Supporting Storage Configuration for I/O Intensive Workflows

Lauro Beltrão Costa, Samer Al-Kiswany, Hao Yang, Matei Ripeanu
NetSysLab, Electrical and Computer Engineering Department
University of British Columbia, Vancouver, BC, Canada
{lauroc,samera,haoy,matei}@ece.ubc.ca

ABSTRACT

System provisioning, resource allocation, and system 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) and configuring each of these optimally (e.g., replication level, chunk size, caching policies in case of storage) all having a large impact on overall application performance. This paper presents our progress on addressing 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. Our evaluation shows that: (i) the predictor is effective in identifying the desired system configuration, (ii) it can scale to model a workflow application run on an entire cluster, while (iii) using over 2000x less resources (machines x time) than running the actual application.

Categories and Subject Descriptors

D.4 [**Operating systems**]: [Storage management]; D.4.8 [**Performance**]: Modeling and Prediction

Keywords

performance prediction; distributed storage systems

1. INTRODUCTION

Assembling workflow applications by putting together standalone binaries has become a popular approach to support large-scale science [9, 21, 28]. 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 [27]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'14, June 10–13 2014, Munich, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597679.

To avoid accessing the platform's backend storage system (e.g., NFS or GPFS or Amazon S3), recent proposals [4, 28] advocate using some of the nodes allocated to the application to deploy a *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.

This approach offers a number of advantages: higher performance - as applications benefit from a wider I/O channel obtained by striping data across several nodes; higher efficiency – as it improves resource utilization; incremental scalability – as it is possible to increase system 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 (e.g., specify chunk-size to optimize data-transfers, configure striping and replication to eliminate hot spots, use a data placement policy to maximize data access locality) [25].

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 [22, 23]. Second, each application may obtain peak performance at a different configuration point [3, 4, 22, 23]. Third, depending on the context, there are multiple metrics of interest to optimize [11, 22, 25]: time-to-solution, throughput, energy, or, increasingly common in cloud computing environments, the cost of resources.

The Problem. In this scenario, the role of the application administrator/user is non-trivial: if the 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., in terms of application turnaround time, storage footprint, energy consumption, or financial cost). This involves allocating resources and configuring the storage system (e.g., chunk size, stripe width, data placement policy, and replication level). Thus, the decision space revolves around: *provisioning the allocation* - total number of nodes, deciding on node type(s) (or node specification) 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., choosing 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 (see examples in §3).

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 or dollar amount spent). More concretely the user is often interested in answering specific questions: *What is the configuration that can achieve the lowest total cost? How should I partition the allocation among application and storage nodes to achieve the highest performance? Which is the allocation that has 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 non-linear interaction among the possible decisions.

Our long-term goal is to design a configuration exploration framework able to explore the multidimensional configuration space to find the provisioning and the storage system configuration that optimizes a user-specific metric [14]. To reach this goal, we design a prediction mechanism - the focus of this paper - that is able to predict the application performance given a certain resources and storage system configuration.

This paper presents our progress to date on designing and harnessing a performance prediction mechanism for an intermediate object-based storage system in the context of workflow applications. 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. This approach can support *autotuning*: a software tool that relies on the proposed mechanism can enable efficiently configuring the storage system [14, 22, 23], through exploring the configuration space without actually running the application.

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

- Synthesizes the key requirements for a prediction mechanism (§2.1).
- Describes a prediction mechanism that relies on a uniform queue-based model for distributed, object-based storage systems (§2.3). More important, it proposes a system identification procedure to seed the model that is simple, lightweight, effective, and does not require storage system or kernel changes to collect monitoring information (§2.4).
- Evaluates the prediction mechanism for workflow applications and synthetic benchmarks in the context of making configuration choices for different scenarios (§3). The evaluation shows that the predictor is lightweight (up to 2000× less resources (machines × time) than running the actual applications) and effective (identify the configuration that achieves the best performance in the experiments).
- Finally, this paper discusses our experience and lessons learnt (§5) from using the prediction mechanism as a performance testing and debugging tool for distributed storage system development.

2. THE DESIGN OF A PERFORMANCE ESTIMATION MECHANISM

Making accurate performance predictions for distributed systems is a challenge. Since in most cases purely analytical models cannot provide adequate accuracy, simulation is the commonly adopted solution. At the one end of the design spectrum, current practice (e.g., NS2 simulator [2]) suggests that while simulating a system at low granularity (e.g., packet-level simulation in NS2) can provide high accuracy, the complexity of the model, the complexity of the seeding process, and the number of events generated make accurately simulating large-scale systems infeasible, and reduces the generality of the model. Further, the improvement in accuracy may not add much value. At the other end, coarse grained simulations (e.g., PeerSim [20]) scale at the cost of lower accuracy.

Two key observations enable us to reduce simulation complexity and increase its scalability: First, as the goal is to support configuration choice for a specific workload, achieving perfect accuracy is less critical as long as the configuration decisiodummy.psns 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 (e.g., the chunk transfer time is dominated by the time to send the data, not accounting the time of the acknowledgments and all metadata messages will not tangibly impact accuracy).

Our solution uses a queue-based storage system model for the system components' operations and their interactions. The model requires three inputs from the user: the storage system configuration, a workload description, and the performance characteristics of storage system components (i.e., system identification §2.4). The simulator instantiates the storage system model with the specific component characteristics and configuration, and simulates the application run as described by the workload description.

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

2.1 Solution Requirements

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

- **Accuracy.** The mechanism should provide *adequate accuracy*. Although higher accuracy is always desirable, in the face of practical limitations to achieve perfect accuracy, 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.
- **Scalability and Response Time.** The predictor should enable quick exploration of the configuration space. The

mechanism should offer performance predictions quickly and scale with: (i) the system size; and (ii) the I/O intensity of I/O workflow applications.

- **Usability and Generality.** The predictor should not impose a burdensome effort to be used. Specifically, the bootstrapping/seeding process should be simple and it should not require storage system redesign (or a particular initial design) to collect performance measurements. Additionally, using the predictor should not require in-depth knowledge of storage system protocols and architecture.

2.2 Object-based Storage System Design

We focus on a widely-adopted object-based storage system architecture (e.g., UrsaMinor [3], PVFS [15], and MosaStore [4]). 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. Client SAIs implement data access protocols.

Data placement. The default data placement generally adopted 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. Additionally, and key for workflows, application driven data placement policies that optimize for a specific application access patterns have seen increasing adoption [25,30]. For instance, the following data placement policies are used to optimize for the workflow applications' data access patterns: local, co-locate and broadcast (detailed in §3).

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

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 [3,4,25]. 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 (Figure 1): each machine hosts a network component and can host one or more system components (each modeled as a service with its own queue).

A system component and its 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, at the high-level the storage system protocol by sending control or data requests to other services, and once a storage operation finished it communicates again with the application. Each of these components is modeled as service that takes requests from its queue (fed by the network service or by the application for the client service) and sends responses back through the network service.

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, 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.

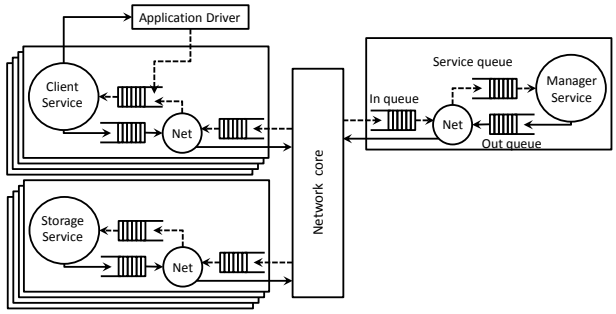


Figure 1: Queue-based model of a distributed storage system. Each component (manager, client component, and storage component) has a single system service that processes requests from its queue. Additionally, each host has a network component with an in- and out- queue. The network core connects and routes the messages between the different components in the system and can model network latency and contention at the aggregate network fabric level. Solid lines show the flow going out from a storage system component while dashed lines show the in-flow path.

The system components can be collocated on the same host (e.g., the client and storage components on the same host). Requests between these collocated services also go through the network, but have a faster service time than remote requests - representing a loopback data transfer (§2.4).

Space limitations prevent us from presenting the full details of the model. A technical report [12] presents more details on the prediction mechanism. As a rule, we accurately model the data paths of the storage system at chunk-level granularity, and the control paths at a coarser granularity: modeling only one control message to initiate a specific storage function while an implementation may have multiple rounds of control messages.

2.4 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 (μ^{net}) and the system components (storage - μ^{sm}, manager - μ^{ma}, and client - μ^{cli}).

Compared to past work (e.g., [23]), our approach focuses on making this *process simple, and not intrusive as no changes are required to the storage system or kernel modules.* The system identification process is automated with a script as follows. First, to measure the service time per chunk/request T^{net}), a script runs a network throughput measurement utility tool (e.g., iperf), to measure the throughput of both: remote and local (loopback) data transfers. Second, this 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. The script computes the average read/write time T^{tot}. The number of files read/written is set to achieve 95% confidence intervals with ±5% error.

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}) is based on a simple analytical model based on network throughput and proportional to the amount of data to be transferred in a packet.

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 to the manager. While 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 chunk size. Therefore, $\mu^{sm} = \frac{T^{sm}}{chunkSize}$.

2.5 Model Implementation: The Simulator

We have implemented the above model as a discrete-event simulator in Java. The simulator receives as inputs: a summarized description of the application workload (§2.6), the system configuration (currently, it supports replication level, stripe-width, and data-placement per file; and chunk size system-wide), the deployment parameters (number of storage nodes and clients, whether they are collocated on the same hosts), and a performance characterization of system components: service times for network, client, storage, and manager (§2.4).

Once the simulator instantiates the storage system, it starts the application driver that processes the application workload. The driver reads the description of the application workload, creates the corresponding events (e.g., read from file x at offset y, z bytes) and places them in the client service queue. File-specific configuration [3, 25] is described as part of the operations in the workload description.

As in a real system, the manager component maintains the metadata of the system (i.e., implements data placement policies, and keeps track of file to chunk mapping and chunk placement). To make the process clearer, consider the following example for a file write operation. First, the 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 to the manager the chunk-map. Once the manager acknowledges, the client returns success to the application driver. In total the write operation generates two requests to the manager and one request per chunk to the storage nodes.

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 is part of the workload description.

2.6 Workload Description

The predictor takes as input a description of the workload. This description contains two pieces of information: per client I/O operations trace (i.e., open, read, write, close calls with timestamp, operation type, size, offset, and client id), and files' dependency graph (capturing workflow execution plan) for scheduling and data placement purposes. The client traces are obtained by running and logging the application operations. The execution plan can be provided by the workflow scheduler (e.g., Swift [27]), by an expert user or extracted from log files. Currently, we use the workflow execution plan from PyFlow scheduler and client traces from MosaStore storage logs, which required no further modification for this work. We have developed a FUSE wrapper to log the storage operations, if the storage system does not provide the needed information. The predictor preprocess the logs to infer the operations' elapsed times and interarrival times based on timestamps, aggregate some operations, and create the events to be simulated.

3. EVALUATION

This section aims to evaluate the mechanism's prediction accuracy and, more important, to demonstrate through a set of experiments the mechanism's ability to support correctly identifying quasi-optimal configuration for a specific application. To this end, we use a set of synthetic benchmarks and real applications. The synthetic benchmarks are designed to mimic the access patterns [25] of workflow applications.

To understand how the prediction mechanisms can be used in a real set-up, we use two real workflow applications: BLAST [5] and Montage [16]. The goal is to evaluate the mechanism's ability to predict time-to-solution to support decisions on the storage configuration and allocation.

Storage system. We use an open source distributed object based storage system [4]. We choose to experiment with RAMDisks as they are frequently used to support workflow applications as intermediate storage: it offers higher performance and are the only option in most supercomputers that do not have spinning disks (e.g., IBM BG/P machines).

Testbeds. We use two testbeds. The first testbed (TB20) is our lab cluster with 20 machines. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4GB RAM, and 1Gbps NIC. The second testbed (TB100), used for larger scale experiments, includes 100 nodes on Grid5000 'Nancy' cluster [10]. Each machine has Intel Xeon X3440 4-core, 2.53-GHz CPU, 16GB RAM, 1Gbps NIC, and 320GB 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. The networks is shared with applications running on different machines. The simulator is seeded according to the procedure described in §2.4.

3.1 Synthetic Benchmarks: Workflow Patterns

This section evaluates the accuracy of the prediction mechanism in capturing the system behavior with multiple clients,

multiple applications, and different data-placement policies designed to support workflow applications [25]. We use synthetic benchmarks that mimic common data access patterns of workflow applications: pipeline, reduce, and broadcast (Figure 2). These are the most popular patterns uncovered by studying over 20 scientific workflow applications by Wozniak et al. [28], Shibata et al. [21], and Bharathi et al. [9]).

The *synthetic benchmarks are designed to explore the limitations of the predictor* as they are composed exclusively of I/O operations, which generates high network and disk contention in the system.

Summary of results. The predictor has good accuracy: our approach leads to prediction errors of 5% on average, lower than 8% in 80% of the studied scenarios, and within 14% 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 storage system setup as described above on the TB20 testbed. 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, 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) [25]. All WASS experiments assume 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), most 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 Figure 2), the *small* (10x smaller than medium), and where possible, a 10x larger, *large* workload. We omit results for a *small* workload, since it exhibits a similar performance 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 (which was enough to guarantee a 95% confidence level).

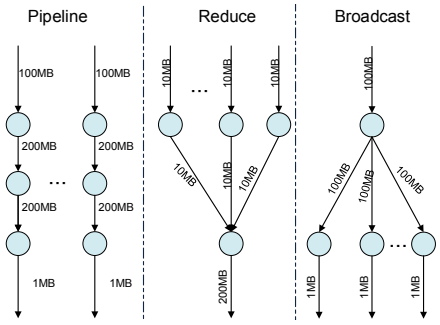


Figure 2: Pipeline, Reduce, and Broadcast benchmarks. Nodes represent workflow stages and arrows represent data transfers through files. The file sizes represent the *medium* workload. The part of the flow that is repeated, ran over 19 machines in this evaluation.

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 (Figure 2). A pipeline-optimized storage system will store the intermediate pipeline 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 data access locality. Here, 19 application pipelines run in parallel and go through three processing stages that read and write files from/to the intermediate storage. (The large workload produces too much data to fit in the in-memory intermediate storage system in the TB20 testbed.)

Evaluation of the results. For the optimized configuration (WASS) the predictor has almost perfect accuracy (Figure 3). For the default data placement policy (DSS), however, predictions are 9% 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 components in the system leading to contention and chunk transfer retries due to connection initiation timeouts caused by network congestion which, we believe, are the main source of prediction inaccuracies.

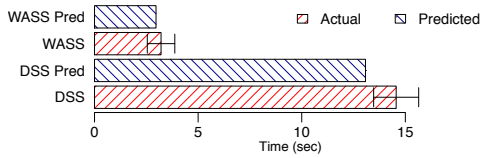
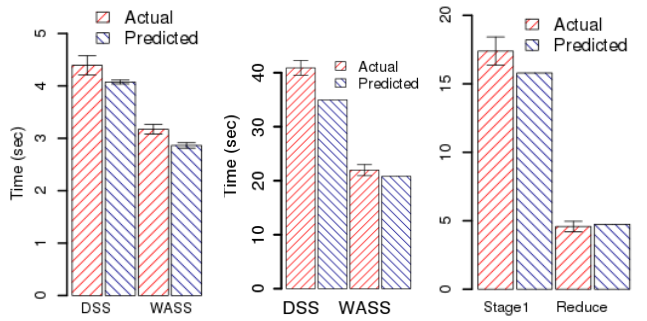


Figure 3: Actual and predicted average execution time for the pipeline benchmark and medium workload.

Reduce benchmark. A single compute task uses input files produced by multiple tasks. In the benchmark, 19 processes run in parallel on different nodes, consume an input file, and produce an intermediate file. In the next stage of the workflow, a single process reads all intermediate files and produces the final output file. A possible data placement optimization is the use of collocation - placing input files on one node and expose their location, which will later be used by the scheduler to run the reduce task on that machine. For WASS configuration, this 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% of the actual average for medium workload, and 13% of the actual performance for large (Figure 4). More important, they capture the relative improvements the pattern-specific data placement policies bring. We note that Figure 4(b) 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.

When the collocation and locality optimizations are not enabled, the challenge of capturing exactly the system behavior is similar to the pipeline case: capture the complex interactions among all machines in the system. When the specific data placement is enabled though, the challenge is different: there is a high contention created by having several clients writing to the same storage machine (the one that



(a) Medium (b) Large (c) Large per Stage for WASS

Figure 4: Actual and predicted average execution time for the reduce benchmark for the medium, large workloads, and per stage for large workload.

performs the reduce phase). Figure 4(c) shows the results per-stage for the two stages of the large workload separately to show how the predictor captures these cases.

Broadcast benchmark. A single task produces a file that is consumed by multiple tasks. In this benchmark, 19 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 that will be consumed by several different tasks.

Evaluation of the results. The results show for broadcast pattern with medium workload with the WASS system have similar performance of aproximately 3 seconds when configured with 1, 2, or 4 replicas (the large workload shows a similar trend) [12]. For this benchmark all predictions match the actual results: predictions are inside the interval of mean of actual ± standard deviation, just 1-3% difference from the mean.

This experiment highlights an interesting case for the predictor. According to the structure of the pattern and the results reported in [25], creating replicas will improve the performance of the broadcast pattern. The results, however, show that creating replicas does not really help here. This happens because data striping already avoids the contention of a single node holding the file. So, although creating replicas reduces the number of accesses to a given machine (since chunks are read in sequence), this gain is not paid off by the overhead of creating a replica. More important, this is another situation where the *predictor captures the impact of different configurations, showing, in this case that they are equivalent and the user can stick with one replica and save storage space.*

3.2 The pipeline benchmark at scale

This section expands the analysis of the synthetic benchmarks to answer the following questions: *"How accurate is the predictor estimates for a different platform?"*, and *"How does the predictor capture the behavior of larger scale systems?"* To answer these question, we ran the pipeline benchmark at scale on our Grid 5000 testbed, TB100. We chose this benchmark because it is the one with the biggest gap between predicted and actual, and it is the most I/O intensive benchmark which stresses the network and the metadata manager, a component well-known for being a potential bottleneck for this type of cluster-based storage system.

We executed this benchmark for a weak scaling set-up using three different scales (25, 50, and 100 nodes), the medium workload, and the two configurations (DSS and WASS). The stripe width is set to 4, reflecting a typical value in distributed storage setup [4, 15].

Figure 5 shows the results. The predictor produces estimates that differ 15% of the actual time on average, are within 22% of the actual results for all cases, and are close to the interval delimited by the standard deviation. Different from the TB20 results, most of the scenarios on TB100 show cases where the predictor underestimate the time, instead of overestimating. We believe it happens because the faster machines of TB100 offer higher parallelism.

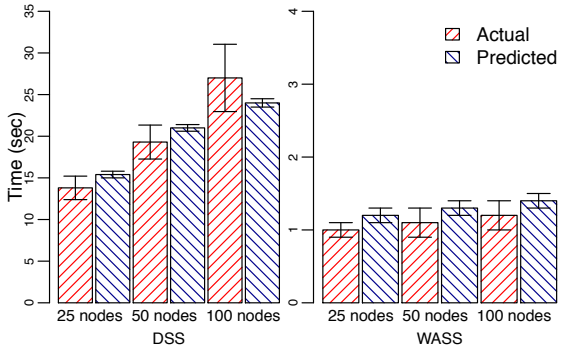


Figure 5: Actual and predicted average execution time for the pipeline benchmark medium workload on TB100.

3.3 Supporting decisions for a real application

Section 3.1 evaluated 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, *the previous section evaluated prediction accuracy when the application and storage system are co-deployed on the same nodes, this section evaluates accuracy when they are deployed on separate nodes.*

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 HPC centers with limited user budget or cloud environments). *Problem: For a fixed workload; what is the cost/turnaround time trade-off space among the deployment options?*

Workload. We explore these two scenarios with a real workflow application: BLAST [5] a DNA search tool for finding similarities between DNA sequences. In the BLAST workflow, each node receives a set of DNA queries as input (a file for each node with 200 search queries) and all nodes search the same DNA database file stored on intermediate storage (total size of 1.67 GB). Each machine produces one output file, and the files are combined at the end of the

application execution. 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 can either execute tasks from the workload or act as storage nodes.

Experimental methodology. The plots report the average of at least 20 runs, leading to 95% confidence intervals for all experiments. Since standard deviation is low (less than 5%), we omit it in plots to reduce clutter.

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 intermediate storage system configuration to yield highest application performance?*

Figure 6 shows the application execution time for different partitioning and storage system configurations. For this application *chunk size* is the configuration parameter that has the highest impact on performance, thus, to limit the number of possible configurations in the figure, we focus on it only. (We note that the predictor correctly captures the lack of impact for other parameters). Additionally, this scenario covers a configuration parameter not evaluated in §3.1.

Figure 6 highlights several important points: First, the performance difference between the different configurations is significant: up to 10x difference between the best and the worst configuration even for the same chunk size. Second, the results show that the system achieves the fastest processing time with a partitioning of 14 application nodes and 5 storage nodes, and chunk size of 256KB (4x 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. Actually, 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 to make, for example, they aim for the best application turnaround within a certain dollar budget. We aim to inform the user's provisioning decisions 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?*

Figure 7 shows, on different Y-axes, the application execution time and allocation cost (measured as number of nodes x allocation time) for different cluster sizes, different partitioning, and different chunk size. Similar to Scenario I, Figure 7 shows that the predictor captures the system performance with reasonable accuracy.

Figure 7 also shows that the an allocation of 11 nodes, with partitioning of 8 application, 2 storage nodes, and chunk size of 256KB offers the lowest cost. However, this figure points out an interesting case for the analysis of cost vs. time-to-solution: The user can analyze the plot to verify that an option with an allocation of 20 nodes actually offers almost 2x higher performance at a marginal 2% higher cost.

3.4 Increasing workflow complexity: Montage

This section aims to answer the following question: *"Can the predictor support user decisions for more complex applications than the one presented in §3.3"*. To answer this question, we focus on evaluating how accurate the estimates are for Montage [16], a complex astronomy workflow composed of 10 different stages (Figure 8) with varying characteristics.

To verify that the predictor can support user's decision, we have executed Montage for different deployments sizes on TB20. For this application, we use clients collocated with storage nodes, verifying yet another configuration parameter. We omit chunk size variations since it does not impact actual Montage (also well captured by the predictor).

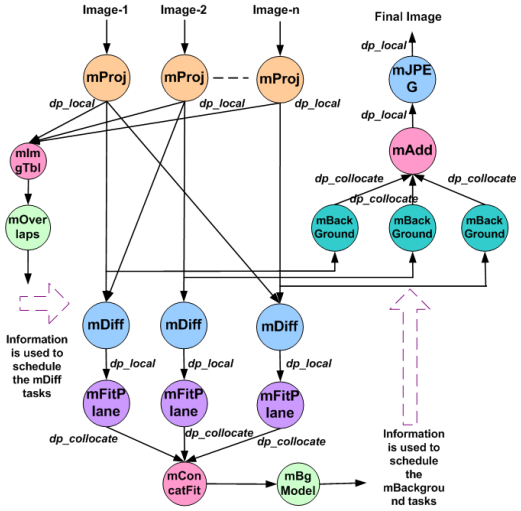


Figure 8: Montage workflow.

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 generates over 650 files with sizes from 1KB to over 100MB and about 3GB of data are read/written from storage.

Stage	Data	#Files	File Size
stageIn	109MB	57	1.7MB-2.1MB
mProject	438MB	113	3.3MB-4.2MB
mImgTbl	17KB	1	
mOverlaps	17KB	1	
mDiff	148MB	285	100KB - 3MB
mFitPlane	576KB	142	4KB
mConcatFit	16KB	1	
mBgModel	2KB	1	
mBackground	438MB	113	3.3MB - 4.2MB
mAdd	330MB	2	165MB
mJPEG	4.7MB	1	
stageOut	170MB	2	4.7MB-165MB

Table 1: Characteristics of Montage workflow stages

Evaluation of the Results. Figure 9 shows the execution time, actual and predicted, for Montage on different cluster sizes. The average summarizes 20 trials. The standard deviation is approximately 3% and omitted to reduce

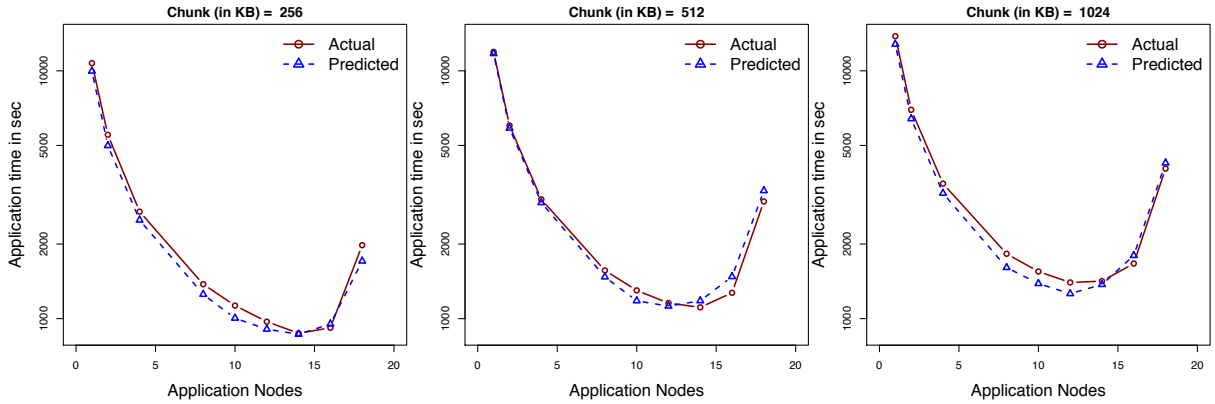


Figure 6: Application runtime (log-scale) for a fixed-size cluster of 20 nodes. X-axis represents number of nodes allocated for the application/storage. Each plot presents results for a different configuration of chunk size.

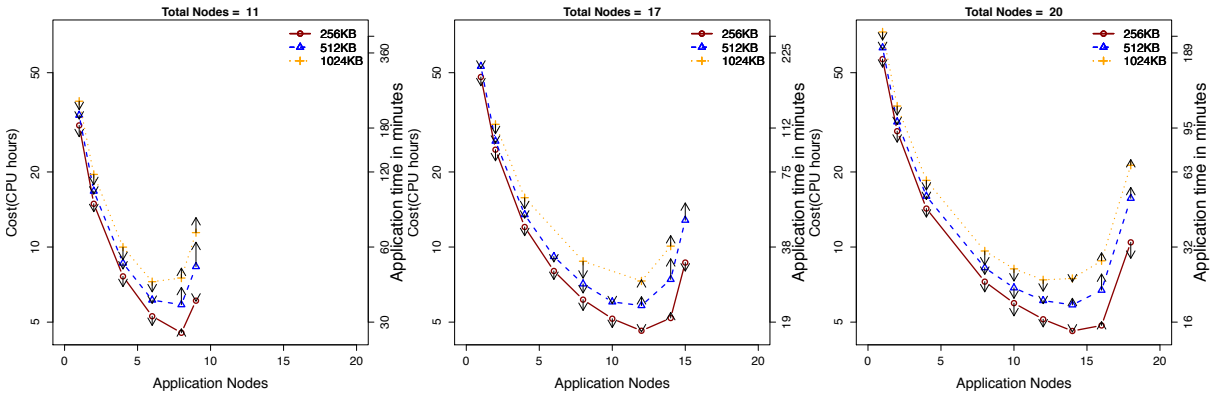


Figure 7: Allocation cost (CPU-hours, log-scale, left Y axis) and application time (right Y axis, log-scale, different scale among plots) for fixed size clusters and different chunk sizes. X-axis represents number of nodes allocated to the application/storage. Lines and arrows present the actual and the predicted cost/performance, respectively.

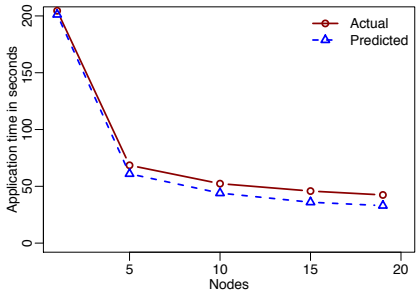


Figure 9: Montage time to solution for fixed size deployments between 1 and 19 nodes - an extra node runs the storage manager on TB20.

clutter. Figure 9 shows that, overall, the predictor captures well the application performance. The increased workflow complexity is a challenge: the predictor is not as accurate as for the BLAST application presented earlier (here the average prediction error is 9%, the smallest is than 1%, and the maximum prediction error is 15%). However, the overall accuracy is "good-enough" to support the provisioning decisions, and the difference is barely visible in the plot.

4. 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 [7] targets centralized storage solution based on one enclosure to recommend an initial system configuration, and Hippodrome [6] 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) [19] 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 oversimplified 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 60%. PFSsim 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++ [26] and disks using DiskSim [1]. Liu et al. [17,18] build a simulation framework for simulating the storage system of supercomputing machines. The framework simulates all hardware components including compute ad IO nodes, storage subsystem, and the supercomputing interconnect. Similar to this work, Thereska et al. [23] proposed a predictor mechanism

for a distributed storage system with a detailed model. To provide such information, they propose Stardust [24] 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% of the actual predictions depending on the workload. Our approach has achieved similar accuracy on 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 approaches targets simulating a generic distributed storage system architecture in a cluster infrastructure (not special supercomputer machines) without simulating particular storage system operations. 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.

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 instead of the average performance for a batch [6, 7] of operations, and of predicting performance of the system from the perspective of one client [23] at a time. Additionally, our work targets the partitioning problem of splitting the nodes among application and intermediate storage.

Monitoring based tuning. Behzad et al. [8] present an auto tuning framework for HDF5 IO library. The proposed solution injects optimization parameters into parallel HDF5 I/O calls. Further, the framework monitors the I/O performance, and explores the tuning parameter space using a genetic algorithm. Zhang et al. [29] propose an approach to determine the storage bottleneck for workflow applications using a set of benchmarks and target workflow application runs.

The approach we propose enables a richer 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 change the configuration without requiring new runs of the application for new generations of the genetic algorithm. Additionally, we target simulating generic POSIX-based storage system not a specific I/O library.

5. SUMMARY AND DISCUSSION

Summary. This paper makes the case for a prediction mechanism to support automating provisioning choices for workflow applications. We focus on predicting the performance of 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.

This paper focuses on predicting the application time-to-solution (turnaround time) of the applications and benchmarks, but we note that the model and approach presented apply to other optimization metrics (e.g., cost, amount of data transferred).

We intend to expand this work in multiple directions: (i) explore a richer space of configuration parameters, (ii) evaluate the system using additional benchmarks and applications, (iii) enable different optimization criteria [22] including energy [11], and (iv) explore different optimization solvers to search the configuration space.

The discussion below aims to clarify our understanding of the limitations of our work and the lessons we have learned.

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 (e.g., support services like garbage collection or storage node heartbeats; the control paths are simplified to match what we believe generic object-based storage would do - while we know that a FUSE-based implementation would need more complex control paths; we model all control messages as having the same size) and the environment (e.g., contention at the network fabric level or scheduling). Second, we constrain and simplify system identification even further at the cost of additional accuracy loss. Third, we do not model the infrastructure in detail (e.g., we do not model the network protocols or the spinning disks). Finally, so far the application driver uses an idealized image of the workflow application (e.g., all pipelines are launched in the simulation exactly at the same time while in the actual experiments coordination overheads make them slightly staggered). We believe the latter one is the main reason of current inaccuracies in the system. In fact, our initial evaluation verified that for Montage the overhead can be up to 5% of the total time.

Can the prediction mechanism support the *development process* of a storage system? Do you have specific experience in this context? One of the lessons we have learned so far is the utility of the mechanism to support the development of the storage system itself. Back of the envelope calculations are a common mechanism to evaluate expected performance bounds for a given system. The predictor takes this a step further and is useful in complex scenarios where back of the envelope estimates are intractable. Not only the developers can use it to evaluate the potential gains of implementing a new complex optimizations or to study the impact of faster network and nodes, but to obtain a performance baseline to detect performance anomalies.

More concretely, we have encountered a number of situations where the predicted and actual performance differed significantly. In some cases these highlighted simplifications in the model or in our simulator. But, there were cases that highlighted complex performance-related anomalies in the storage system such as: non-trivial implementation problems (e.g., limited randomness in the data placement decisions that led to an artificial bottleneck, or unreasonable locking overheads). Similarly, the prediction mechanism helped us revisit assumptions about the middleware stack the storage system is implemented over (e.g., significant impact of the TCP connection initiation timeout of 3s in some scenarios). Finally, the predictions highlighted shortcomings of the seeding process or incorrect assumptions about the deployment platform (e.g., we were ignoring platform heterogeneity). A technical report describes this experience [13].

How general is the proposed prediction mechanism? We have designed the predictor to model a generic

object-based storage system and all the system identification process to work at the application level only (thus, easily portable across storage systems and deployment platforms). While so far we have evaluated the predictor in depth only for the DSS/WASS storage system we have encouraging preliminary experience with using it to predict the relative application performance on two other storage solutions, Ceph and GlusterFS, for a limited number of scenarios.

Acknowledgements

This research was supported in part by the DoE ASCR XStack program (ER26013) and by the Natural Sciences and Engineering Research Council of Canada.

6. REFERENCES

[1] DiskSim. `http://www.pdl.cmu.edu/DiskSim/`.

[2] The network simulator NS2. `http://www.isi.edu/nsnam/ns/`, 2012.

[3] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proc. of the Conf. on File and Storage Technologies*, Dec. 2005.

[4] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. The case for a versatile storage system. *SIGOPS Oper. Syst. Rev.*, 44:10–14, March 2010.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J. of Molecular Biology*, 215(3):403–410, Oct. 1990.

[6] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proc. of the Conf. on File and Storage Technologies*, pages 175–188, 2002.

[7] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst.*, 23(4):337–374, Nov 2005.

[8] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. A. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Auto-Tuning. In *SC*, 2013.

[9] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. 3rd Workshop on*, pages 1–10, 2008.

[10] F. Cappello, E. Caron, M. Daydé, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Grid Comp.. Proc of the 6th IEEE/ACM Intl. Workshop on*, 2005.

[11] L. B. Costa, S. Al-Kiswany, R. V. Lopes, and M. Ripeanu. Assessing data deduplication trade-offs from an energy and performance perspective. In *2011 Intl. Green Computing Conf. and Workshops*, 2011.

[12] L. B. Costa, A. Barros, S. Al-Kiswany, E. Vairavanathan, and M. Ripeanu. Predicting intermediate storage performance for workflow applications. *CoRR*, abs/1302.4760, 2013.

[13] L. B. Costa, J. Brunet, L. Hattori, and M. Ripeanu. Experience on Applying Performance Prediction during Development: a Distributed Storage System Tale. Technical report, UBC/ECE/NetSysLab, Sep. 13. `http://www.ece.ubc.ca/~lauroc/tr/tech2.pdf`.

[14] L. B. Costa and M. Ripeanu. Towards Automating the Configuration of a Distributed Storage System. In *11th ACM/IEEE Intl. Conf. on Grid Computing - Grid 2010*, Oct. 2010.

[15] I. F. Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux Journal*, 2000(80es), Nov. 2000.

[16] A. C. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, J. C. Jacob, D. S. Katz, and T. Prince. Montage: An Astronomical Image Mosaic Service for the NVO. In *Astronomical Data Analysis Software and Systems XIV*, volume 347 of *Astronomical Society of the Pacific Conf. Series*, page 34, Dec 2005.

[17] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn. Modeling a leadership scale storage system. In *Proc. of the 9th Intl. Conf. on Parallel Processing and Applied Mathematics - Vol. Part I*, PPAM'11, pages 10–19, 2012.

[18] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symp. on*, pages 1–11, 2012.

[19] E. Molina-Estolano, C. Maltzahn, J. Bent, and S. Brandt. Building a Parallel File System Simulator. 180(1):012050, 2009.

[20] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conf. on Peer-to-Peer (P2P'09)*, pages 99–100, Sep 2009.

[21] T. Shibata, S. Choi, and K. Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In *Proc. of the 19th ACM Intl. Symp. on High Perf. Distributed Computing*, HPDC '10, pages 746–755, 2010.

[22] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *6th USENIX Conf. on File and Storage Technologies, FAST*, pages 313–328, 2008.

[23] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *Proc. of the 3rd Intl. Conf. on Autonomic Computing*, pages 187–198, 2006.

[24] E. Thereska, B. Salmon, J. D. Strunk, M. Wachs, M. Abd-El-Malek, J. López, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS/Perf.*, pages 3–14, 2006.

[25] E. Vairavanathan, S. Al-Kiswany, L. B. Costa, Z. Zhang, D. S. Katz, M. Wilde, and M. Ripeanu. A workflow-aware storage system: An opportunity study. In *Cluster Computing and the Grid, IEEE Intl. Symp. on*, pages 326–334, 2012.

[26] A. Varga. Using the OMNeT++ Discrete Event Simulation System in Education. *Education, IEEE Trans. on*, 42(4), 1999.

[27] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. T. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[28] J. M. Wozniak and M. Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proc. of the 4th Workshop on Petascale Data Storage*, PDSW '09, pages 16–20, 2009.

[29] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. Foster. MTC Envelope: Defining the Capability of Large Scale Computers in the Context of Parallel Scripting Applications. In *Proc. of the 22Nd Intl. Symp. on High Perf. Parallel and Distributed Computing*, HPDC '13, pages 37–48, 2013.

[30] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 85:1–85:11, 2012.