Madison
Madison, Wisconsin, USA

Jerome Soumagne
jerome.soumagne@hpe.com
Hewlett Packard Enterprise
Spring, Texas, USA

Dong Dai
dai@udel.edu
University of Delaware
Newark, Delaware, USA

Robert Ross
rross@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Abstract

The scientific computing community relies on distributed data services to augment file systems and decouple data management functionality from applications. These services may be native to high-performance computing or adapted from cloud environments, and they encompass diverse use cases such as domain-specific indexing, in situ analytics, AI data orchestration, and special-purpose file systems. They unlock new levels of performance and productivity but also introduce new tuning challenges. In particular, how do practitioners assess performance, select deployment footprints, and ensure that services reach their full potential? Roofline models could address these challenges by setting practical performance expectations and providing guidance to achieve them.

This paper outlines initial steps toward establishing an empirical roofline modeling methodology for distributed data services, focusing exclusively on network characteristics as a proof of concept. We first explore how to maximize performance on modern platforms. We next propose an adaptation of the classic roofline model and a methodology for collecting model parameters using the Mochi data service framework. We then survey four large-scale HPC systems and construct a roofline model for each of them. We evaluate the models and identify next steps toward a comprehensive roofline modeling framework for distributed data services.

CCS Concepts

• **Information systems → Distributed storage**; • **General and reference** → *Measurement*; *Performance*.

Keywords

Mochi, Distributed Data Services, Storage, High-Performance Computing, Roofline models, Mercury, Slingshot

ACM Reference Format:
Philip Carns, Matthieu Dorier, Rob Latham, Shane Snyder, Amal Gueroudji, Seth Ockerman, Jerome Soumagne, Dong Dai, and Robert Ross. 2025. Towards Empirical Roofline Modeling of Distributed Data Services: Mapping the Boundaries of RPC Throughput. In *Proceedings of Cray User Group (CUG '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The scientific computing community relies on distributed data services to augment file systems and decouple data management functionality from applications. These services may be native to high-performance computing (HPC) or adapted from cloud environments, and they encompass diverse use cases such as domain-specific indexing, in situ analytics, AI data orchestration, and special-purpose file systems. This rich variety of services offers unprecedented potential for innovation and specialization to improve performance and increase productivity. However, that inherent diversity also complicates efforts to assess performance in relation to the baseline capabilities of the platform. Analytical models based on hardware specifications do not capture the complex interactions between network, disk, CPU, and memory at scale, nor do they account for software factors such as data encoding, concurrency management, and interfaces. Empirical models based on real-world service benchmarks provide a more realistic representation of service capabilities [29], but we need methods that are flexible enough to represent a full range of data services and deployment possibilities as the community embraces specialized user-level services.

This situation necessitates a new approach: a generalized method for data service roofline modeling, based on empirical benchmarks, that can be adapted to different platforms, deployment footprints, and performance metrics. Roofline modeling [24] is a compelling concept; roofline models are lightweight, familiar to HPC practitioners, intuitive to construct, and easy to visually interpret. However, the original roofline concept was intended for assessment of computational performance, and hence we must carefully adapt it for use in distributed data services. While many potential applications for such a model exist, we are most interested here in the following:

- Evaluating the performance of novel distributed data services to determine whether performance optimization is warranted
- Selecting the optimal deployment configuration for a given workload and service combination
- Demonstrating best practices, and providing a reference point for optimization of distributed data services

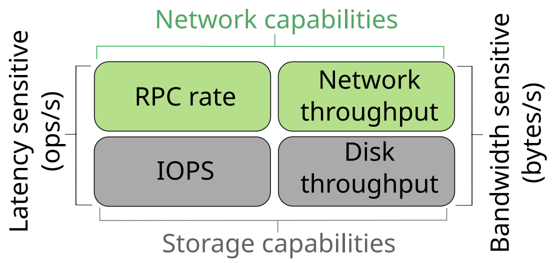


Figure 1: Summary of key data service performance metrics. The remote procedure call (RPC) rate refers to the rate at which servers can respond to requests, while IOPS refers to the rate at which the storage system can service small I/O operations. Network and storage bandwidth refer to data transfer speeds. In this study we focus on the network metrics highlighted in green.

Each target use case requires a quantitative metric to assess its performance. However, no one shared I/O performance metric is most relevant to all data services and use cases (e.g., some are latency sensitive, some require high bandwidth, some use storage devices). We therefore seek to develop a roofline framework that can be adapted to a variety of metrics as shown in Figure 1. We focus on network performance metrics in this study as initial building blocks. The first is the remote procedure call (RPC) rate; this is the aggregate rate at which services can respond to distinct application requests. The second is the network bandwidth, which is aggregate rate at which data payloads can be transferred to and from the service. Our goal is to develop an intuitive roofline model methodology for distributed data services that can be applied to different performance metrics depending on the scenario.

We investigate three key challenges in the construction of roofline benchmarks for distributed data services on HPC systems:

(1) What techniques are needed to maximize network metrics on current-generation HPC systems?
(2) What visual representation is most appropriate for a data service roofline model?
(3) Can we generalize the capture of model parameters so that it can be easily repeated on diverse platforms?

The remainder of this paper is organized as follows. In Section 2 we summarize the related work that we build on. In Section 3 we provide background on the Mochi data service framework and experimental platforms. In Section 4 we describe our methodology for maximizing performance and constructing roofline models. In Section 5 we perform a survey that applies our roofline approach to four leadership-class computing systems. We discuss our findings in Section 6 and our plans for future work in Section 7.

2 Related work

Our work was inspired in large part by the exploration of large-scale file system characteristics in previous work by Lang et al. [16]. In their work, component measurements were used to identify platform performance limitations at scale and assess the degree to which a parallel file system (PVFS in this case) could exploit them. Subsequent work by Carns et al. [6] explored the limitations of component-level microbenchmarks for assessment of platform capabilities, which motivated the development of parameterized microservices to capture interactions between components in end-to-end data service environments.

The fundamental concept of roofline modeling was introduced by Williams et al. [24]. It provides straightforward insight into the performance and bottlenecks of computational kernels on sophisticated multicore microprocessor architectures. Their work had a lasting impact on performance analysis, both in practice and in research, and has been extended in various ways over time. For instance, Ilic et al. [14] proposed a cache-aware roofline model that plots multiple rooflines based on the cache levels used. Yang et al. extended the model to a hierarchical version that incorporated GPUs [26]. Yang et al. also developed the Empirical Roofline Toolkit to collect system metrics and empirically generate roofline models [25]. Roofline models have also been extended to distributed systems [8] and scientific workflows [9]. More recently, the roofline model has been applied to analyze bottlenecks in large language model training and inference [27]. Our work builds upon the original concept of the roofline model but differs from many of its extensions by focusing on assessing the I/O performance of distributed data services.

Zhaobin et al. extended the roofline concept to assess the I/O performance of large-scale storage systems [28, 29]. Similarly to our approach, they use empirical measurements to establish model parameters. However, unlike our approach, they focus on file systems that can be assessed with IOR [22] and Darshan [5] and static deployments in which the number of servers is not tunable. In contrast, our work investigates user-level data services that do not necessarily present file system semantics and can be deployed at arbitrary scales.

Liem et al. [19] also explored intuitive visual representations of parallel file system capabilities, expressing performance as a bounding box for user expectations based on empirical IO500 [15] measurements. Similarly to a roofline model, their approach aids users in assessing if application performance falls within anticipated ranges on a given platform.

3 Background

In this section we provide background information about the Mochi data service framework and detail the experimental platforms used in our study.

Table 1: Experimental platform compute node characteristics

	CPU sockets	CPU cores (total)	CPU cores (usable)	NICs	NIC connectivity
Aurora (ALCF)	2	104	102	8	CPU
Polaris (ALCF)	1	32	32	2	CPU
Frontier (OLCF)	1	64	56	4	GPU
Perlmutter (NERSC)	1	64	64	4	CPU

3.1 Mochi data services

We use the Mochi data service framework [20] as the basis for our study. Mochi is both a software framework and a methodology for the rapid construction of specialized data services. The key concept underpinning Mochi is *composability*; Mochi provides a collection of reusable components (microservices and libraries) that can be combined as needed for different data management use cases. It has been used to develop a wide variety of data services [4] and is already optimized for use on HPC systems, making it a representative framework for a broad range of distributed services. Its inherent composability also makes it more adept than monolithic service implementations at isolating distinct factors in data service performance.

Three Mochi components are especially critical to this study. The first is the **Mercury** RPC library [23]. Mercury underpins all network communication in Mochi. It operates atop high-performance, RDMA-capable network libraries such as `libfabric` and `UCX` and provides asynchronous generalized remote procedure call primitives. It also provides explicit bulk transfer primitives for efficient abstraction of network RDMA functionality. The Mochi framework combines the Mercury RPC library and the Argobots user-level threading package [21] into a coherent data service runtime system via the Mochi Margo component [7].

The second key component is **Mochi Bedrock** [10]. Bedrock is responsible for configuration and bootstrapping in Mochi; it provides a method to launch services according to a JSON-formatted configuration specification. It also enables subsequent query and modification of that configuration at runtime. We use Bedrock to standardize our benchmark configurations and capture a complete view of all runtime parameters.

Third, we rely on **Mochi Quintain**[1] to generate and execute parameterized synthetic workloads and isolate data service performance characteristics. Quintain is implemented as a microservice that can be embedded in any other Mochi service to provide a "self-test" capability. All benchmark results presented in this paper, unless otherwise noted, were performed using synthetic Quintain workloads.

3.2 Experimental platforms

We use four large-scale HPC systems located at the Argonne Leadership Computing Facility (ALCF), the Oak Ridge Leadership Computing Facility (OLCF) and the National Energy Research Scientific Computing Center (NERSC) as experimental platforms for this study. They are summarized in Table 1.[2] We focus on compute nodes characteristics because we intend to investigate the performance of user-level data services (i.e., those that execute on demand within the context of a user's job allocation).

The GPU configuration is a critical characteristic for computational science applications, but we focus on services that execute on the host CPU. We therefore omit GPU details except to note that Frontier employs a unique node topology in which network cards have a direct PCI connection to GPUs rather than CPUs. We also note that systems that report fewer usable cores than total cores do so because they employ core specialization in which a subset of cores are reserved for system tasks and are not available to end users.[3]

These systems share several factors that influence data service performance. The first is that they use HPE Slingshot networks. Each compute node is equipped with multiple network interface cards (NICs), and each NIC is connected to the same unified fabric on a given system. Second, the compute nodes contain numerous compute resources (e.g., these systems follow a heavyweight compute node design philosophy). These factors suggest that data service performance will hinge on a service's ability to efficiently leverage a complex combination of compute cores and network links simultaneously.

The Mochi software stack was configured identically on each system using the following versions of key components: mochi-bedrock 0.15.2, mochi-quintain 0.6.0, mochi-margo 0.19.1, mochi-plumber 0.1.4, and mercury 2.4.0rc5. All software was compiled using the latest version of the `gcc` compiler available on each system. Default values were used for all Mochi runtime configuration parameters with one exception: the user-level thread stack size was limited to 64 KiB (via the `"abt_thread_stacksize":65536` Bedrock JSON parameter) to reduce memory allocation overhead.

Each system was configured to utilize the HPE Slingshot virtual network interface (VNI) access control mechanism in different ways. The Frontier and Perlmutter systems used a job-level VNI by launching executables with `srun` command line arguments that included `–network=job_vni,single_node_vni` and a Mochi Bedrock JSON configuration that instructs Mercury to autodetect the optional VNI (using the `"auth_key":"0:0"` parameter). Polaris and Aurora used the default system service VNI by passing the `–no-vni` command line argument to `mpiexec` to prevent use of additional job-level or job step-level VNIs. The VNI configuration is not expected to have any impact on performance; we simply opted for the most convenient configuration on each system at the time that our experiments were performed.

4 Methodology

This section summarizes the methods we used to optimize performance on our experimental platforms, adapt roofline models

[1]https://github.com/mochi-hpc/mochi-quintain

[2]Note that the Perlmutter system contains two partitions; we focus on the "GPU" partition in this study. The "CPU" partition has two CPU sockets but only one network link per node.

[3]Note that the empirical measurements on Aurora were performed in March 2025, shortly before a revised core specialization configuration was applied in production. We used an explicit core mapping to constrain our experiments to the 102 cores that would be used in its final default configuration.

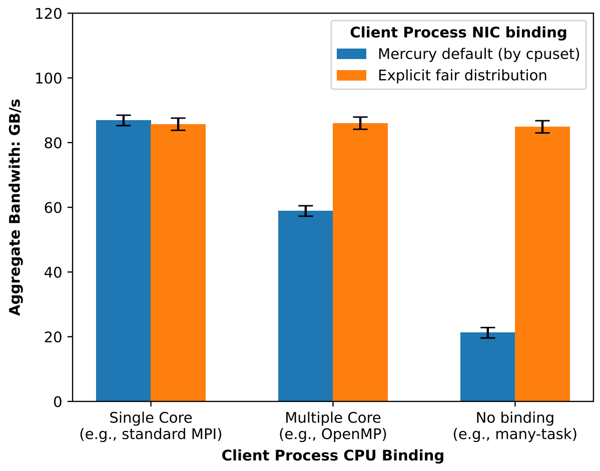


Figure 2: Mean aggregate service throughput measured on Aurora for 8 server processes on one compute node and 16 client processes on a separate compute node across 10 experimental iterations. The client process binding is varied to represent three different application use cases. The blue and orange bars distinguish between two client network card selection strategies, and the error bars show standard deviation across runs.

for use with distributed data services, and collect empirical model parameters.

4.1 Performance optimizations

An effective roofline model must provide a reasonable estimate of ideal performance on a given platform. We therefore began our study by profiling the Mochi software using HPCToolkit [1] to ensure that it made effective use of available compute resources under load. We also added explicit HPCToolkit support to Quintain so that performance profiling could be deferred until synthetic workload intervals of interest. Furthermore, we implemented the optimizations described below in order to maximize performance on the systems described in Table 1.

4.1.1 Network selection with Mochi-plumber. Distributed services or applications can be configured to make optimal use of available network links given a priori knowledge of CPU, NIC, NUMA, and PCI topology. However, manual configuration is labor-intensive, non-portable, and error-prone. We therefore sought to automate this process in Mochi to obtain maximum performance more reliably on arbitrary platforms. Other runtime frameworks such as MPICH [13] have also encountered and solved this problem. Data services, however, present unique challenges that preclude the reuse of existing solutions. In the user-level data service client/server model, processes are inherently asymmetric, may be ephemeral, may be launched at different times, may or may not be multithreaded, and have no concept of a global "rank" that can be leveraged to arbitrate conflicts.

We conducted an experiment to quantify this issue on Aurora. We launched 8 Quintain servers on one compute node and 16 Quintain client processes on a second node. Each server process was bound to a specific CPU core and a distinct network interface card, while we varied the client process CPU binding and network selection policies to observe the impact. Each client process issued a 5-second burst of RPCs, with each RPC requesting a 4 MiB RDMA transfer. Figure 2 shows the resulting aggregate throughput for a range of scenarios. The blue bars show bandwidth when each process uses the default hwloc-enabled Mercury network selection policy, while the orange bars show bandwidth for processes using an explicit network card mapping that evenly distributes the 16 processes over the 8 network cards in all scenarios.

The default Mercury network card selection algorithm assigns network cards based on the `cpuset` of the process (i.e., the range of cores that a process is allowed to execute on), taking into account hardware locality as reported by `hwloc` [3]. This policy performs well when all processes are pinned to individual cores, as would be the case in a conventional MPI application. The second set of bars represents a scenario in which each process is bound to multiple cores (two in this example), as might be the case in a multithreaded or OpenMP application. Aggregate throughput declines when using the default Mercury policy in this scenario because four network cards are overloaded on the client node while four others are left unused. The final set of bars represents a scenario with core binding completely disabled, as may be the case for a throughput-oriented task-based application. In this final scenario all processes are technically in the same cpuset (0..N) and thus select the same network card when using the default Mercury policy, leading to dramatic underutilization of available bandwidth. In contrast, the measurements in which processes are explicitly and evenly distributed over available network cards (shown in orange) maintain maximum throughput for all three scenarios. These measurements suggest that an alternative automatic algorithm may be able to provide a better general solution for different classes of applications and different types of service processes.

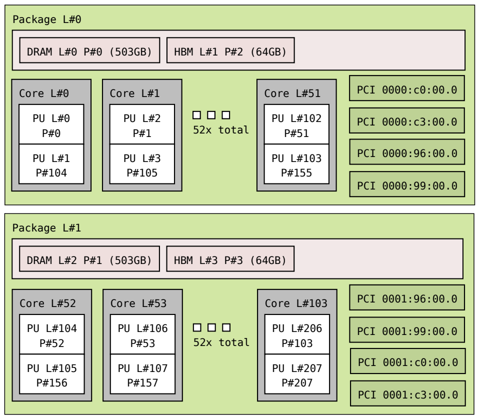


Figure 3: Simplified view of Aurora compute node topology as reported by the `hwloc` command line tool, `lstopo`. GPUs and full PCI topology are omitted for clarity. The 4 PCI devices local to each package each correspond to Slingshot network cards.

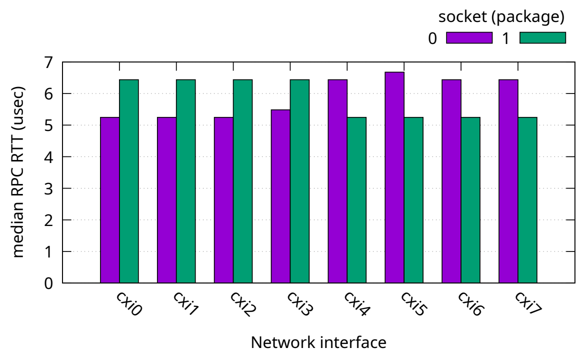


Figure 4: Median point-to-point round-trip noop RPC latency between Mochi processes on separate Aurora nodes as we explicitly vary the network interface (cxi*) and whether the processes are bound to a core on the first or second socket.

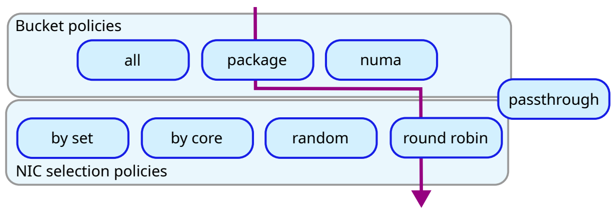


Figure 5: Conceptual diagram of policies in Mochi Plumber. The bucket policy divides network cards into groups, while the NIC selection policy determines how cards are selected from within those groups. The "passthrough" policy delegates network card selection to underlying layers of the network stack.

Network card dispersion is not the only factor to consider when assigning network cards to processes. In some node topologies, there is also a latency benefit to selecting network cards with better locality. For example, consider the node topology of Aurora as shown in the simplified `lstopo` diagram in Figure 3. In this case each package (i.e., CPU socket) has greater locality to four of the eight available network cards.

We can measure the impact of this topology on network performance as illustrated in Figure 4. It shows the median point-to-point round-trip RPC latency between processes on separate nodes for each combination of network card and CPU socket. Note that we used the simpler `margo-p2p-latency` benchmark[4] to collect these measurements rather than Quintain. Our experiments show that traffic over a non-local network card incurs a median round-trip latency penalty of over 1.2 microseconds.

Based on these findings, we designed a new Mochi component – `mochi-plumber`[5] – as a more general solution to network card selection that can accommodate a broader range of data service scenarios. It implements a two-tier selection policy as illustrated in Figure 5. The first tier is responsible for dividing available network cards into *buckets* based on locality. It programmatically navigates the node topology using `hwloc` to find PCI identifiers reported by `libfabric`. We implemented the following bucket policies: "all", in which all network cards are grouped together, and "package" and "numa", in which network cards are grouped by CPU package (i.e., socket) or NUMA domain locality, respectively. The second tier is responsible for selecting a specific network card within the appropriate bucket for a given process. We implemented multiple NIC selection policies as well. The "by set" and "by core" policies use static mappings based on either the cpuset or the core on which the process is currently executing. The "random" policy selects a random network card. The "round robin" implements a fair distribution by coordinating across arbitrary processes using control files in a temporary directory. This last policy incurs higher latency to perform selection, but this algorithm is performed only once at process startup.

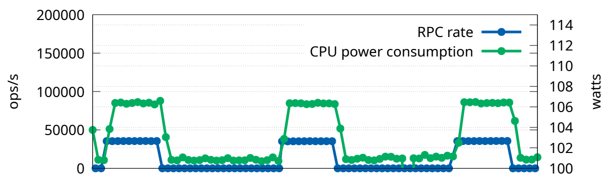
Based on our experimental findings across a variety of platforms, we configured `mochi-plumber` to use a default bucket selection policy of "package" and a default NIC selection policy of "round robin." This policy is used for all remaining experiments in this study. Note that three of the systems in this study have only a single CPU socket, in which case the "package" policy effectively becomes an "all" policy, in which all network cards belong to the same bucket. This works well for many common use cases, including those shown in Figure 2, but notably it works by optimizing network card selection at process launch time. It is therefore not guaranteed to provide a balanced load at runtime if processes migrate or if some processes are more active than others.

4.1.2 Adaptive polling with Margo spindown. Another factor that data services on heavyweight node architectures should consider is how much CPU utilization and power they consume. These are especially critical when data services are provisioned alongside application or workflow tasks; ideally, data service processes should relinquish CPU resources when idle in order to allow more capacity for other tasks.

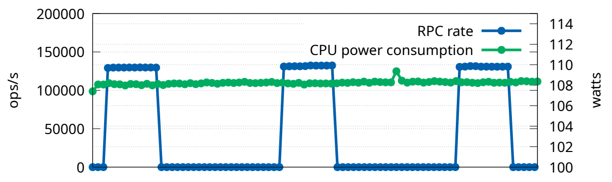
The Mochi Margo component [7], which underpins all Mochi services, is central to meeting this objective. It manages interactions between Mercury RPC operations and Argobots user-level threads to form a coherent, simplified framework for implementing highly concurrent data services. One of its key features is a transparent network progress loop that abstracts this functionality away from the data services themselves. The progress loop may elect to block while waiting for network operations to complete (a semantic that gets translated into `WAIT_FD` operations at the `libfabric` level) or actively poll for network events. The former minimizes CPU load and power consumption, while the latter yields higher performance [6]. By default, Margo employs an adaptive strategy in which quiescent services block while waiting for new requests and actively poll while operations are in progress.

This adaptive polling strategy works well for concurrent workloads but not for sequential workloads in which a single client waits for individual requests to complete before issuing the next request. In this case the server experiences a brief (roughly one network RTT) period of inactivity between each operation, causing the server to rapidly oscillate between blocking and polling mode

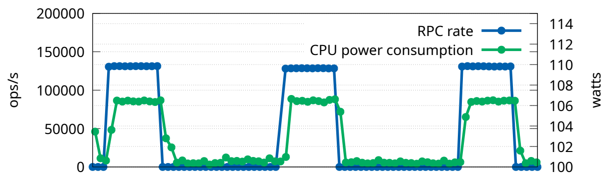
[4]https://github.com/mochi-hpc-experiments/mochi-tests/tree/main/perf-regression
[5]https://github.com/mochi-hpc/mochi-plumber



(a) Adaptive polling



(b) Continual polling



(c) Adaptive polling with spindown optimization

Figure 6: Sampling of RPC rate (left y-axis and blue line) and server CPU power consumption (right y-axis and green line) over time for a sequential, single-client RPC workload on Improv. Each point on the line represents the rate over a 1-second interval.

and incurring additional interrupt and context switching overhead for every operation.

This is a significant performance consideration when attempting to isolate the constituent elements of data server performance. However, the effect is obfuscated on our experimental platforms by a software bug that prevents processes from truly blocking for new requests (the `WAIT_FD` mechanism always reports that events need to be processed).[6] We anticipate this being a key issue in the near future, however, so we elected to explore the phenomenon on a platform that already honors the `WAIT_FD` semantic. For this purpose we performed a set of experiments on the Improv system at Argonne's Laboratory Computing Resource Center to serve as a proxy for upcoming Slingshot network characteristics. Each Improv node is equipped with two 64-core CPUs and one InfiniBand network adapter.

Figure 6 illustrates a workload in which a single server (executing on one compute node) services a sequential noop RPC workload from a single client (executing on a different compute node). The client issues operations in three 10-second bursts with 20-second gaps between each burst in order to observe both idle and active behavior. Full operation tracing was enabled in Quintain so that precise operation rates can be calculated for each 1-second interval. We also concurrently sampled the Linux kernel's Running Average Power Limit (RAPL) metrics via `/sys/class/powercap/intel-rapl/$i/energy_uj` on 1-second intervals in order to correlate CPU power consumption with server activity. The results of this experiment are shown in Figure 6, with the left y-axis displaying RPC rate and the right y-axis displaying CPU power consumption. The first plot (Figure 6a) shows the baseline Margo adaptive polling strategy in which the network progress loop blocks waiting for new operations when idle. The second plot (Figure 6b) shows an alternative configuration in which Mochi actively polls for network events at all times. The second scenario yields a 3.5x improvement in perceived RPC rate but does not allow the CPU to idle when there is no activity to process.

To bridge this gap, we introduced a new optimization in Margo called "spindown." It uses a configurable parameter for the amount of time that the network progress loop should continue to actively poll after completing all known operations. Figure 6c illustrates the impact of activating this optimization with a default value of 10 milliseconds. The spindown optimization enables the server to sustain peak performance during bursts of sequential workload activity while still relinquishing the CPU gracefully during idle periods. Interestingly, it also consumes no more power than the original default policy during bursts of activity. The spindown optimization is enabled by default in Margo 0.19.1 and is used for all subsequent experiments in this paper.

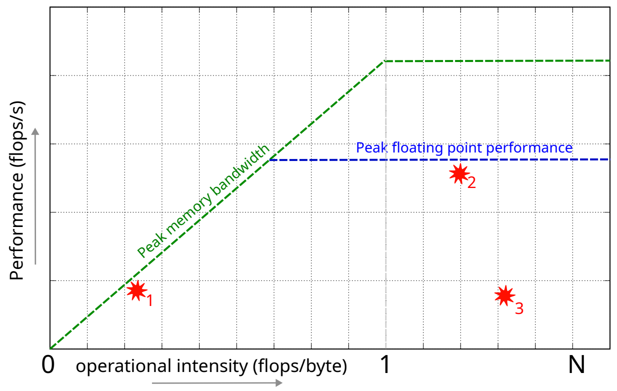
We note that a more comprehensive power monitoring framework such as GEOPM [11] would provide a more complete picture of node power consumption (including impact on the network card and PCI bus); we intend to investigate this more fully once blocking network capability is available on the Aurora platform. We also note that these experiments are in some ways using CPU power consumption as a proxy metric for CPU utilization. CPU utilization could also be measured directly and plotted for comparison in future work. This would help illustrate the potential performance (jitter) impact for colocated processes in addition to the power consumption impact.

4.2 Adapting roofline models to data services

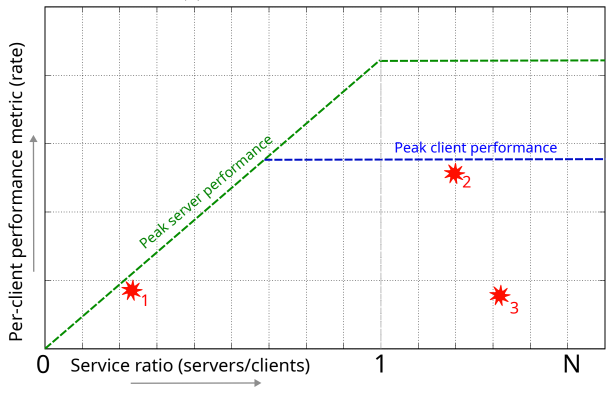
The traditional roofline model as put forward by Williams et al. [24] is illustrated by the conceptual diagram in Figure 7a. It provides a simple, intuitive framework for interpreting application computational performance. Applications are characterized by operational intensity (or arithmetic intensity) as a ratio of flops per byte on the x-axis. Application performance is indicated by flops per second on the y-axis. The upper boundaries on achievable performance are denoted by lines showing peak memory bandwidth and peak floating-point performance for the platform. Plotting application characteristics vs. performance (as in points 1, 2, and 3) indicate whether the application's performance is gated by platform capabilities or some other factor. The roofline model can be augmented in various ways to account for additional performance factors.

We propose an adaptation of the classic model for use with distributed data services in Figure 7b. Rather than operational intensity, the independent variable is the service ratio, or number of servers divided by number of clients. It can therefore be applied to services and workloads at arbitrary scales. As noted previously in Figure 1, no one universal performance metric is most important for

[6]This is expected to be resolved in the next major software release; see https://github.com/ofiwg/libfabric/pull/10681.



(a) Classic roofline model



(b) Proposed data service roofline model

Figure 7: Conceptual illustrations of roofline models. Both the traditional roofline model and the proposed data service roofline model employ straightforward parameters to express performance boundaries for a range of scenarios. In these figures hypothetical points 1 and 2 represent scenarios that are limited by platform capabilities, while point 3 represents a scenario that is limited by some other inefficiency.

all data service use cases. We therefore intend to explore multiple I/O performance metrics on the y-axis, but in all cases focusing on sustained *rates* of steady-state performance.

Note that the per server performance has an inflection point at a service ratio of $X = 1$. At this point each server is saturated and will be oversubscribed for higher service ratios; there is no additional per server performance to be extracted. We also note that we expect to find intersection points between the peak server performance and the peak client performance (where the green and blue lines intersect in Figure 7b. This intersection point could occur at different points depending on the nature of the platform and the metric being measured. Regardless, its position and shape give an indication of the optimal service ratio to maximize performance. Service ratios to the left of this point indicate that the data service is underutilized. This is a unique aspect of the distributed data service roofline model as compared with the traditional roofline model: when used to assess user-level services, the user has the option

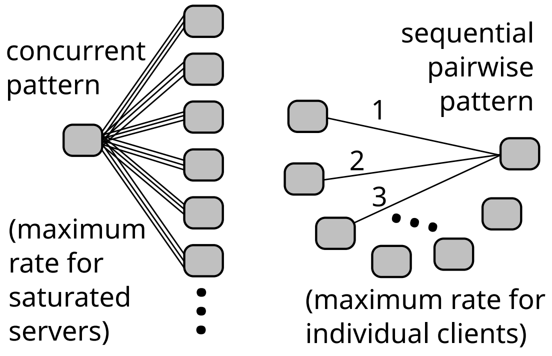

concurrent pattern
(maximum rate for saturated servers)
sequential pairwise pattern
1
2
3
(maximum rate for individual clients)

Figure 8: Diagram of client/server patterns used to gather roofline parameters. Peak server performance is measured under saturation, with a single server node and N client nodes, each executing multiple processes per node. Peak client performance is measured with point to point benchmarks, cycling through all $(N(N-1)/2)$ possible pairings to mitigate the impact of network topology.

to *alter* the server/client ratio to improve efficiency by moving to different points on the x-axis. This is also the key reason why we elected to use a server/client ratio on the x-axis rather than an I/O operational intensity metric as explored in previous work [28, 29].

The parameters of this distributed data service roofline model are straightforward to collect with small-scale empirical measurements, as we will see in the next section. The model relies on multiple simplifying assumptions, however. We assume that node performance is homogeneous. Furthermore, we assume that the fabric topology is effectively infinite in that it can support full connectivity between any ratio of servers and clients at scale. Neither of these assumptions holds in a real-world system, but the objective of the roofline model is not to achieve perfect accuracy but to provide intuitive insight into distributed data service performance. We will validate the model against a range of configurations in Section 5 and discuss the trade-offs that we observe between accuracy and simplicity.

4.3 Collecting model parameters

As noted in Section 1, we could base the roofline parameters on hardware specifications, but this approach is unlikely to yield a practical reference point for data services that rely on complex combinations of hardware components. We therefore focus our methodology on how to collect the *peak client performance* and *peak server performance* empirically for a given metric. Furthermore, ideally the metric measurement should be encapsulated in a standard benchmark or benchmark suite that can be executed on any arbitrary platform to quickly construct a roofline model for data services on that platform using a modest number of compute nodes. In this study we focus exclusively on network-oriented performance metrics: RPC rate and network throughput. For simplicity, we assume that all operations are issued synchronously, which is the most common data service usage pattern. For each metric we must collect two parameters: the peak server (per process) rate and the peak client (per process) rate.

Our expectation is that each server process is responsible for servicing a concurrent workload from many clients, so we focus on measuring a server's maximum performance when saturated. We can measure this as illustrated on the left side of Figure 8 by designating one node as a server and having multiple client nodes measure the aggregate rate achieved for a concurrent workload. Also note that we assume each node runs multiple clients and multiple servers, as indicated by multiple lines connecting the nodes. We collect five samples of this workload pattern to account for variability. We also divide the aggregate rate by the number of server processes executing on the server node to produce a per-server-process rate rather than a per-server-node rate.

Client process peak rate has different characteristics; for synchronous sequential operations a given client process can go no faster than the rate it achieves when communicating with a single quiescent server process. We therefore measure the peak client rate using point-to-point measurements. Such measurements are particularly susceptible to network topology, however, as there is no amortization of costs across a collection of processes. We therefore employ the pattern illustrated on the right side of Figure 8 in which we sample the performance between every $N(N-1)/2$ possible unique pair of nodes to collect a range of possible peak client rates.

We employ this methodology to collect parameters using 17 nodes on each of our experimental platforms. This approach allows us to measure server saturation rate for a 16/1 server/client ratio and allows us to capture 136 distinct point-to-point client rate measurements.

The remaining benchmark parameter to consider is how many server processes should be executed per server compute node during the saturation test. It is conceptually plausible that the peak server process rate could decline if too many server processes per node degrade performance because of CPU, memory, or network contention. We therefore conducted a set of experiments where we designated a single compute node as a server node and 16 compute nodes as client nodes, with each client node executing a number of processes equivalent to the number of usable cores per node. For example, on Aurora this approach yields a total of $16 \times 102 = 1632$ client processes issuing work. We then varied the number of server daemons executing on the server node to determine the optimal number of server daemons to use. The server daemons in Quintain are single-threaded in this scenario, and each daemon is bound to a physical CPU core. Early trial runs (not shown) indicated that binding server daemons to logical SMT threads rather than physical cores did not improve aggregate performance on any of our test platforms.

The results of this experiment for the RPC rate metric (as measured by configuring Quintain to issue sequential noop RPCs from each client simultaneously) are shown in Figure 9 for all four experimental platforms. In each case we see that the aggregate RPC rate grows nearly linearly with the number of server daemons executed on the node. This indicates that servicing small requests in a data service is almost entirely CPU-bound. Servicing a request requires decoding an incoming request, routing it to the appropriate service handler, creating a user-level thread, executing the user-level thread, encoding a response, and injecting the response into the network. In fact, the distinction in performance between platforms is negligible beyond the fact that some platforms have more available

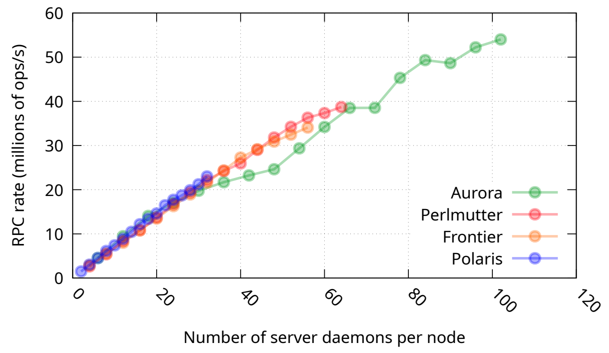


Figure 9: Aggregate RPC rate from a single server node as the number of server processes is varied. For each system we ramp up until the number of server processes is equal to the number of usable physical cores.

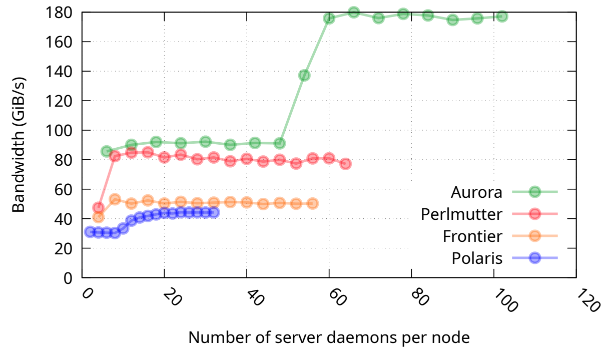


Figure 10: Aggregate network bandwidth from a single server node as the number of server processes is varied. For each system we ramp up until the number of server processes is equal to the number of physical cores.

CPU cores per node than others (Polaris has the fewest at 32, and Aurora has the most at 102). Aurora obtained the highest aggregate rate in this experiment, with a single server node sustaining over 54 million operations per second when running 102 concurrent server processes.

Figure 10 repeats this experiment but measuring network throughput rather than RPC rate. To measure network throughput, we altered the Quintain configuration so that each RPC triggers a 4 MiB RDMA transfer from the server to the client (i.e., mimicking a storage service "read" operation). In previous work we found that the Slingshot network throughput was saturated at 64 KiB transfers [17], so 4 MiB should be more than sufficient to maximize throughput, even accounting for control RPC overhead. In contrast with the previous RPC rate experiment, the aggregate peak rate is not gated by the number of CPU cores but rather by the number of network links available for use by the server processes. This is particularly noticeable in the dramatic jump in bandwidth on Aurora as it crosses the 51 process boundary. The default launcher process binding populates all cores on the first socket before utilizing cores on the second socket. Because the network cards are bucketed by socket according to Mochi Plumber as described in Section 4.1.1, this means that half of the network cards are unused until processes are placed on the second socket. This behavior could be altered by specifying a different processing binding strategy or different Mochi Plumber algorithm, but it helps illustrate the resource bottleneck in this particular experiment. Aurora also achieves the highest peak rate in this experiment, achieving over 177 GiB/s from a single server node when the server node is executing 102 server processes.

Fortunately, although bandwidth does not *improve* when using additional CPU cores beyond a certain point, it also does not *decline*. Thus, if we wish to optimize both RPC rate and network bandwidth, our best strategy is to fully populate compute nodes. Our roofline parameter benchmark suite therefore fully populates n server processes on the target server node (where n is the maximum number of usable physical cores on the node).

In future work we will revisit the parameter benchmark suite to include access to storage devices, at which point we will need to reevaluate our assumptions for how to maximize node performance. We also note that some deployments use cases will intentionally not fully populate a node (for example, if server daemons are meant to execute in tandem with application processes), which will also require further refinement to the model.

5 Survey and evaluation

To evaluate our roofline methodology, we collected empirical roofline parameters for each platform, constructed the corresponding roofline models, and used validation samples to assess the model across a range of deployment footprints.

The empirical roofline parameters were collected by running a benchmark suite as described in Section 4.3. This was executed as a single job script using 17 compute nodes with a maximum wall time of 4 hours. The actual execution time of the benchmark suite was dominated by two factors unrelated to the length of the measurement intervals (5-second workload bursts). The first factor was the overhead of repeatedly launching and stopping sets of processes, a step that could likely be optimized by reducing library load time using tools such as Spindle [12] or Copper [18]. The second factor was conservative estimates of how long the script should delay waiting for servers to launch before initiating client measurements. This could be optimized by adding a Mochi mechanism for gracefully waiting for all servers in a given group to be ready.

Table 2 shows a summary of the roofline parameters collected on each system. The table values are shown as a range from minimum to maximum. The parameter with the widest range by far is the client RPC rate. This is due to the fact that the single-client sequential operation rate is almost entirely bound by the round-trip latency, which in turn is extremely sensitive to the network topology. Polaris exhibited the largest variation in RPC rates, which we will discuss further in Section 5.1.

We collected validation samples by running a single 20-node job in which we varied the total number of server nodes from 1 to 4 and the total number of client nodes from 1 to 16, yielding 64 performance samples spanning a range of the roofline space at different aggregate scales. These samples were collected using the same Mochi Quintain configurations as were used to collect input parameters to the roofline model. Figure 11 shows the maximum aggregate RPC rate achieved by each system in its largest validation configuration with 4 server nodes and 16 client nodes. In the Aurora case, for example, each node contains 102 usable cores; thus, the total process count was 408 server processes and 1,632 client processes. The relative performance of the systems mostly follows expected trends based on our earlier finding that this workload is mostly CPU-bound (Aurora has the most available cores per node at 102, while Polaris has the fewest at 32). The Frontier and Perlmutter results invert expectations; however, Perlmutter should intuitively achieve a higher aggregate rate by virtue of having 64 rather than 56 available cores. The fact that it does not may be an indication of the benefits of core specialization at scale on Frontier. Additional experimentation would be needed to confirm.

Figure 12 shows the maximum aggregate bandwidth achieved by each system in its largest validation configuration with 4 server nodes and 16 client nodes, as in the previous RPC rate figure. Aurora, Polaris, and Perlmutter performance follows expected trends; overall performance is gated by the number of network cards per node in each system (8, 2, and 4, respectively, in this case), and the aggregate throughput is accordingly proportional. Frontier does not follow this trend (it should notionally achieve the same aggregate bandwidth as Perlmutter). This case requires further investigation,

Table 2: Empirical roofline parameter ranges (per process)

	Client RPC rate (K ops/s)	Server RPC rate (K ops/s)	Client net BW (GiB/s)	Server net BW (GiB/s)
Aurora	148-173	524-530	20.2-20.5	1.7-1.8
Polaris	84-180	801-804	19.0-21.1	1.4-1.4
Frontier	127-146	601-603	14.1-20.7	0.9-0.9
Perlmutter	96-131	604-612	19.0-20.3	1.2-1.2

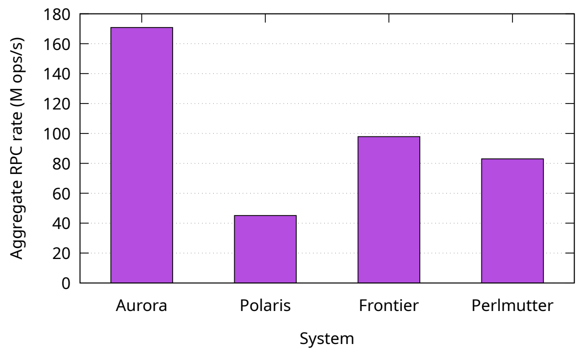


Figure 11: Aggregate RPC rate measured with 4 server nodes and 16 client nodes. Server nodes and client nodes were both configured to run one process per physical core according to the number of usable cores shown in Table 1

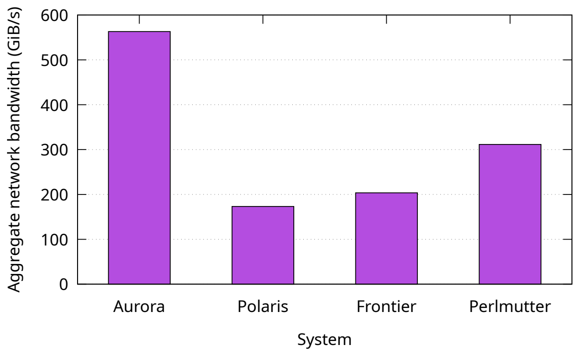


Figure 12: Aggregate network bandwidth measured with 4 server nodes and 16 client nodes. Server nodes and client nodes were both configured to run one process per physical core according to the number of usable cores shown in Table 1

but it may be the result of a node topology that favors connectivity to GPU devices over connectivity to CPUs [2].

We note that although these four systems exhibit significant differences in aggregate performance, they should be easily comparable using our proposed distributed data service roofline model because it factors out scale and instead focuses on server/client ratio and per process performance. As expected because of its higher number of CPU cores and network links, Aurora achieved the highest aggregate performance measurement, with a peak RPC rate of over 170 million ops/s and a peak network throughput of over 563 GiB/s. Those rates were achieved with 4 server nodes (408 server processes) and 16 client nodes (1,632 client processes).

5.1 RPC rate

Distributed data service rooflines, using RPC rate as the performance metric, for all four experimental systems are shown in Figure 13. Both the client process rate ceilings (horizontal lines) and server process rate ceilings (curved slope lines) are given by two lines to indicate the minimum and maximum estimated ceilings according to the range of samples collected by the roofline parameter benchmark suite. In all cases the range for the server rate is narrow enough that the lines overlap. The server rate limit is visually represented by a curved line because the x-axis is logarithmic but the y-axis is not. We elected to utilize a linear scale on the x-axis to show greater detail. The points plotted on the roofline graphs indicate per process rates achieved by the validation samples collected across a range of server/client permutations measured within a single 20-node job. They are color coded to indicate whether the point represents a sample with 1, 2, 3, or 4 server nodes. Because the model is scale agnostic, the points often overlap, and thus we use semi-transparent points to give a visual indication of overlapping values.

All four systems exhibit close agreement when the configurations are clearly bound by server rates (in the 1/16 to 1/8 server/client ratio region). The roofline models indicate that maximum efficiency for aggregate RPC rate will be achieved with approximately a 1/4 server/client ratio on each platform. This is confirmed by the validation samples, which mostly reach steady state around the same point. We note, however, that, with the exception of Polaris, each system approaches the minimum range of the projected client rate ceiling rather than the maximum range. This result suggests that the aggregate rate may be constrained by the slowest links between the sets of nodes used for validation, but further experiments controlling for topology would be necessary to confirm.

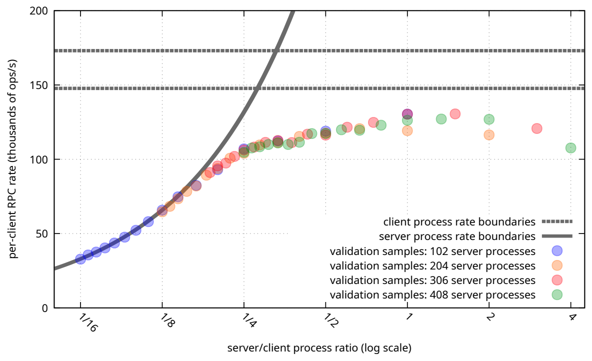
Polaris has the most unusual roofline model of the four systems, because the range of possible per client RPC rate limits is far larger than on any other system; in fact, the RPC rate validation samples exceed the minimum value in the range. Our first intuition was that an outlier sample may have skewed the range. Upon closer inspection we found that there were in fact 11 significantly slower samples that achieved rates below 100,000 operations per second. These 11 samples had a clear commonality in that they all included one specific node as either a client or server in the point-to-point pairing. While we have no post hoc method to determine why this particular node underperformed relative to its peers, it does illustrate that the sampling method we used to calculate the maximum per client performance is particularly sensitive to non-homogeneous performance.

5.2 Network throughput

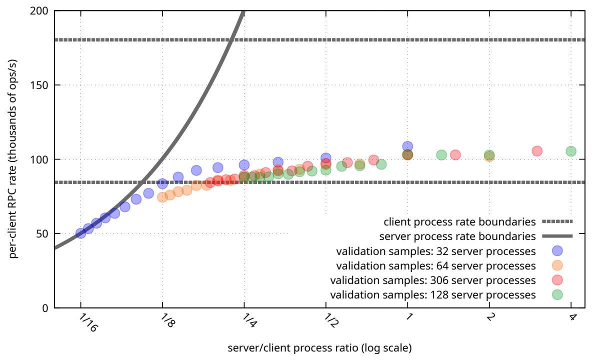
Distributed service rooflines using bandwidth as the performance metric are shown in Figure 14. Other than the performance metric, the notation is identical to that used for the RPC rooflines in Section 5.1. One notable difference is that the client process rate ceilings are not visible in these models. The potential bandwidth of a single client to an idle server approaches a significant fraction of the full capacity of a network link, which is far higher than the fractional bandwidth available to a server process when it is one of N server processes sharing node capacity. Per client maximum bandwidth is therefore not a meaningful factor in aggregate bandwidth modeling.

Polaris and Perlmutter exhibit good agreement between the projected roofline model and the validation samples. On Aurora, there was a wider gap between the maximum rate indicated by the roofline and the achieved rate in the validation samples. This could be a characteristic of the network topology on Aurora but would require further experimentation to isolate. All systems approached maximum bandwidth at the expected ratio of 1:1 servers:clients, which is the point at which the number of network links available to clients is in equilibrium with the number of network links available to servers.

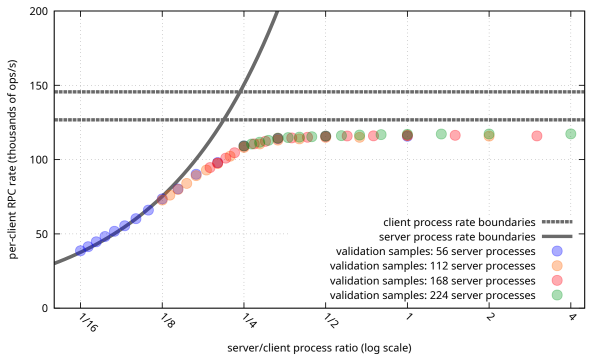
The Frontier roofline is particularly unusual because it is the only roofline in the study in which validation examples consistently exceeded the roofline values. This phenomenon is evident after the 1/1 server/client ratio, when there are more servers available than necessary to meet the client bandwidth demand. We conducted additional roofline parameter benchmarks runs and validation sample runs (yielding 3 total runs of each) and plotted the results in Figure 14e to see whether our first parameter and sample collection was an aberration. Although this established a broader range of maximum bandwidth estimates, we found that the validation examples still exceeded this rate in many cases. This suggests that,



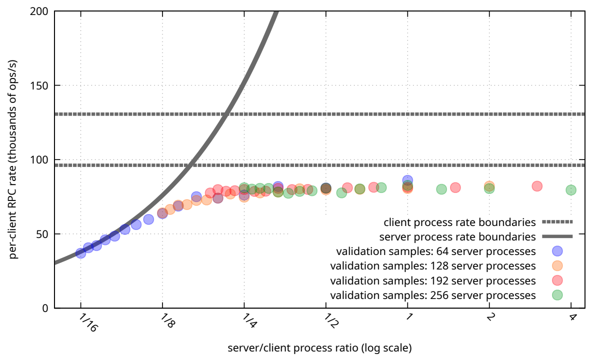
(a) Aurora



(b) Polaris



(c) Frontier



(d) Perlmutter

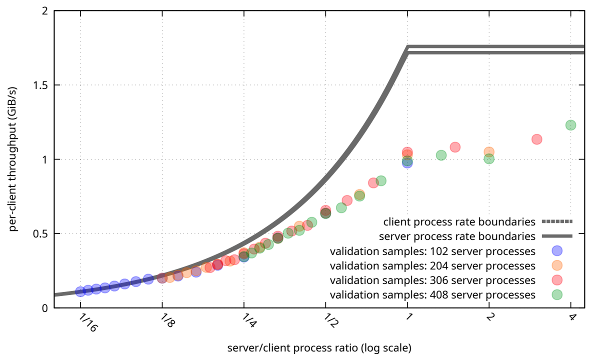
Figure 13: RPC rate rooflines for each system. The boundary lines are based on the empirical parameters from Table 2, while the points represent validation samples measured separately over a range of configurations. Note that the roofline slope is curved because the x-axis is logarithmic but the y-axis is linear to show more detail.

in aggregate, the servers on Frontier can exceed projected performance when individual server processes are *not* oversubscribed. It is not obvious what accounts for this discrepancy, although, as noted in the previous discussion of Figure 12, Frontier exhibits different aggregate bandwidth behavior from the other three systems. This may be due to its unique node topology that favors GPU-network connectivity [2].

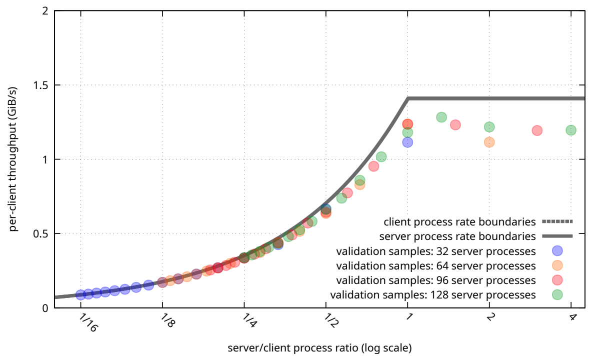
6 Discussion

We found that the proposed roofline modeling technique is a promising method for rapid, intuitive understanding of distributed data service performance, although thus far we have evaluated it only for metrics that focus on network performance. We also made a number of key observations over the course of our study, as summarized below.

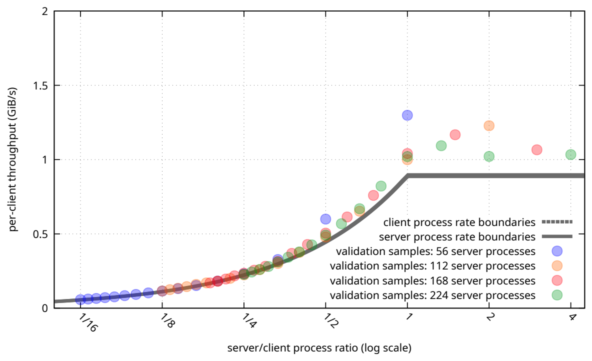
- The proposed roofline model was more sensitive to network topology factors than we anticipated.
  - The models would benefit from a better method to account for network topology than sampling performance on a small collection of nodes.
  - Topology will play less of a role when we incorporate storage device metrics in future work, since data services on most platforms will leverage nonshared local devices that are not impacted by network topology.
- The models were most accurate when servers were oversubscribed, which fortunately is often the common case in real-world computer science applications.
- The roofline models were also effective at predicting the optimal saturation point for a given configuration.
- Roofline parameters can be collected empirically for a given system with modest resource consumption, and we believe that the time to produce these parameters could be reduced further by applying optimizations to how processes are launched on a platform and how clients and servers are coordinated in Mochi.
- Aggregate RPC rate is gated almost entirely by CPU capacity, while aggregate network bandwidth is gated almost entirely by available network links and fabric topology.
- Automated methods for distributing traffic over multiple network cards are crucial to achieving maximum performance on the current generation of HPC platforms.
- Adaptive polling methods that account for bursty sequential patterns while still preserving favorable idle behavior are also crucial for some workloads.



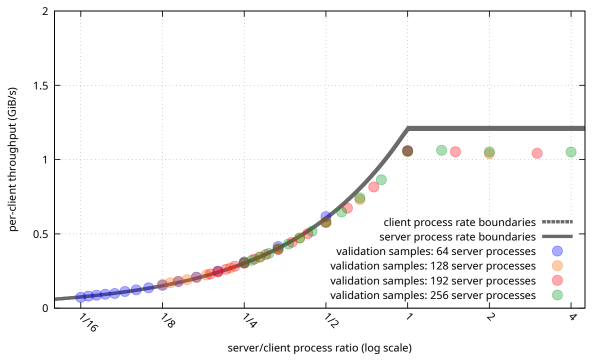
(a) Aurora



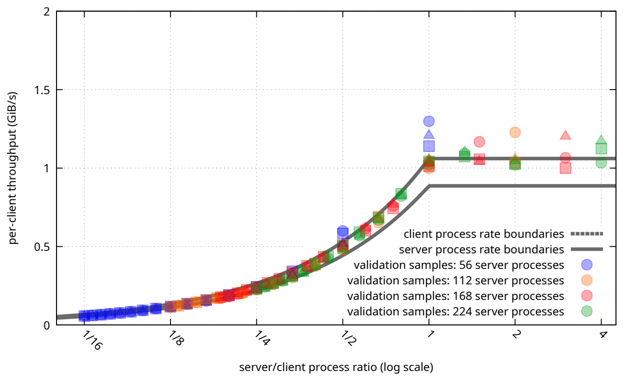
(b) Polaris



(c) Frontier



(d) Perlmutter



(e) Frontier bandwidth roofline, revised to include additional parameter measurements and validation samples. The server process rate boundaries have a wider range after including samples from three distinct parameter benchmark jobs. Validation samples from three distinct validation jobs are denoted by different shapes to highlight the observation that each validation job produced samples that exceeded the roofline boundaries.

Figure 14: Network throughput rooflines for each system. The boundary lines are based on the empirical parameters from Table 2, while the points represent validation samples measured separately over a range of configurations. Note that the roofline slope is curved because the x-axis is logarithmic but the y-axis is linear to show more detail.

- The compute nodes and network fabric on all four evaluation systems are remarkably efficient. Aurora, for example, exhibited a peak per node aggregate bandwidth of over 177 GiB/s, or 89% of the combined line rate of the eight 200-gigabit/second network cards on the node.

7 Future work

Based on the encouraging results of this study, we would like to expand the scope of our roofline methodology to incorporate storage device characteristics (in particular, IOPS and storage bandwidth metrics), develop a method for combining multiple metric rooflines to suit target use cases, and apply the roofline to real-world data service implementations. In the longer term we could incorporate additional factors such as peripherals (e.g., for data services that rely on GPU capabilities) or alternative interfaces (e.g., quantifying the overhead of Python bindings).

Based on our findings to date, we would like to explore new methods to account for network topology. Possibilities include staging machine reservations with particular locality properties to investigate corner cases, employing broader statistical sampling of parameters, or exploring methods for interrogating node allocations at runtime in order to reason about the topology programmatically.

We also invite the broader community to improve upon our synthetic testing framework as a means to explore best practice in data service performance and keep pace with platform evolution. Our objective is to eventually create a portable user-friendly benchmark suite that not only gathers roofline model parameters but also can assist in platform validation and benchmarking.

We will also work with the Mercury community to assess whether the Mochi Plumber capabilities, or a subset of them, would be appropriate for native inclusion in the Mercury RPC library. We will also investigate mechanisms for improving process launch and coordination time, an effort that would benefit not only our parameter capture but also the data service community at large.

Acknowledgments

We gratefully acknowledge John Mellor-Crummey's assistance in using HPCToolkit to profile Mochi services.

This work was 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, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award ASCR-ERCAP-32105. We gratefully acknowledge the computing resources provided on Improv, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701. doi:10.1002/cpe.1553 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553

[2] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 52, 16 pages. doi:10.1145/3581784.3607089

[3] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 180–186. doi:10.1109/PDP.2010.67

[4] Philip Carns, Matthieu Dorier, Rob Latham, Robert B. Ross, Shane Snyder, and Jerome Soumagne. 2023. Mochi: A Case Study in Translational Computer Science for High-Performance Computing Data Management. *Computing in Science & Engineering* 25, 04 (July 2023), 35–41. doi:10.1109/MCSE.2023.3326436

[5] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Trans. Storage* 7, 3, Article 8 (Oct. 2011), 26 pages. doi:10.1145/2027066.2027068

[6] Philip Carns, Kevin Harms, Bradley W. Settlemyer, Brian Atkinson, and Robert B. Ross. 2020. Keeping It Real: Why HPC Data Services Don't Achieve I/O Microbenchmark Performance. In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. 1–6. doi:10.1109/PDSW51947.2020.00006

[7] Philip Carns, John Jenkins, Sangmin Seo, Shane Snyder, Robert B Ross, Charles D Cranor, Scott Atchley, and Torsten Hoefler. 2016. Enabling NVM for Data-Intensive Scientific Services. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*.

[8] Fabio Checconi, Jesmin Jahan Tithi, and Fabrizio Petrini. 2022. Ridgeline: A 2D roofline model for distributed systems. *arXiv preprint arXiv:2209.01368* (2022).

[9] Nan Ding, Brian Austin, Yang Liu, Neil Mehta, Steven Farrell, Johannes P Blaschke, Leonid Oliker, Hai Ah Nam, Nicholas J Wright, and Samuel Williams. 2024. A Workflow Roofline Model for End-to-End Workflow Performance Analysis. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[10] Matthieu Dorier, Philip Carns, Robert Ross, Shane Snyder, Rob Latham, Amal Gueroudji, George Amvrosiadis, Chuck Cranor, and Jerome Soumagne. 2024. Extending the Mochi Methodology to Enable Dynamic HPC Data Services. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 414–422. doi:10.1109/IPDPSW63119.2024.00091

[11] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 394–412.

[12] Wolfgang Frings, Dong H Ahn, Matthew LeGendre, Todd Gamblin, Bronis R de Supinski, and Felix Wolf. 2013. Massively parallel loading. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 389–398.

[13] Yanfei Guo, Ken Raffenetti, Hui Zhou, Pavan Balaji, Min Si, Abdelhalim Amer, Shintaro Iwasaki, Sangmin Seo, Giuseppe Congiu, Robert Latham, Lena Oden, Thomas Gillis, Rohit Zambre, Kaiming Ouyang, Charles Archer, Wesley Bland, Jithin Jose, Sayantan Sur, Hajime Fujita, Dmitry Durnov, Michael Chuvelev, Gengbin Zheng, Alex Brooks, Sagar Thapaliya, Taru Doodi, Maria Garazan, Steve Oyanagi, Marc Snir, and Rajeev Thakur. 0. Preparing MPICH for exascale. *The International Journal of High Performance Computing Applications* 0, 0 (0), 10943420241311608. doi:10.1177/10943420241311608 arXiv:https://doi.org/10.1177/10943420241311608

[14] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2013. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 13, 1 (2013), 21–24.

[15] J Kunkel, John Bent, Jay Lofstead, and George S Markomanolis. 2016. Establishing the io-500 benchmark. *White Paper* (2016).

[16] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. 2009. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. doi:10.1145/1654059.1654100

[17] Rob Latham, Robert B. Ross, Philip Carns, Shane Snyder, Kevin Harms, Kaushik Velusamy, Paul Coffman, and Gordon McPheeters. 2024. Initial Experiences with DAOS Object Storage on Aurora. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1304–1310. doi:10.1109/SCW63240.2024.00171

[18] Noah Lewis, Kaushik Velusamy, Kevin Harms, and Huihuo Zheng. 2024. Copper: Cooperative Caching Layer for Scalable Data Loading in Exascale Supercomputers. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1320–1329. doi:10.1109/SCW63240.2024.00173

[19] Radita Liem, Dmytro Povaliaiev, Jay Lofstead, Julian Kunkel, and Christian Terboven. 2021. User-Centric System Fault Identification Using IO500 Benchmark. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*. 35–40. doi:10.1109/PDSW54622.2021.00011

[20] Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, et al. 2020. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology* (2020).

[21] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrian Castello, Damien Genet, Thomas Herault, et al. 2017. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[22] Hongzhang Shan, Katie Antypas, and John Shalf. 2008. Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark. In *Proceedings of Supercomputing*.

[23] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*.

[24] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[25] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, et al. 2018. An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 14–23.

[26] Charlene Yang, Thorsten Kurth, and Samuel Williams. 2020. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience* 32, 20 (2020), e5547.

[27] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, and Yong Jae Lee. 2024. LLM inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363* (2024).

[28] Zhaobin Zhu, Niklas Bartelheimer, and Sarah Neuwirth. 2023. An Empirical Roofline Model for Extreme-Scale I/O Workload Analysis. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 622–627. doi:10.1109/IPDPSW59300.2023.00106

[29] Zhaobin Zhu and Sarah Neuwirth. 2023. Characterization of Large-scale HPC Workloads with non-naïve I/O Roofline Modeling and Scoring. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 737–744. doi:10.1109/ICPADS60453.2023.00112