Support for Provisioning and Configuration Decisions for Data Intensive Workflows

Lauro Beltrão Costa, Samer Al-Kiswany, Matei Ripeanu, and Hao Yang

Abstract—System provisioning, resource allocation, and configuration decisions for I/O-intensive workflow applications are complex even for expert users. Users face choices at multiple levels: allocating resources to individual sub-systems (e.g., the application layer, the storage layer) as well as configuring each of these optimally (e.g., replication level, chunk size, caching policies in case of storage) all having a large impact on the overall application performance. This paper presents a solution to address the problem of supporting these provisioning, allocation and configuration decisions for workflow applications. To enable selecting a good choice in a reasonable time, we propose an approach that accelerates the exploration of the configuration space based on a low-cost performance predictor that estimates total execution time of a workflow application in a given setup. We evaluate the predictor in a number of different scenarios including the Montage application: a workflow composed of over 7,500 tasks structured in 10 different stages with varying characteristics. Our evaluation shows that: (i) the predictor is effective in identifying the desired system configuration, (ii) it can scale to model a complex workflow application run on a 100-node cluster, while (iii) using orders of magnitude less resources than running the actual application. Additionally, we extend the predictor to estimate the energy usage of the system, and we present our experience with incorporating it in the development process of a distributed storage system.

Index Terms—Storage systems, workflow applications, data-intensive, performance prediction, storage provisioning

1 Introduction

Assembling workflow applications by putting together standalone binaries has become a popular approach to support large-scale science [1], [2], [3]. The processes spawned from these binaries communicate via temporary files stored on a shared storage system. In this setup, the workflow runtime engines are basically schedulers that build and manage a task-dependency graph based on the tasks' input/output files (e.g., SWIFT [4]).

To avoid accessing the platform's backend storage system (e.g., NFS, GPFS or Amazon S3), recent proposals [1], [5] advocate using some of the nodes allocated to the application to deploy an *intermediate storage system*. That is, aggregating (some of) the resources of an application allocation to provide a shared temporary in-memory storage system dedicated to (and co-deployed with) the application, an approach also known as burst buffer [6] or staging area [7].

While other approaches exist to run workflow applications (e.g., FlexPath [8]), this approach has gained popularity since it tends to offer a number of advantages [1], [6], [7], [9], [10]: *higher performance*—as applications benefit from a wider I/O channel by striping data across several nodes; *higher efficiency* as it improves resource utilization; and *incremental scalability* as it is possible to increase capacity in small increments. This scenario also opens the opportunity for optimizing the intermediate storage system for the target workflow application: a storage system used by a single workflow, and co-deployed on the application allocation, can be configured specifically for the I/O patterns generated by the workflow. For example, the optimizations include choosing specific chunk-size to optimize data-transfers, configuring striping and replication to eliminate hot spots, using a data placement policy to maximize data access locality, or appropriately provisioning the intermediate storage [9].

These benefits, however, come at a price: configuring the intermediate storage system becomes increasingly complex for multiple reasons. First, the optimization techniques commonly used in distributed environments expose trade-offs that rarely exist in centralized solutions [11], [12]. Second, each application may obtain peak performance at a different configuration point [5], [11], [12], [13]. Third, typical provisioning choices involve deciding at least the allocation size and type (e.g., number of nodes in batch-computing environments, and the node type in cloud environments). Finally, depending on the context, there are multiple metrics of interest to optimize [9], [10], [12]—e.g., time-to-solution, network usage, energy, or the cost of resources. Further complexity emerges from the fact that often users face decisions that entail trade-offs between cost and turn-around time.

The problem. In this scenario, the role of the application user is non-trivial: if a user wants to extract maximum performance, in addition to being in charge with running the workflow application, the user has to configure the deployment and the intermediate-storage system to achieve high performance (e.g., application turnaround time, storage footprint, network usage, or financial cost).[1] This involves

• *The authors are with the ECE Department—The University of British Columbia. E-mail: {lauroc, samera, matei, haoy}@ece.ubc.ca.*

Manuscript received 6 Oct. 2014; revised 14 July 2015; accepted 23 July 2015. Date of publication 0 . 0000; date of current version 0 . 0000. Recommended for acceptance by A.R. Butt. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2015.2497693

1. Staging (i.e., moving initial/final data from/to permanent storage) may impact the performance of the application. However, since the user has less control over the backend storage system, we target the intermediate storage layer where the user has control/flexibility over the configuration.

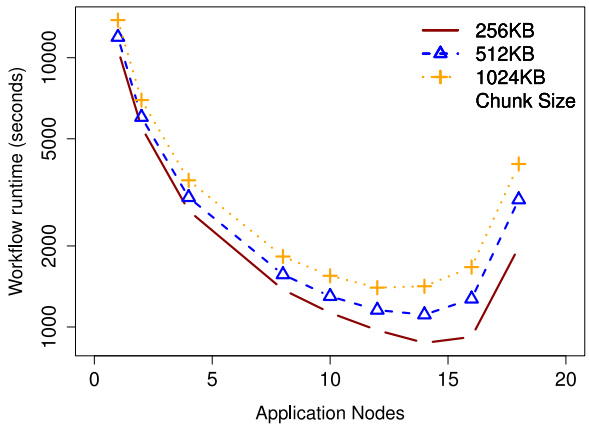


Fig. 1. Different configurations deliver widely different application turnaround time (log-scale on Y-axis) and the choice of the optimal configuration is not intuitive. BLAST workflow [14] execution time on top of an intermediate storage system with different configurations: different ways of partitioning a fixed set of 20 nodes allocated to the application among storage and applications nodes, and different chunk sizes. As the number of nodes allocated to application increases, the overall response time decreases until an optimal point. Then the increasing I/O pressure on the intermediate storage leads to performance loss. The chunk size also has a high impact: up to two-fold difference between the default (1,024 KB) and its optimal (256 KB).

allocating resources and configuring the storage system. Specifically, the decision space revolves around: *provisioning the allocation*—total number of nodes, deciding on node type (s) for cloud environments; *allocation partitioning*—splitting or not these nodes between the application and the intermediate storage system; and *storage system configuration parameters*—choosing the values for several configuration parameters, e.g., chunk size, replication level, cache/prefetching and data placement policies for the intermediate storage system. Consequently, provisioning the system entails searching a complex multi-dimensional configuration space to determine the user's ideal cost/performance balance point. Fig. 1 shows an example of this scenario for BLAST application [14].

In this complex space, generally the user's goal is to optimize a multi-objective problem, in at least two dimensions: maximize performance (e.g., reduce application execution time) while minimizing cost (e.g., reduce the total CPU hours, energy, or dollar amount spent). More concretely, the user is often interested in answering specific questions: *What is the configuration that achieves the lowest total cost? How should I partition the allocation among application and storage nodes to achieve the highest performance? What is the most cost efficient allocation* (i.e., lowest cost per unit of performance)*?*

Manually fine-tuning the storage system configuration parameters and allocation decisions is hard and time-consuming due to the time to consider the potentially large configuration space, and the non-linear interaction among the possible decisions.

To this end, we designed a prediction mechanism able to estimate the application performance, given certain resources and a generic storage system configuration, at low-cost. Specifically, this paper presents our experience on *designing and harnessing a performance prediction mechanism for an object-based storage system in the context of workflow applications using an intermediate storage system*. Given a storage system configuration, an application I/O profile, and a characterization of the deployment platform based on a simple system identification process (e.g., storage nodes service time, network characteristics), the mechanism predicts the application turnaround time, generated network traffic and energy envelope.

This approach has multiple uses. First, it supports *autotuning*: a software tool that relies on the proposed mechanism can enable efficiently configuring the storage system [10], [11], [12], through exploring the configuration space without actually running the application. Second, it can be used to uncover performance anomalies due to software or hardware defects. Finally, it supports exploring additional "what...if..." scenarios; for example, to inform hardware platform acquisition, to estimate the impact of a storage optimization, or estimate the impact of increased service time variability.

The contributions of this paper lie over multiple axes. It:

- Synthesizes the key requirements for a prediction mechanism (Section 2.1) and proposes a design that relies on a uniform queue-based model for distributed, object-based storage systems (Section 2.3). More importantly, the proposed solution relies on application-level operations for the system identification procedure used to seed the model, which makes it simple, lightweight, effective, by not requiring storage system or kernel changes to collect monitoring information (Section 2.5). We have also extended the performance prediction mechanism to estimate the energy envelope of a workflow application (Section 4).
- Evaluates the prediction mechanism for two workflow applications and multiple synthetic benchmarks in the context of making configuration choices for different scenarios (Section 3) and success metrics. Applications include Montage: a complex workflow composed of over 7,500 tasks structured in 10 different stages with varying characteristics. Our experiments highlight the application execution time variation depending on the configuration and provisioning choices made, and demonstrate that the predictor is able to guide the search for the desired balance cost/time-to-solution. The evaluation over a large space of configuration points shows that the predictor is *lightweight*—uses up to 2,000× less resources (machines × time) than running the actual applications, and *effective*—it correctly identifies the configuration that achieves the best performance in the experiments.
- Discusses our experience (Section 5) with using the prediction mechanism beyond our original design goal: we use the performance prediction to better understand and debug a distributed storage system that our group develops. This allowed for a significant improvement of the system's performance and decrease in the response time variance.

Finally, this paper discusses the predictor limitations, the challenges, and the lessons learned (Section 7).

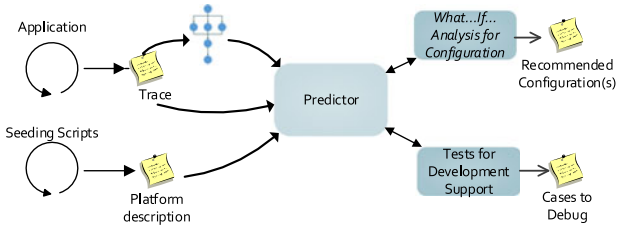


Fig. 2. The predictors input and possible use-cases. To predict an applications performance, the predictor (Sections 2.3 and 2.4) receives the platform (Section 2.5) and workload (Section 2.6) descriptions. The use-cases include *What...If analysis* to support provisioning and configuration decisions - focus of this paper, and supporting the development of a storage system as described in Section 5.

2 The Design of a Performance Estimation Mechanism

Challenges. Making accurate performance predictions for distributed systems is a challenge. Since purely analytical models cannot provide adequate accuracy in most cases, simulation is the most commonly adopted solution. At the one end of the design spectrum, current practice (e.g., NS2 simulator [15]) suggests that while simulating a system at fine granularity (e.g., packet-level network simulation for NS2) can provide high accuracy, the complexity of the model and the seeding process, as well as the number of events generated, make accurately simulating large-scale systems unfeasible, and may reduce the applicability of this approach to small systems and/or short traces. At the other end of the spectrum, coarse grained simulations (e.g., PeerSim [16]) scale well, but at the cost of lower accuracy for complex scenarios.

Two key observations enable us to reduce simulation complexity and increase its scalability: First, as the goal is to support configuration choice, achieving perfect accuracy is less critical as long as the supported configuration decisions are good. Second, we take advantage of workload characteristics generated by workflow applications: relatively large files and specific data access patterns. These observations enable us to reduce the simulation complexity by not simulating in detail some of the control paths that do not significantly impact accuracy. For example, the chunk transfer time is dominated by the time to send the data, thus only coarsely accounting acknowledgments and all metadata messages do not tangibly impact accuracy). Finally, we followed an iterative approach: from a simple model and seeding process, we added more components and interaction details until accuracy obtained was within 10 percent for a set of benchmarks. While our evaluation demonstrates we have obtained adequate accuracy for the scenarios tested, the uniform design we employ (Sections 2.3, 2.4) ensures that additional accuracy can be obtained at the cost of a limited increase in simulation complexity.

The solution space is limited by a number of constraints. First, approaches that rely on human intervention have lower practical use as they assume a user's deep understanding of the modelled system. Second, approaches that rely on a fine-grain monitoring or deep instrumentation of the modelled system or the underlying platform (e.g., kernel) have additional overheads and, more importantly, face adoption barriers. Our approach avoids these challenges by relying only on application-level benchmarks for model seeding. While this approach may encourage adoption, it limits the level of details that can be modelled to only application observable system traits. Another challenge is introduced by the constraints on *model seeding procedure* (i.e., identifying the proper values for model's parameters for a given system deployment, also know as *system identification*).

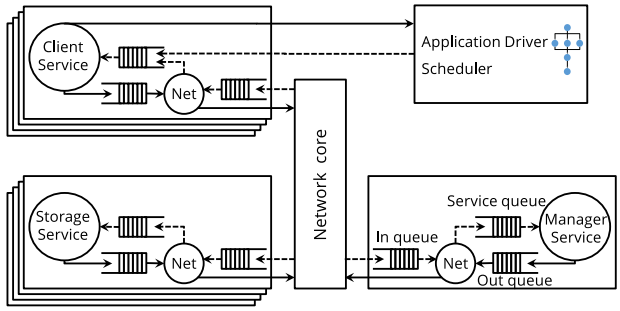


Fig. 3. Queue-based model of a distributed storage system. Each component (manager, client component, and storage component) has a single system service that processes requests from its queue. To model host-level network contention, each host has a network component with an in- and out- queue. Solid lines show the flow going out from a storage system component; dashed lines, the in-flow path. The application driver and scheduler receives the workload description, associates events to the proper clients, and generates the simulation events.

Solution overview. Our solution uses a queue-based storage system model. The model requires three inputs from the user: the storage system configuration, a workload description (Section 2.6), and the performance characteristics of the systems components (i.e., system identification Section 2.5). The predictor instantiates the storage system model with the specific component characteristics and configuration, and simulates the application run as described by the workload description. Fig. 2 shows how these different components interact with the predictor.

This section discusses the requirements for a practical performance prediction mechanism (Section 2.1), and presents the key aspects of the object-based storage system architecture modeled (Section 2.2). Then, it focuses on the proposed solution: it presents the model (Section 2.3, Fig. 3), its implementation (Section 2.4), the system identification process to seed the model (Section 2.5), and an overview of the workload description (Section 2.6). Costa [10] presents additional details on the design and implementation of the predictor.

2.1 Solution Requirements

A practical performance prediction mechanism should meet the following, partially conflicting, requirements that bound the solution space:

- *Accuracy.* The mechanism should provide *adequate accuracy*. Although higher accuracy is always desirable, there are strong practical limitations to achieving perfect accuracy. Additionally, there are decreasing incremental gains for improved accuracy. For example, to support configuration decisions, a predictor only needs to correctly estimate relative performance or trends resulting from changing a configuration parameter. Similarly, if two

configurations offer near performance, perfect accuracy is less important as long as the prediction mechanism places their performance as similar.

- *Scalability and response time.* The predictor should enable quick exploration of the configuration space. The mechanism should offer performance predictions quickly and scale with both (i) the system size, and (ii) the I/O intensity of workflow applications.
- *Usability and generality.* The predictor should not impose a burdensome effort to be used/adopt. Specifically, the bootstrapping/seeding process should be simple and should not require storage system redesign (or a particular initial design) to collect performance measurements. Additionally, the predictor should model a generic object-based distributed storage design and using it should not require in-depth knowledge of storage system protocols and architecture.
- *Ability to explore "what-if" scenarios.* A prediction mechanism should be able to support exploring hypothetical scenarios, such as scenarios that assume new/different hardware configurations.

2.2 Object-Based Storage System Architecture

We focus on the widely-adopted object-based storage system architecture (e.g., GoogleFS, Ceph, UrsaMinor [13], PVFS [17], MosaStore [5]). This architecture includes three main components: a centralized metadata manager, storage nodes, and a client-side system access interface (SAI). The manager maintains the stored files' metadata and system state. To speed up data storage and retrieval, the architecture employs striping: files are split into chunks stored across several storage nodes. The client SAIs implement data access protocols. We model a generic object-based storage system and support exploring the performance impact of a generic set of configuration parameters.

Data placement. The default data placement is round-robin: when a new file is created on a stripe of n nodes the file's chunks are placed in a round-robin fashion across these nodes. Key for workflows, application-driven data placement policies that optimize for specific application access patterns [9], [18] may be specified at file-level granularity. We demonstrate the ability to decide configuration for the following data placement policies: local, co-locate and broadcast (detailed in Section 3).

Replication. Data replication is often used to improve reliability or access performance. While a higher replication level reduces contention on the node storing a popular file, it increases the write time and the storage footprint.

We explore the accuracy of the prediction mechanism assuming that the chunk size, stripe width, replication level, and data placement policy are configurable as suggested in past work [5], [9], [13]. Our approach can be extended to support other configuration parameters.

2.3 System Model

All participating machines are modeled similarly, regardless of their specific role (Fig. 3): each machine hosts a network component, and can host one or more system components, each modeled as a service with its own service time per request type and an infinite FIFO queue.

Each system component and its infinite queue represent a specific functionality: The *manager* component is responsible for storing files' and storage nodes' metadata. The *storage* component is responsible for storing and replicating data chunks. Finally, the *client* component receives the read and write operations from the application, implements the storage system protocol at high-level by sending control or data requests to other services, and communicates again with the application driver once a storage operation finishes. Each of these storage system components is modeled as a service that takes requests from its queue and uses the network service to send requests and responses. The application driver is responsible for scheduling the tasks to the available clients, and issuing requests directly to the client service queue according to the scheduling done.

Using these components we model the four main storage operations: open, close, read, write. As a rule, for each we accurately model the data paths of the storage system at chunk-level granularity, and the control paths at a coarser granularity: we model only one control message to initiate a specific storage function while an implementation may have multiple rounds of control messages (Section 2.4 offers a concrete example for a write operation).

The network component and its in- and out- queues model the network-related activity of a host. Key here is to model network-related contention while avoiding modeling the details of the transport protocol (e.g., dealing with packet loss and retransmission, connection establishment and teardown details). The requests in the out-queue of a network component are broken in smaller pieces that represent network frames and sent to the in-queue of the destination host. Once the network service processes all the frames of a given request in the in-queue, it assembles the request and places it in the queue of the destination service.

The system components can be collocated on the same host (e.g., the client and storage components on the same host). In this situation, requests between collocated services also go through the network, but have a faster service time than remote requests—as a loopback data transfer (Section 2.5).

In addition to the storage system modeling, the prediction mechanism captures part of the execution behavior of the application. The predictor also performs scheduling according to the input describing the application, which includes a task's dependency graph (capturing workflow execution plan) used for scheduling and data placement purposes (see Fig. 2 and Section 2.6 for more details).

2.4 Model Implementation: The Simulator

We have implemented the above model as a discrete-event simulator in Java. The simulator receives as input: (i) a summarized description of the application workload (Section 2.6), (ii) the system configuration (currently, it supports replication level, stripe-width, and data-placement per file; and chunk size system-wide), (iii) the deployment parameters (number of storage nodes and clients, and whether or not they are collocated on the same hosts), and

(iv) a performance characterization of system components: service times for network, client, storage, and manager (Section 2.5).

Once the simulator instantiates the storage system, it starts the application driver that schedules the task to the clients and issues the application workload (Section 2.6). The driver reads the description of the application workload, creates the corresponding events (e.g., at time t after last I/O call read from file x at offset y, z bytes), and places them in the client service queue. File-specific configuration (as proposed by [9], [13]) is described as part of the operations in the workload description.

As in a real system, the manager component maintains the system's metadata (i.e., implements data placement policies, and keeps track of file to chunk mapping and chunk placement). To make the process clearer, consider the example for a file write operation. A client contacts the manager asking for free space, the manager replies specifying a set of storage services with free chunks. Then, the client requests each storage service to store chunks in a round-robin fashion. After processing a request to store a chunk, a storage service replies to the client acknowledging the operation success. After sending all the chunks, the client sends the chunk-map to the manager. Once the manager acknowledges, the client returns success to the application driver. The write operation generates two requests to the manager and one request per chunk to the storage nodes, and it does not capture any further detail about the execution path present in the storage system (e.g., FUSE module).

The manager implements a number of data placement policies. The default policy selects, for a write operation, a stripe-width of storage services. To model per-file optimizations, the client can overwrite system-wide configurations by requesting the manager to use a specific data placement policy. For example, the client may require that a file is stored locally, that is, on a storage service that is located on the same host. In this case, the manager attempts to allocate space on that specific storage service for that write operation. The file-specific data placement policy request is part of the workload description.

2.5 Model Seeding: System Identification

To instantiate the storage system model, one needs to specify the number of storage and client components in the system, and the service times for the network ($\mu^{localNet}$ and $\mu^{remoteNet}$) and each of the system components (storage read and write requests—μ^{smRead}, $\mu^{smWrite}$), manager—μ^{ma}, and client—μ^{cli}). The number of components in the system can be freely used by the predictor and depends on the scenario currently under the *what if analysis*. The service times are part of the platform description, and its identification is described in this section.

Compared to past work (e.g., Stardust [11], [19]), our approach focuses on making this *system identification process non-intrusive, as no changes are required to the storage system or kernel modules*. In addition to relying on application-level benchmarks, seeding relies only on a small deployment of up to only three machines, regardless of the size of the system simulated—keeping the identification process simple and low-cost. The model described in Section 2.3 captures the concurrency overheads (e.g., by having one key queue per component). While more accurate models or seeding processes are certainly possible the one we propose here proved to offer adequate accuracy for the application space of interest.

The process is automated with a script as follows: To identify the service time per chunk/request T^{net}), it runs a network throughput measurement utility tool (e.g., iperf), to measure the throughput of both remote and local (loopback) data transfers. The script measures the time to read/write a number of files to identify client and storage service time per data chunk. To this end, the system identification script deploys one client, one storage node and the manager on different machines, and writes/reads a number of files. For each file read/write, the benchmark records the total operation time, and computes the average read/write time (T^{tot}). The number of files read/written is set to achieve 95 percent confidence intervals with $\pm 5\%$ relative errors.

The operation total time (T^{tot}) includes the client side processing time (T^{cli}), the storage node processing time (T^{sm}), the total time related to the manager operations (T^{man}), and the network transfer time (T^{net}). The network service time for the network (μ^{net}) uses a simple analytical model based on the network throughput, and is proportional to the amount of data to be transferred in a packet. The storage processing time has two different requests, read and write, which are seeded in a similar way; due to space constraints, we represent them here as just one. The same is true for network service time, which be can local or remote.

To isolate just $T^{cli} + T^{man}$, the script runs a set of reads and writes of 0-size. This forces a request to go through the manager, but it does not touch the storage module. Since decomposing T^{cli} and T^{man} is not possible without probes in the storage system code, we opted to associate the $T^{cli} = 0$ and associate the whole cost of 0-size operations with the manager to obtain μ^{man} from T^{man}. Iperf can estimate T^{net}, and the script can infer $T^{cli} + T^{man}$, and therefore $T^{sm} = T^{tot} - T^{net} - T^{man}$. To obtain the service time per chunk, the times are normalized by the number of chunks.

While using application-level measurements is important to keep the seeding process simple and less costly—thus easy to be adopted, inferring the service per component is key to a wider range of configuration and provisioning scenarios.

2.6 Workload Description

The predictor takes as input a description of the workload, which contains three pieces of information: (i) per task I/O operations trace (i.e., open, read, write, close calls with timestamp, size, offset, and client id), (ii) application compute times between I/O requests, and (iii) a workflow execution plan (for scheduling and data placement purposes, see Fig. 2). Note that the first two characteristics above are independent of the number of nodes used in the cluster and can be collected by running the workflow on a single or limited number of nodes. The third characteristic can be easily simulated to explore the decision space.

The predictor preprocesses the description of the storage operations to potentially reduce the amount of events to be

simulated. Specifically, this phase aggregates some of the storage operations issued by the client, and estimates the impact of the cache size on the amount of data that each operation will actually transfer.

The client traces are obtained by running and logging the application operations. We have developed a FUSE wrapper to log the storage operations and the necessary scripts to preprocess the logs to infer the operations elapsed times and interarrival times based on timestamps, and create the events to be simulated. The execution plan can be provided by the workflow description (e.g., used by Swift [4]), by an expert, or extracted from log files. Currently, we use the execution plan from PyFlow scheduler [5].

3 Evaluation

This section presents the evaluation of the mechanism's prediction accuracy and, more importantly, demonstrates through a set of experiments the *mechanism's ability to support correctly identifying quasi-optimal configuration* for the scenario tested. To this end, we use a set of synthetic benchmarks and real applications. The synthetic benchmarks are designed to mimic the access patterns [9] of real workflow applications while stressing the system. To highlight how the prediction mechanism can be used in a real set-up, we use two real workflow applications, BLAST [14] and Montage [20], and demonstrate its ability to support intermediate storage configuration and provisioning decisions.

Storage system. We use MosaStore, an open source distributed object based storage system [5]. Except for experiments using Montage, the storage nodes are backed up by RAMDisk.[2]

The experiments also consider that the scheduler is data-location aware, by placing the tasks on nodes that already store the input data or by applying a data-placement optimized for a specific data-access pattern [9], [22].

Testbeds. To demonstrate the ability of our system to perform well independently of hardware deployment and scale, we use two testbeds. The first testbed (*TB20*) is part of our lab cluster with 20 machines. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4 GB RAM, and 1 Gbps NIC. The second testbed (*TB101*), used for larger scale experiments, includes 101 nodes on Grid5000 Nancy cluster [23]. Each machine has Intel Xeon X3440 4-core, 2.53-GHz CPU, 16 GB RAM, 1 Gbps NIC, and 320 GB SATA II.

In all the experiments, one node runs the metadata manager and the workflow coordination scripts, while the other nodes run the storage nodes, the client SAI, and the application processes. We point out that while network was shared with other applications, the network was not under provisioned, and there was no significant variability due to competing traffic. The simulator is seeded according to the procedure described in Section 2.5.

2. We choose to experiment with RAMDisks as they are frequently used to support workflow applications as intermediate storage: they offer higher performance [6], [7], [9], [10] and are the only option for compute nodes in many supercomputers as individual compute nodes do not have spinning disks [6], [7], [21] (e.g., IBM BG/P machines).

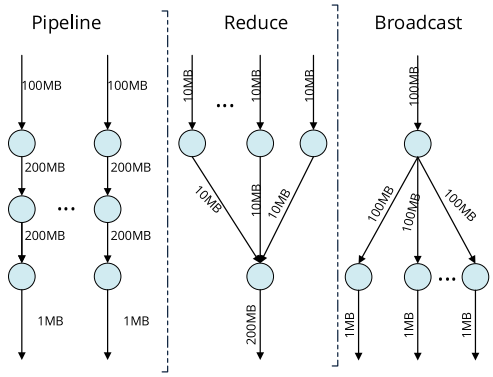


Fig. 4. Pipeline, Reduce, and Broadcast benchmarks. Nodes represent workflow stages and arrows represent data transfers through files. File sizes represent the *medium* workload. The part of the flow that is repeated, ran over 19 machines.

3.1 Synthetic Benchmarks: Workflow Patterns

This section evaluates the accuracy of the prediction mechanism with multiple clients, multiple data access patterns, and different data-placement policies designed to support workflow applications [9]. We use synthetic benchmarks that mimic the common data access patterns of workflow applications: pipeline, reduce, and broadcast (Fig. 4). These are the most popular patterns uncovered by studying over 20 scientific workflow applications by Wozniak and Wilde [1], Shibata et al. [2], and Bharathi et al. [3]). This past work shows that the workflow applications are typically a combination of these patterns and the code path does not significantly depend on input data.

The *synthetic benchmarks are designed to explore the limitations of the predictor* as they are composed exclusively of I/O operations, which stress network and storage components.

Summary of results. The predictor has good accuracy: our approach leads to prediction errors of 5 percent on average, lower than 8 percent in 80 percent of the studied scenarios, and within 14 percent in the worst case. More important, the mechanism correctly differentiates between the different configurations and can support choosing the best configuration for each evaluated scenario.

Experimental setup. We use the *DSS* label for experiments where we use the Default Storage System configuration: client and storage modules run on all machines, client stripes data over all 19 machines of the TB20 testbed, and no optimizations are enabled for any data-access pattern. We use the *WASS* label (Workflow Aware Storage System) when the system configuration is optimized for a specific access pattern (including data placement, stripe width or replication). All WASS experiments use data location aware scheduling: for a given compute task, if the input file chunks exist on a single storage node, the task is scheduled on that node to increase access locality.

The goal of showing results for two different configurations choices is two-fold: (i) demonstrate the accuracy of the predictions for two different scenarios, and (ii), more important, show that the predictions correctly indicate which configuration is the best. To understand the impact of the data size, for each benchmark, we use three workloads labeled as *medium* (data sizes are indicated in Fig. 4), the *small* (10× smaller than medium), and where possible, a 10× larger, *large* workload. We omit results for a *small* workload, since

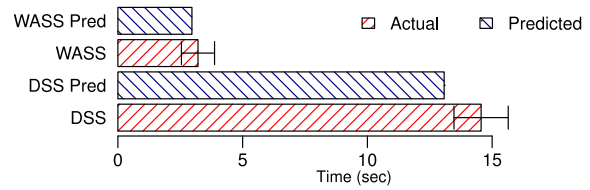


Fig. 5. Actual and predicted average execution time for the pipeline benchmark and medium workload.

it exhibits low variability between different configurations and the predictions are inside the confidence interval.

For actual performance, the figures show the average turnaround time and standard deviation (in error bars) for 15 trials (enough to guarantee a 95 percent confidence level).

Pipeline benchmark. A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task in the chain (Fig. 4). A pipeline-optimized storage system will store the intermediate files on the storage node co-located with the application. Later, the workflow scheduler places the task that consumes the file on the same node, increasing access locality. Here, 19 application pipelines run in parallel through three processing stages.

Evaluation of the results. For the optimized configuration (WASS) the predictor has almost perfect accuracy (Fig. 5). For the default data placement policy (DSS), however, predictions are 9 percent lower than actual results. For this case, all clients stripe (write) data to all machines in the system; similarly, all machines read from all others. This creates complex interactions among all system components leading to contention and chunk transfer retries due to connection initiation timeouts caused by network congestion, which are the main source of prediction inaccuracies.

Reduce benchmark. A single compute task uses input files produced by multiple tasks. Nineteen processes run in parallel on different nodes, consume an input file, and produce an intermediate file each. In the next stage, a single process reads all intermediate files and produces the final output. A possible data placement optimization is to use collocation—placing all intermediate input files on one node and exposing their location, then scheduling the reduce task on that machine. For WASS configuration, the collocation optimization is enabled for the files used in the reduce stage, for the remaining files the locality optimization is enabled.

Evaluation of the results. Similar to the pipeline benchmark, predictions for the reduce benchmark are close to the actual performance. In fact, they are within 9 percent of the actual average for medium workload, and 13 percent of the actual performance for large (Fig. 6). More important, they capture the relative improvements the pattern-specific data placement policies bring. We note that Fig. 6b captures the behavior of a heterogeneous scenario: to accommodate the amount of data produced, we used a faster machine with a larger RAMDisk to run the reduce stage. With proper seeding, the predictor captures the system performance with accuracy similar to a homogeneous system.

DSS data placement strategy places the data chunks on a random node, rendering two network transfers and

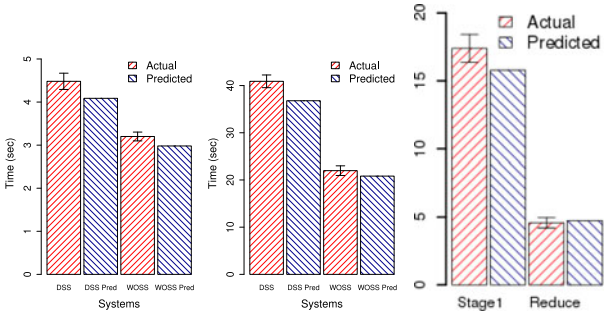


Fig. 6. Actual and predicted average execution time for the reduce benchmark for the medium, large workloads, and per stage for large workload.

storage device accesses. The WASS optimizes this I/O path transferring data directly to the reduce node. The challenge of capturing exactly the system behavior in DSS is similar to the pipeline case: complex interactions among all machines in the system. When the specific data placement is enabled though, there an additional challenge: high contention created by having several clients writing to the same storage node (the one that performs the reduce phase). Fig. 6c shows the results per-stage of the large workload to show how the predictor captures these cases.

Broadcast benchmark. Nineteen processes running in parallel on different machines consume a file that is created in earlier stage by one task. A possible optimization for this pattern is to create replicas of the file to be consumed by several tasks.

Evaluation of the results. We executed the broadcast benchmark with medium and large workload for the WASS configured with 1, 2, or 4 replicas. For this benchmark, all predictions match the actual results: predictions are inside the interval of mean and $\pm$ standard deviation, just 1-4 percent difference from the mean. Costa [10] presents in more detail. (There are no visual differences, so we omit the plot.) Creating replicas does not improve performance since data striping already avoids the contention on a single node. The predictor captures well the impact of different configurations.

3.2 The Pipeline Benchmark at Scale

This section expands the analysis of the synthetic benchmarks to answer the following questions: *How accurate are the predictor estimates for a different platform?*, and *How well does the predictor capture the behavior of larger scale systems?* To answer these questions, we ran the pipeline benchmark at larger scale on our Grid 5,000 testbed: TB101. We chose this benchmark because it is the one with the largest gap between predicted and actual, and it is the most I/O intensive—which stresses the network and the metadata manager, a well-known potential bottleneck for this storage system architecture. The results show a weak scaling set-up using three different scales (25, 50, and 100 nodes), the medium workload, and the two configurations (DSS and WASS).

Fig. 7 shows the results. The predictor produces estimates that differ on average by 14 percent from the actual

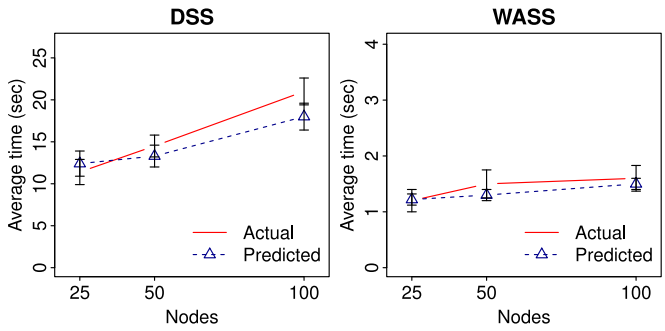


Fig. 7. Actual and predicted average execution time for the pipeline benchmark medium workload for 25, 50, and 100 nodes for testbed TB101.

time, are at worst within 20 percent of the actual results, and close to the interval delimited by the standard deviation.

3.3 Supporting Decisions for a Real Application

Section 3.1 evaluates predictor's ability to accurately estimate the turnaround time of synthetic benchmarks and the impact of different data placement optimizations. This section targets more complex scenarios where the user has to deal with a real application, allocation decisions, as well as the choice of the storage system configuration. Further, while the previous section evaluated prediction accuracy when the application and storage system are co-deployed, this section evaluates accuracy when they are deployed on separate nodes.[3]

This section demonstrates the predictor's ability to properly guide the user or a search algorithm to the desired configuration focusing on two provisioning scenarios:

- Scenario I assumes that the user has full access to a fixed-size cluster, a common set-up in several university research labs. Problem: *How should the system be partitioned between the application and the intermediate storage and what will be the intermediate storage system configuration for best overall performance?*
- Scenario II explores the provisioning problem with cost constraints (e.g., in cloud environments). Problem: *For a fixed workload; what is the cost/turnaround time trade-off space among the deployment options?*

Workload. We use BLAST [14] a DNA search tool for finding similarities between DNA sequences. In the BLAST workflow, each node receives a set of DNA queries as input and all nodes search the same DNA database file stored on intermediate storage. Each machine produces one output file, and the files are combined in the last workflow stage. The input files are transferred to the intermediate storage system prior to application execution.

Deployment scenario. Among the 20 machines of the testbed TB20, one node coordinates BLAST tasks' execution and runs the storage system manager. The remaining nodes either execute workload tasks or act as storage nodes.

Experimental methodology. The plots report the average of at least 20 runs, leading to 95 percent confidence intervals for all experiments. Since these are small (less than 5 percent of actual values), we omit them to reduce clutter.

3. Several factors may influence the decision of collocating or not the components (e.g., the amount of memory per node and the application memory footprint). Past work [5], [10], [11], [24] employed both approaches and, thus, we decided to explore both in our evaluation.

3.3.1 Scenario I: Configuring a Fixed-size Cluster

We explore the following question: Given a fixed size cluster, *how should the nodes be partitioned between the application and the intermediate storage, and what is the storage system configuration that yields the highest application performance?*

Fig. 8 shows workflow execution time for different partitioning and storage system configurations. In this case the *chunk size* is the configuration parameter that has the highest impact on performance, thus, to limit the points plotted in the figure, we focus on it only. (We note that the predictor correctly captures the lack of impact for other parameters). Additionally, we note that this evaluation covers a configuration parameter not covered in Section 3.1.

Fig. 8 highlights several points: First, the performance difference between the different configurations is significant: up to 10× difference between the best and the worst configuration even for the same chunk size. Second, the results show that the system achieves the best performance with a partitioning of 14 application nodes and five storage nodes, and chunk size of 256 KB (4× smaller than the default size) a non-obvious configuration beforehand. Third, the experiment shows that the predictor accurately captures the system performance under different partitioning strategies, and storage system configurations. The overall error of the predictions are small (always within the standard deviation), and smaller than obtained for synthetic benchmarks since there is less stress on the storage system. Finally, the most important point is that the predictor can correctly lead the user or a search algorithm to the desired configuration.

3.3.2 Scenario II. Provisioning in an Elastic and Metered Environment

This scenario assumes an environment where users are charged (proportional to the cumulative CPU-hours used) and have a more complex tradeoff between cost and time-to-solution, for example, they aim for the best application turnaround within a certain dollar budget. We aim to inform the user's provisioning decision by revealing the details of this trade-off. Specifically, this scenario helps the user to answer the following question: *What is the allocation size, and how should it be partitioned and configured to best fit the constraints and optimization criterion?*

Fig. 9 shows the application execution time and allocation cost (measured as number of nodes x allocation time) for different cluster sizes, different partitioning, and different chunk sizes. Similar to Scenario I, Fig. 9 shows that the predictor captures the system performance with significant accuracy.

Fig. 9 also shows that an allocation of 11 nodes, with partitioning of eight application, two storage nodes, and chunk size of 256 KB offers the lowest cost. More importantly, this plot points out an interesting case for the analysis of cost verus time-to-solution: The user can analyze it to verify that an option with an allocation of 20 nodes actually offers almost 2× higher performance at a marginally 2 percent higher cost.

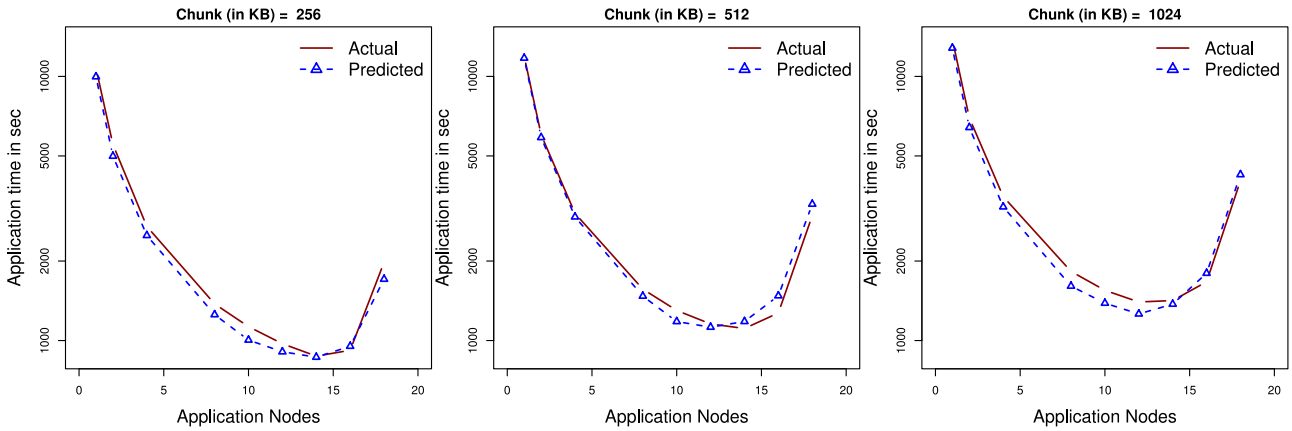


Fig. 8. Application runtime (log-scale) for a fixed-size cluster of 20 nodes. X-axis represents number of nodes allocated for the application (storage nodes = 20 - application nodes). The three plots represent runtime for different configurations (chunk sizes).

3.4 A Large, Complex Workflow: Montage on TB101

This section aims to answer the following question: *Can the predictor support user decisions for more complex scenarios?*. Specifically, we are interested in a workflow composed of more stages, tasks, and patterns executing a more data intensive workload at a larger scale. To answer this question, we focus on evaluating how accurate the estimates are for Montage [20], a complex astronomy workflow composed of 10 different stages (Fig. 10), with varying characteristics in terms of number of tasks, volume of data, and access patterns. The workflow uses the reduce pattern in two stages and the pipeline patterns in four stages (as indicated by the labels on the arrows in Fig. 10). We point out that Montage diversity offers a rich usecase for the predictor and, in fact, each stage could be seen as a different application. Zhang et al. [24] present a detailed characterization of Montage.

To verify that the predictor can support configuration decisions, we have executed Montage for different deployments sizes on TB101. We use clients collocated with storage nodes, verifying yet another configuration parameter. We omit chunk size variation since it does not impact Montage performance, which is well captured by the predictor.

Workflow characteristics. The I/O communication intensity between workflow stages is highly variable (presented in Table 1 for the workload we use). Overall, the workflow has over 7,500 tasks and generates over 10,000 files with sizes from 2 KB to over 3 GB; about 30 GB of data is read/written from/to storage.

Evaluation of the results. Fig. 11 shows the actual and predicted runtime on different allocation size. We report the average of 10 runs; the standard deviation is approximately 3 percent and omitted to reduce clutter. Overall, the predictor captures well the application performance: despite the complexity of the workflow and the scale, the predictor is accurate with average prediction error being 3 percent (the smallest $<$ 1 percent; the largest $<$ 7 percent).

Prediction accuracy per stage. Since I/O intensity and access patterns vary widely across stages, we analyze prediction accuracy at stage-level: the average error per stage is 5 percent, with the five stages (mProject, mDiff, mFitPlane, mAdd, and mJpeg) that account for over 70 percent of the runtime having a maximum 4 percent error. The highest error was 25 percent for mBackground on five nodes, which we believe is because of the short duration of each task (less than 2 s) which makes it sensitive to any variation in the platform.

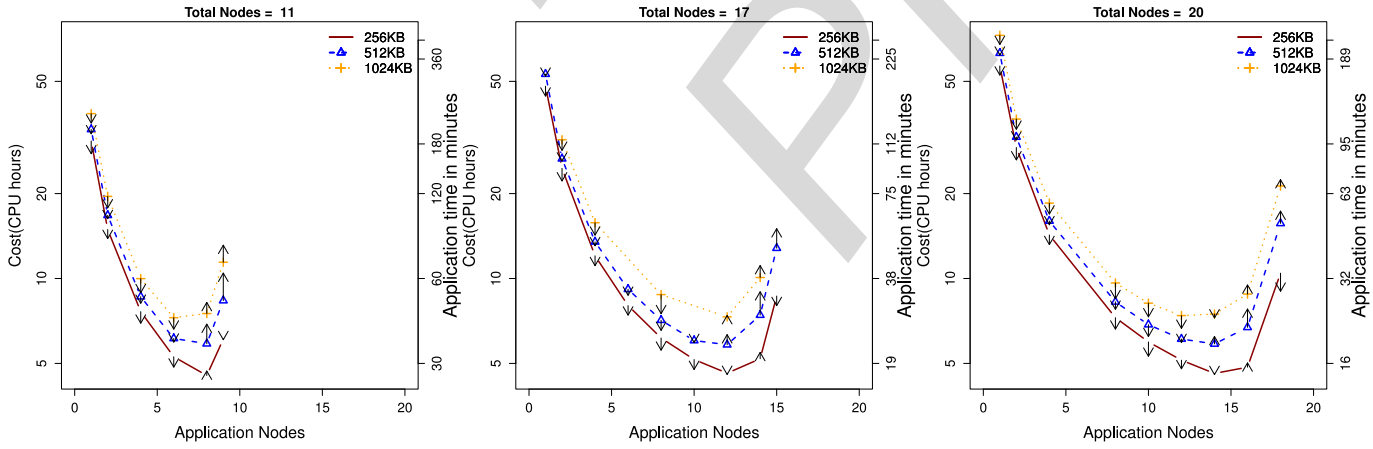


Fig. 9. Allocation cost (total CPU-hours on the left Y-axis, log-scale) and application turnaround time (right Y-axis, log-scale, different scale among plots) for fixed size clusters of 11, 17, and 20 nodes while varying the chunk size. X-axis represents number of nodes allocated to the application (storage nodes = 20 - application nodes). The figures show the actual (lines) and the predicted (arrows) cost/performance. Note: Log scale on both Y-axes.

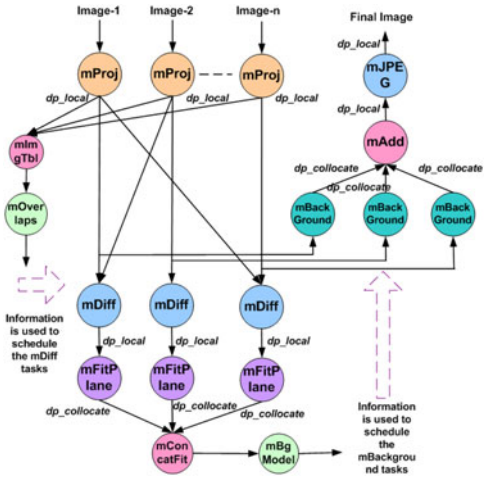


Fig. 10. Montage workflow. Labels on the arrows represent the tags used to indicate the data usage patterns.

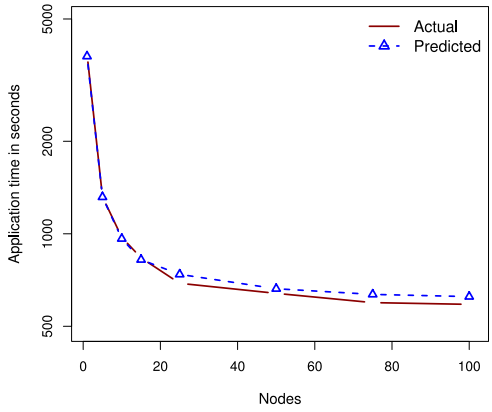


Fig. 11. Montage time to solution (y-axis in log scale) for varying fixed size deployments on TB101.

Results on TB20. To explore a different execution platform, an additional evaluation considers TB20 [10]. The workload is smaller; the workflow generates over 650 files, with sizes ranging from 1 KB to over 100 MB, and about 3 GB of data are read from or written to the storage system. Overall, the predictions obtain "good-enough" accuracy to support the user's decisions about provisioning. The average prediction error is 8 percent, the smallest is less than 1 percent, and the maximum prediction error is less 15 percent.

3.5 Prediction Effort and Scalability

This section demonstrates that the predictor satisfies the response time and scalability requirements listed in Section 2.1. Note that using the predictor requires: (i) [once-per-application] collecting the application trace—typically, a collecting the trace leads to a 10 percent overhead for a worst-case I/O intensive application, and (ii) [once-per-platform] profiling the platform, which also includes the time of housekeeping (i.e., deploy, initialize, and shutdown) tasks related to the storage system and takes less than 5 minutes in our experiments.

Prediction effort. Overall, the predictor uses orders of magnitude less resources (i.e., CPU hours) than the corresponding workflow execution: for BLAST, 500× (approximately 4.5 CPU hours versus 32 seconds); for Montage 600× (844 CPU seconds versus 1.41 seconds), for reduce benchmark medium workload, 2,000× less CPU time (81 CPU seconds versus 41 CPU ms). All reported numbers are for TB20 when using all 20 nodes, taking into account only the processing phase (no setup time) in the fastest configuration of the workflow application.

Scalability. The simulation time scales linearly with the number of events generated by the workflow application, which is proportional to the data size. Predicting WASS is shorter than DSS as the data placement optimizations generate fewer (network) events. In a weak scaling experiment in which the data size scales up with the number of nodes up to 1,000 nodes and time is measured in seconds, slopes are approximately 0.03 for WASS, and 0.038 for DSS.

TABLE 1
Characteristics of Montage Workflow Stages

Stage	Data	#Files	#Tasks	File Size
stageIn	1.9 GB	957	1	1.7 MB-2.1 MB
mProject	8 GB	1,910	955	3.3 MB-4.2 MB
mImgTbl	17 KB	1	1	
mOverlaps	336 KB	1	1	
mDiff	2.6 GB	5,640	2,833	100 KB - 3 MB
mFitPlane	5 MB	1,420	2,833	4 KB
mConcatFit	150 KB	1	1	
mBgModel	20 KB	1	1	
mBackground	8 GB	1,913	955	3.3 MB - 4.2 MB
mAdd	5.9 GB	3	1	165 MB-3 GB
mJPEG	47 MB	1	1	
stageOut	3 GB	2	1	47 MB-3 GB

4 Predicting Energy Consumption

The previous section presents the design of the prediction mechanism focusing on traditional performance metrics such as workflow runtime or cost. This section uses a simple model based on power profiles to extend this mechanism to make energy predictions, a concern of growing importance due its impact on cost or even on the feasibility of building larger computing infrastructures. Due to space limitations, we present here a compressed description of our experience, Yang [25] provides a detailed description of this experience.

4.1 Energy Model Extension

A typical task of a workflow application progresses in multiple rounds, with little overlap, of: (i) the machine running the task reads some (or all) input data from the intermediate storage (likely through multiple I/O operations), (ii) the task processes this data, (iii) the machine writes the output to the intermediate storage.

To estimate the energy consumption of a task the model associates a different power profile with each processing stage: (i) *idle profile*—power to keep the machine on (P^{idle}); (ii) *application processing profile*—extra power drawn to run a task on the CPU (P^{app}); (iii) *local storage profile*—additional power to perform read/write operations on a storage node (P^{sm}); and (iv) *network profile*—additional power to perform network data transfers (P^{net}).

The total energy spent during a workflow application execution can be approximated as the sum of the energy consumption of individual nodes $E_n^{tot} = E_n^{idle} + E_n^{app} + E_n^{sm} + E_n^{net}$. E_n^{idle} is the energy consumed to keep the node n on during the whole workflow runtime: $E_n^{idle} = P_n^{idle} \times T_{app}$. At same time, E_n^{app}, E_n^{sm}, and E_n^{net} are the *additional* energy consumed to keep the node to each of the power states above: running the task, serving local storage, and transferring data (e.g., $E_n^{app} = T_app \times (E_n^{app} - E_n^{idle})$). To estimate these, the predictor needs to obtain two sets of data. First, the power characteristics of the platform, which can be obtained through a seeding process that stresses the CPU, storage, and network components of the platform.

4.2 Energy Evaluation

Similarly to Section 3, the evaluation uses synthetic benchmarks and real workflow applications on different storage configurations and power tuning techniques of the platform (DVFS). The experimental setup is the same as in Section 3.

Testbed. Infrastructure to measure power is available on a smaller testbed of Grid5000, the 'Lyon' cluster [23]: each node has two 2.3 GHz Intel Xeon E5-2630 CPU, 32 GB RAM and 10 Gbps NIC. This platform limits the scale of the experiments to 11 nodes. The seeding obtained, $P^{idle} = 91.6$ W, $P^{App} = 33.6$ W, $P^{sm} = 37.5$ W, and $P^{net} = 36.1$. P_{peak} (not used for seeding) is 225.3 W.

Power measurements. Each node is connected to a SME OmegaWatt power-meter, with 0.1 W power resolution at 1 Hz sampling rate. To measure the total energy consumption, the scripts that run the experiments aggregate the power consumption for the duration of the benchmark execution. The evaluation does not take into account the energy consumed by the node that runs metadata service and workflow scheduler as these are not subject to the configuration changes that the prediction mechanism targets.

Energy predictions for synthetic benchmarks. Due to space constraints we do not present the detailed experiments. Overall, the energy consumption predictions have an average of 12.5 percent error and typically close to one standard deviation interval. For DSS, the pipeline benchmark has the best accuracy with 5.2 percent error, reduce is 16.4 percent, and broadcast is 15.9 percent. For WASS, predictions have 13.4 percent prediction error for pipeline, 12.2 percent for reduce, and 16 percent for broadcast.

Importantly, although the prediction mechanism is more accurate for runtime than energy, it is accurate enough for both DSS and WASS to support storage configuration choices when energy is used as the optimization criterion.

Energy predictions for real applications. To understand how accurate the predictor is for real applications, we use again BLAST and Montage. Similar to time predictions, overall, the energy predictions are more accurate for the real applications than for synthetic benchmarks since the synthetic benchmarks are designed to produce a high stress on the storage system, which results in contention and higher variance that are harder to capture. BLAST predictions exhibit an 5.2 percent average error while Montage 15.9 percent.

Predicting the impact of DVFS techniques. DVFS (also known as CPU throttling) is a technique that limits processor frequency to reduce power consumption. This has the drawback of also limiting the number of instructions a processor can issue in a given amount of time potentially increasing application runtime. Thus, although this technique saves power, it is not clear whether (or when) reduces energy consumption.

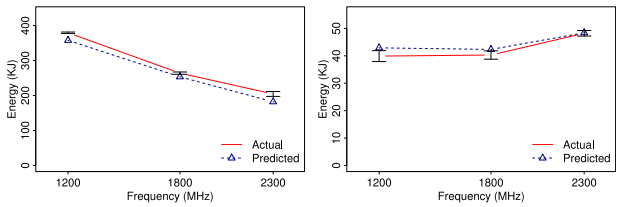


Fig. 12. Actual and predicted average energy consumption for BLAST (left) and pipeline benchmark (right) for various CPU frequencies.

To evaluate the predictor's ability to guide CPU frequency scaling choices, Fig. 12 presents energy consumption for two benchmarks: (i) BLAST (left), and (ii) the pipeline benchmark (right), performing only I/O operations. DVFS is used to set the processors at 1.2, 1.8, and 2.3 GHz. The seeding procedure is performed independently for each frequency.

BLAST is more CPU intensive: reducing the frequency increases the runtime and thus the energy consumed, leading to 85.5 percent more energy consumption between the minimum and maximum frequencies evaluated. Pipeline benchmark is not strongly affected by CPU performance, thus, using minimum frequency does not increase the benchmark execution time. More importantly, the predictions are useful to support configuration decisions.

5 Using the Predictor to Support System Development

This section describes our experience with employing this tool to support the development of MosaStore distributed storage system [5]. Our goal is to investigate the following questions: *How can a performance predictor bring benefits to the software development process?*, and *What are the challenges of using the predictor as part of the development process?* A technical report describes this experience in more detail [26].

Developing a distributed system is a complex and error-prone task. Performance should be addressed since the early development stages [27], especially given that maintenance and debugging costs increase over time. The state-of-the-practice consists of analyzing a system using *profilers* to understand its behavior [27]. Although profiling-based optimization is undoubtedly a key part of the development process, deciding when the efficiency of a system has reached a "good enough" level, and additional effort is unlikely to render benefits, is still a challenge. Developers should be able to rely on tools that provide an estimate of the expected performance (i.e., a baseline) and compare it to the actual performance to decide whether to invest time in performance-oriented debugging.

Similar to back of the envelope calculations, a predictor indicates performance bounds for the system. Due to its ability to model complex scenarios, the predictor can, however, be used where back of the envelope estimates are intractable or inaccurate. Not only developers can use it to

obtain a baseline to detect performance anomalies, but also to evaluate the potential gains of implementing new complex optimizations.

5.1 Development Process Changes

The performance predictor was included in the development process as part of the testing suite. The developers ran a selected set of synthetic benchmarks on the top of MosaStore, measured system performance (Section 3) and compared it to the performance estimates produced by the predictor. Large discrepancies between the actual performance and the estimates were flagged as performance anomalies.

To debug performance anomalies, the developers can turn on a log option that measures the response time for each internal storage system request. By comparing to the predicted response time, the developers can narrow down the potential cause of anomalies. The developer then can benchmark the specific code path to investigate further.

Fixing anomalies using this process improved the system performance by up to 30 percent for the tested benchmarks and decreased the response time variance by up to 10× as described next.

5.1.1 Our Experience: Cases of Improvement

This section presents three cases where we detected performance anomalies.

P1: Lack of randomness. When a client creates a file, it contacts the manager to obtain a list of storage nodes to use. The manager randomizes the order of storage nodes returned. However, the manager used the same seed to shuffle the node list. Thus, when the system was configured to use the maximum stripe-width, all clients obtained the same list, issuing operations to the storage nodes in the same order and creating temporal hot-spots.

P2: Lock overhead. Concurrent reads or updates to the metadata at the manager needs to be serialized to avoid metadata corruption. The developers of the initial manager version chose a conservative approach by opting to lock large code blocks, rather than locking only the critical regions. By comparing the actual and predicted manager's service times, the developers detected large and unnecessary lock scopes, and reduced them. Using finer grain locking brings 15 percent performance improvement (2.5 s) and a decrease in variance as shown in Fig. 13a, and combined with the lack of randomness, in Fig. 13b.

P3: Connection timeout. If a client fails to establish a TCP connection, it waits for a specific timeout before retrying. When too many clients tried to send data to a single storage node at the same time (as in the reduce pattern), the storage node dropped SYN packets (used to establish TCP connections). The client then waited for 3 seconds—the timeout defined by the TCP OS-implementation, taking the average write time to a longer value than expected. The developers observed a bi-modal distribution in the service time where most of the requests were processed in the predicted time, while the others 3 s later. The developers were able to assign this to TCP connection retries and implemented a customized retry mechanism. The gain, for the reduce benchmark: 30 percent and 10× less variance (Fig. 13b).

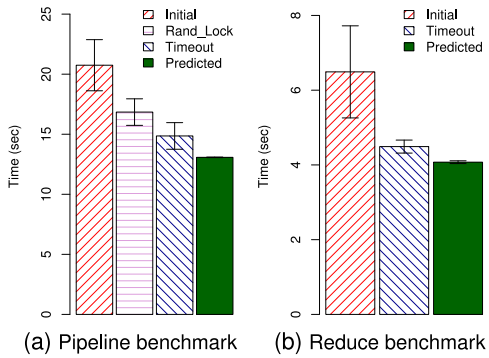


Fig. 13. Impact of fixing performance issues for pipeline and reduce benchmarks. Impact of fixing performance issues. '*Randlock*' bar shows results for fixing two performance anomalies together (P1 and P2). Error bars show the standard deviation (15 trials, 95 percent confidence interval).

6 Related Work

This section describes previous work on different approaches to predict storage system performance and tune its configuration parameters.

Model based analysis. A number of projects use model-based approach to estimate the storage system performance with a given configuration or workload. Ergastulum [28] targets centralized storage solution based on one enclosure to recommend an initial system configuration, and Hippodrome [29] relies on Ergastulum to improve the configuration based on online monitoring of the workload. By considering a distributed system, our solution handles more complex interaction among the system components and more configuration options.

Simulation based systems. IMPIOUS (Imprecisely Modeling Parallel I/O is Usually Successful) [30] is a trace-driven simulator that uses an abstract storage system models designed to capture the key mechanism of parallel file systems. The simulator is simplified to be able to simulate thousands of client and storage nodes. Consequently the simulator is not accurate, producing performance estimates that under- or over-estimates the performance by up to 50 percent. PFSsim [31] is a trace-driven simulator designed specifically for evaluating I/O scheduling algorithms in parallel file systems. PFSsim simulates the storage system at low component level, simulating the network using OMNeT++ and disks using DiskSim [32]. Liu et al. [33] build a simulation framework for simulating the storage system of supercomputing machines. The framework simulates all hardware components including compute and IO nodes, storage subsystem, and the supercomputing interconnect. FileSim [34] is a parallel file system simulation framework that simulates file system operations at low granularity, including disk operations and packet level simulation of network operations. The simulator simulates specific storage system operations, such as data placement, or locking algorithms, and can be used to validate new algorithms or metadata service. Similar to this work, Thereska et al. [11] proposed a predictor mechanism for a distributed storage system with a detailed model. To provide such information, they propose Stardust [19] a detailed monitoring information system that required changes to the storage system and kernel modules to add monitoring points. This

approach enabled their predictor to achieve prediction within 18 percent of the actual predictions depending on the workload. Our approach has achieved similar accuracy given our target workloads, however with a lightweight approach to seed the model, not requiring changes to the system design or kernel modules.

Unlike these efforts, our approach targets simulating a generic and most common distributed storage system architecture in a locally interconnected infrastructure without simulating operations of a particular storage system. Our approach avoids detailed low-level simulation (e.g., disk or packet level simulation), without significantly compromising accuracy, enabling our framework to efficiently simulate large-scale deployments. Additionally, our approach also covers the seeding procedure via application-level measurements, not requiring human-input, changes to the storage system, or specialized monitoring.

An important difference to past work on storage systems simulation is our focus on a whole workflow application and the potential interaction among the workflow's phases and scheduling decisions instead of the average performance for a batch (e.g., [28], [29]) of operations, and of predicting performance of the system from the perspective of just one client [11]. Additionally, our work targets the partitioning problem of splitting the nodes among application and intermediate storage.

Monitoring and/or machine learning based tuning. Several projects explore the configuration space by running the application under different configurations (e.g., DataSteward [35]). Behzad et al. [36] present an auto tuning framework for HDF5 IO library, their solution intercepts the HDF5 IO calls and injects optimized parameters into parallel I/O calls. Further, the framework monitors the I/O performance and explores the tuning parameter space using a genetic algorithm. ACIC [37] uses a CART-based model (Classification and Regression Trees) to build the model used to guide the optimization of parallel applications. Zhang et al. [24] propose an approach to determine the storage bottleneck for workflow applications using a set of benchmarks and target workflow application runs. Crume et al. [38] use machine learning to model disk behaviour. Elastisizer [39] targets a similar problem of supporting configuration choices for an entire application, it focuses on a different class of application: Map Reduce jobs.

The approach we propose enables an exploration of the system at a lower cost since the predictor is able to estimate performance of a scenario that adds or reduces resources and to change the configuration without requiring new runs (or a larger training set) of the application for new generations of the genetic algorithm. Further, unlike [36], [37] we target workflow applications and predicting a POSIX-based storage system performance, including the impact of scheduling decisions in combination with data placement. In fact, our proposed solution can be used through an adaptation of these machine learning techniques, or other optimization approaches, to our target context in order to perform autotuning.

Finally, the accuracy we obtained is similar to or better than past work that predicted the behavior of distributed storage systems or distributed applications in order to support configuration decisions (e.g., Thereska et al. [11], [19], Elastizer [39])

7 Conclusions and Discussion

Summary. This paper makes the case for a prediction mechanism to support automating configuration and provisioning choices for workflow applications. We focus on workflow applications when running on top of an intermediate object-based storage system. We propose a solution based on a queue-based model with a number of attractive properties: a generic and uniform system model; supported by a simple system identification process that does not require specialized probes or system changes to perform the initial benchmarking; with a low runtime to obtain predictions; and, finally, with adequate accuracy for the cases we study. We demonstrate the utility of the predictor in scenarios that include optimizations focussed on time-to-solution (turnaround time), cost (total node hour), and energy.

Our experience demonstrates that we can rely on some characteristics of the workflow applications to provide a relatively simple system model that can accurately support provisioning and configuration decisions in a complex and potentially large configuration. The discussion below aims to clarify our understanding of other limitations of our work and the lessons we have learned so far.

What are the main sources of inaccuracies? Currently, there are sources of inaccuracies at multiple levels. First, the model does not capture all the details of the storage system. For example, support services like garbage collection or storage node heartbeats or the control paths are simplified to match a generic object-based storage (MosaStore uses a FUSE-based implementation that would need more complex control path), and this model assumes that all control messages are of the same size. Second, the system identification mechanism is constrained to be simplified even further, at the cost of additional accuracy loss. Third, the model does not capture the infrastructure in detail (e.g., contention at the network fabric level, OS scheduling). Finally, the application driver uses an idealized image of the workflow application (e.g., we do not model workflow manager overheads, task launch overheads). We believe the latter one is the main reason of current inaccuracies in the system. In fact, an initial evaluation showed that Montage overhead overhead can be up to 5 percent of the total time.

Could a less detailed model and a simpler system identification provide similar results? We aim to model only the key interactions between system components. Modeling all system subcomponents and all their interactions in detail would be too complex. Such complexity could improve prediction accuracy, but would have significant drawbacks: a more complex model (as complex as the actual storage system and the underlying environment (e.g., network protocols, operating system buffers, scheduling), complex seeding process, lower scalability, and loss of the model generality. Further, the improvement in accuracy may not add much value (e.g., when the prediction mechanism is used to decide between system configurations). We followed a top-down approach: we started from a simple model and added more components or interaction details until the accuracy of the all predictions was within 10 percent of actual performance (and the median error was within 5 percent).

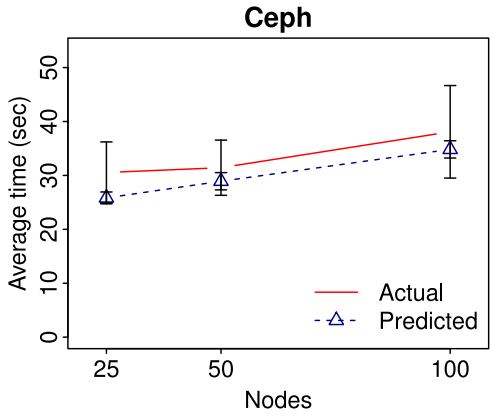


Fig. 14. Actual and predicted performance for the pipeline benchmark on TB101 using Ceph on spinning disks.

How general is the proposed prediction mechanism? The goal of the prediction mechanism is to model a generic object-based storage system and have a system identification process that works entirely at the application level, to be easily portable across storage systems and deployment platforms. While, so far, the predictor has been evaluated in depth for the MosaStore system in its DSS/WASS configurations, we also have encouraging results when using it to predict the relative application performance of the pipeline benchmarks on DSS in two other storage solutions, Ceph [40] and GlusterFS [41].

The experiments use the pipeline benchmark using the same traces used in our evaluation of MosaStore predictions, no new traces were collected. Seeding is the only input not shared among the systems.

Fig. 14 shows the actual and the predicted performance for the pipeline benchmark using Ceph deployed on spinning disks. This experiment also varies the number of nodes for TB101. Predictions for the average performance are within 15 percent of the values for the actual performance. Additionally, Ceph has a high standard deviation, which places the predictions overlapping the intervals considering the standard deviations for all scales. For GlusterFS, the predictor obtained error around 25 percent on top of 100 nodes.

More importantly, this accuracy is "good-enough" to compare the performance among different scale and evens those systems and MosaStore, capturing which one should provide the best performance.

How accurate is the prediction mechanism when the intermediate storage is deployed on spinning disks? Since we have focused on the intermediate storage deployed over RAMDisks (see Sections 1 and 3), the storage service does not model history-dependent behaviour. Thus, it is expected to achieve lower accuracy predictions when the system is deployed over spinning disks, which can be mitigated by integrating a more sophisticated model or simulator of the storage [32], [38].

A preliminary evaluation, however, shows how the current (unchanged) model performs when using spinning disks [10], on testbed TB20, for the synthetic benchmarks described in Section 3.1. The key observation here is that, although prediction accuracy is lower (approximately 20 percent), predictions are good enough to make the correct choice between DSS and WASS—that is, the choice of whether to use the data co-placement optimization for each of the workloads.

The results when using TB101, which has newer machines than TB20, and MosaStore deployed on spinning disks are similar to those of the RAMDisks as well.

Additionally, Fig. 14 presents the predicted runtime for pipeline benchmark running on top of Ceph storage system deployed on spinning disks using TB101.

References

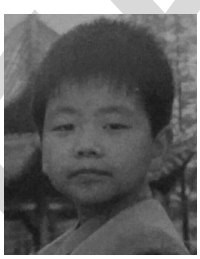
[1] J. M. Wozniak and M. Wilde, "Case studies in storage access by loosely coupled petascale applications," in *Proc. 4th Annu. Workshop Petascale Data Storage*, 2009, pp. 16–20.
[2] T. Shibata, S. Choi, and K. Taura, "File-access patterns of data-intensive workflow applications and their implications to dist. filesystems," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 746–755.
[3] S. Bharathi, A. Chervenak, et al., "Characterization of scientific workflows," in *Proc. 3rd Workshop Workflows Support Large-Scale Sci.*, 2008, pp. 1–10.
[4] M. Wilde, M. Hategan, et al., "Swift: A language for distributed parallel scripting," *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, 2011.
[5] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu, "The case for a versatile storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 10–14, 2010.
[6] N. Liu, J. Cope, et al., "On the role of burst buffers in leadership-class storage systems," in *Proc. 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.
[7] J. Lofstead, F. Zheng, et al., "Adaptable, metadata rich IO methods for portable high performance IO," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–10.
[8] J. Dayal, D. Bratcher, et al., "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2014, pp. 246–255.
[9] L. B. Costa, H. Yang, et al., "The case for workflow-aware storage: An opportunity study," *J. Grid Comp.*, vol. 13, pp. 95–113, 2015.
[10] L. B. Costa, "Support for configuration and provisioning of intermediate storage systems," Ph.D. dissertation, Univ. of British Columbia, Vancouver, BC, Canada, 2015.
[11] E. Thereska, M. Abd-El-Malek, et al., "Informed data distribution selection in a self-predicting storage system," in *Proc. 3rd Int. Conf. Autonomic Comput.*, 2006, pp. 187–198.
[12] J. D. Strunk, E. Thereska, et al., "Using utility to provision storage systems," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 313–328.
[13] M. Abd-El-Malek, W. V. Courtright II, et al., "Ursa minor: Versatile cluster-based storage," in *Proc. Conf. File Storage Technol.*, 2005, p. 5.
[14] S. F. Altschul, W. Gish, et al., "Basic local alignment search tool," *J. Molecular Biol.*, vol. 215, no. 3, pp. 403–410, 1990.
[15] (2012). The network simulator NS2 [Online]. Available: http://www.isi.edu/nsnam/ns/
[16] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. 9th Int. Conf. Peer-to-Peer*, 2009, pp. 99–100.
[17] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, vol. 2000, no. 80es, pp. 1–8, 2000.
[18] Z. Zhang, D. S. Katz, et al., "Design and analysis of data management in scalable parallel scripting," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 85:1–85:11.
[19] E. Thereska, B. Salmon, et al., "Stardust: Tracking activity in a distributed storage system," in *Proc. Joint Int. Conf. Meas. Modeling Comput. Syst.*, 2006, pp. 3–14.
[20] A. C. Laity, N. Anagnostou, et al., "Montage: An astronomical image mosaic service for the NVO," in *Proc. Astron. Data Anal. Softw. Syst. XIV*, vol. 347, pp. 34–38, 2005.
[21] D. P. Siewiorek and P. J. Koopman, *The Architecture of Supercomputers: Titan, a Case Study*. New York, NY, USA: Academic, 2014.
[22] Z. Zhang, D. S. Katz, et al., "Ame: An anyscale many-task computing engine," in *Proc. 6th Workshop Workflows Support Large-Scale Sci.*, 2011, pp. 137–146.
[23] F. Cappello, E. Caron, et al., "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Proc. 6th IEEE/ACM Int. Workshop Grid Comput.*, 2005, pp. 99–106.
[24] Z. Zhang, D. S. Katz, et al., "Mtc envelope: Defining the capability of large scale computers in the context of parallel scripting applications," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 37–48.

[25] H. Yang, "Energy prediction for I/O intensive workflow applications," Master's thesis, The Univ. of British Columbia, Vancouver, BC, Canada 2014.
[26] L. B. Costa, J. Brunet, et al. (2013). Experience on applying performance prediction during development: A distribution storage system tale, *2nd Intl. Workshop Software Eng. for High Perf. Comp. in Computational Science and Eng. (SE-HPCCSE)*, 2014, pp. 13–19.
[27] S. Balsamo, A. Di Marco, et al., "Model-based Perf. prediction in software development: A survey," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 295–310, May 2004.
[28] E. Anderson, S. Spence, et al., "Quickly finding near-optimal storage designs," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 337–374, 2005.
[29] E. Anderson, M. Hobbs, et al., "Hippodrome: Running circles around storage administration," in *Proc. Conf. File Storage Technol.*, 2002, pp. 175–188.
[30] E. Molina-Estolano, C. Maltzahn, et al., "Building a parallel file system simulator," in *J. Phys.: Conf. Series*, vol. 180, no. 1, 2009.
[31] Y. Liu, R. Figueiredo, et al., "Towards simulation of parallel file system scheduling algorithms with pfssim," in *Proc. 7th IEEE Int. Workshop Storage Netw. Archit. Parallel I/O*, 2011, pp. 1–9.
[32] (2015). Disksim [Online]. Available: http://www.pdl.cmu.edu/DiskSim/
[33] N. Liu, C. Carothers, et al., "Modeling a leadership-scale storage system," in *Proc. 9th Int. Conf. Parallel Proc. Appl. Math. - Vol. Part I*, 2012, pp. 10–19.
[34] M. Erazo, T. Li, et al., "Toward comprehensive and accurate simulation perf. prediction of parallel file systems," in *Proc. 42nd IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12.
[35] R. Tudoran, A. Costan, and G. Antoniu, "Datasteward: Using dedicated compute nodes for scalable data management on public clouds," in *Proc. 12th IEEE Int. Conf. Trust, Security Privacy Comput. Commun.*, 2013, pp. 1057–1064.
[36] B. Behzad, S. Byna, et al., "Taming parallel i/o complexity with auto-tuning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 68:1–68:12.
[37] M. Liu, Y. Jin, et al., "ACIC: Automatic cloud i/o configurator for HPC applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 111–112.
[38] A. Crume, C. Maltzahn, et al., "Fourier-assisted machine learning of hard disk drive access time models," in *Proc. 8th Parallel Data Storage Workshop*, 2013, pp. 45–51.
[39] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 18:1–18:14.
[40] S. A. Weil, S. A. Brandt, et al., "Ceph: A scalable, high-perf. dist. file system," in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, 2006, pp. 307–320.
[41] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux J.*, vol. 2013, no. 235, pp. 72–82, Nov. 2013.

Lauro Beltrão Costa joined Google after receiving his PhD from the ECE Department at the University of British Columbia (UBC). Before, he received BSc and MSc degrees from UFCG, Brazil, where he worked on the OurGrid project. He is interested in distributed systems with focus on high-performance systems.

Samer Al-Kiswany is a PostDoc fellow in the University of Wisconsin—Madison and received his MSc and PhD degrees from the ECE Department at UBC. He is a PostDoc fellow in the University of Wisconsin–Madison. His interests are distributed systems with focus on high performance computing systems, and cloud computing.

Matei Ripeanu received his PhD degree in Computer Science from the University of Chicago in 2005 before joining the ECE Department of UBC. Matei is interested in distributed systems, focusing on self-organization and decentralized control in large-scale systems. His research group's work can be found at http://netsyslab.ece.ubc.ca.



Hao Yang joined Amazon, Canada after receiving his MASc degree in the ECE Department at UBC.