Predicting Intermediate Storage Performance for Workflow Applications

Lauro Beltrão Costa, Samer Al-Kiswany, Abmar Barros*, Hao Yang, Matei Ripeanu
University of British Columbia, *Universidade Federal de Campina Grande

ABSTRACT

System configuration decisions for I/O-intensive workflow applications can be complex even for expert users. Users face decisions to configure several parameters optimally (e.g., replication level, chunk size, number of storage node) - each having an impact on overall application performance. This paper presents our progress on addressing the problem of supporting storage system configuration decisions for workflow applications. Our approach accelerates the exploration of the configuration space based on a low-cost performance predictor that estimates turn-around time of a workflow application in a given setup. Our evaluation shows that the predictor is effective in identifying the desired system configuration, and it is lightweight using 2000-5000× less resources (machines × time) than running the actual benchmarks.

1. INTRODUCTION

Assembling workflow applications by putting together standalone binaries has become a popular approach to support large-scale science [8, 15, 20]. The processes spawned from these binaries communicate via temporary files stored on a shared storage system. In this setup, a workflow runtime engine is basically a scheduler that builds and manages a task-dependency graph based on the tasks' input/output files (e.g., SWIFT [19]).

To avoid accessing the platform's backend storage system (e.g., NFS or GPFS or Amazon S3), recent proposals [3, 20] advocate using some of the nodes allocated to the application to deploy an *intermediate storage system*: a shared temporary in-memory storage system dedicated to (and co-deployed with) the application.

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) [18].

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 [16,17]. Second, each application may obtain peak performance at a different configuration point [2,3,9,16,17]. Third, depending on the context, there are multiple metrics of interest to optimize [10,16,18]: time-to-solution, throughput, energy, or, increasingly common in cloud computing environments, the cost of resources.

In this scenario, the role of the application administrator/user becomes non-trivial: in addition to running the workflow application, the user has to configure the intermediate-storage system to achieve high performance. This involves allocating resources (e.g., how many nodes) and configuring the storage system (e.g., data placement policy, and chunk size). *Manually* tuning the storage system configuration and allocation decisions is hard, and time-consuming due to the large configuration space to consider and non-linear interaction among the possible decisions.

Our *goal is to design a mechanism that enables the user to explore the multidimensional configuration space of the intermediate storage system.* To this end, we propose a prediction mechanism, the focus of this paper, that is able to predict the storage system impact on the application performance.

This paper summarizes 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 characterization, and the deployment platform characteristics, the mechanism predicts the application turnaround time. This approach can also support *autotuning* at runtime: a software tool can rely on the proposed mechanism to efficiently configure the storage system [12,16,17], by exploring the configuration space without actually running the application (as it is time-consuming).

The contributions of this paper is twofold: First, it synthesizes the key requirements for a prediction mechanism (§2.1). Second, it presents a prediction mechanism based on a simple queue-based model for distributed object-based storage system (§2). An important feature of the proposed model is that it relies on a simple, and lightweight system identification procedure to seed the model. Section 3 evaluates the prediction mechanism with synthetic benchmarks and a real application; giving a glimpse of how the application execution time can vary depending on the choices made, and how the predictor can guide the search for the desired configuration.

2. THE DESIGN OF A PERFORMANCE PREDICTOR

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 we target (§2.2). Then, it focuses on the proposed solution: 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).

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 [1]) 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. Further, the improvement in accuracy may not add much value. At the other end, coarse grained simulations [14] tend to scale at the cost of lower accuracy

Two key observations allow us to reduce model 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 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 complexity by not modeling 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 acknowledgements and all metadata messages do not tangibly impact accuracy).

2.1 Solution Requirements

A practical prediction mechanism should meet the following, partially conflicting, requirements:

- **Accuracy.** The mechanism should provide *adequate accuracy.* Although higher accuracy is always desirable, practical limitations to achieve perfect accuracy result in decreasing incremental gains for improved accuracy. For example, to support configuration decisions, a predictor only needs to correctly estimate relative performance (between different configuration options) or trends [17] to allow choosing the desired configuration.
- **Scalability and Response Time.** The predictor should enable quick exploration of the configuration space. The mechanism should offer performance predictions quickly and scale across (i) the system size; and (ii) and I/O intensity of 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 changes to the storage system design to collect performance measurements.

2.2 Object-based Storage System Design

We focus on a widely-adopted object-based storage system architecture (e.g., MosaStore [3], PVFS [13], and UrsaMinor [2]). 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, data placement policies that optimize for a specific application access patterns have seen higher adoption [18, 21]. We detail some popular policies in §3.

Replication, for reliability or improving performance. However, while a higher replication level reduces contention on the node storing a popular file, it increases the file write time and the space consumption.

We explore the accuracy of the prediction mechanism assuming that the number of clients, number of storage nodes, chunk size, cache size, stripe width, replication level, and data placement policy are configurable or can change as suggested in [2, 3, 18].

2.3 System Model

The predictor uses a queue-based model for the system components' operations and their interactions. The model requires the following 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 (including task scheduling) as described by the workload description.

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 implements, at a high-level, the storage system protocol by sending control or data requests to other services, and serves the application's read and write operations. Each of these components is modeled as a 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 at destination host processes all the frames of a given request in its in-queue, it assembles the request and places it in the queue of the corresponding service.

Space limitations prevent us from presenting the full details of the model. A technical report [7] presents more details. 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, and the system components (storage module, manager, and client).

Different from past work (e.g., [17]), our approach focuses on making this process simple and not intrusive (no changes to storage system or to kernel modules). The system identification process is automated with a script as follows. First, to

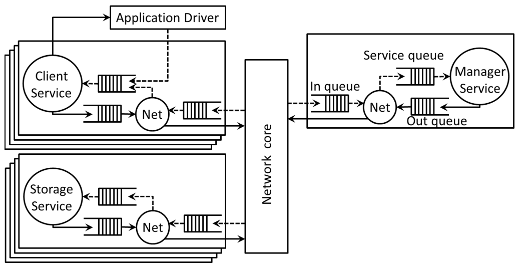

Figure 1: Queue-based model of a distributed storage system. Each component (manager, client component, and storage component) has a single system service that processes requests from its queue. Additionally, each host has a network component with an in- and out- queue. The network core connects and 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.

measure the service time per chunk/request, 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, it 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.

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 time for the network is based on a simple analytical model based on network throughput from utility tool and proportional to the amount of data to be transferred in a packet.

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 (replication level, stripe-width, cache size, 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 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 is described as part of the operations in the workload description [2, 18].

As in a real system, the manager component maintains the metadata of the system (e.g., data placement policies, and keeps track of file to chunk mapping). 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 per-file configuration.

2.6 Workload Characterization

The simulator takes as input a characterization of the workload. This characterization contains two parts: *per workflow task* I/O operations trace (i.e., open, read, write, close calls with timestamp, operation type, size, and offset), and task/files' dependency graph (capturing the workflow execution plan). The traces can be obtained by running and profiling the application. The storage system logs already provide these traces. The execution plan can be provided by the workflow scheduler (e.g., Swift [19]), by an expert user or extracted from log files. Currently, we use workflow execution plan from PyFlow scheduler, and client traces from storage logs. The application characterization used for current prediction is based on one execution only.

3. EVALUATION

This section evaluates the mechanism's prediction accuracy and demonstrates through a set of experiments the mechanism's ability to support correctly identifying a quasi-optimal configuration for a specific application. To this end, we use of synthetic benchmarks and a real application.

Deployment platform. We use MosaStore storage system [3, 18] on a testbed of 20 machines each with an Intel Xeon E5345 2.33GHz CPU, 4GB RAM, and 1Gbps NIC. A machine runs the MosaStore manager; the other 19 machines each runs both a storage node and a client access module.

3.1 Synthetic Benchmarks for Workflow Patterns

This section evaluates the accuracy of the prediction mechanism using synthetic benchmarks designed to mimic real workflow application access patterns(Figure 2) running on a real cluster deployment: multiple clients, multiple storage nodes, and different data-placement policies designed to optimize workflow applications [18]. The goal is to evaluate the mechanism's ability under system interactions that resemble the workflow application ecosystem. To this end, this section focuses on pipeline, reduce, and broadcast patterns (Figure 2). These are among the most used patterns uncovered by studying over 20 scientific workflow applications by Wozniak et al. [20], Shibata et al. [15], and Bharathi et al. [8]).

We note that, these benchmarks are designed to explore the limitations of the predictor and be a worst case in terms of accuracy , and are harder to predict accurately than real applications as they are composed exclusively of I/O operations, which leads to contention in the real storage system.

Experimental setup. We use the MosaStore setup described above. *DSS* label (Default Storage System) indicates experiments for MosaStore default configuration: client stripes data over all 19 machines, and no data-access pattern optimization is enabled. *WASS* label (Workflow Aware Storage System) indicates an optimization for a specific access pattern [18]. All WASS experiments use data location aware scheduling: for a given compute task, if all input file chunks exist on a single storage node, the task is scheduled on that node to increase 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 in a variety of data placement policies, and (ii), most important, show that the prediction correctly indicates the desired configuration. To understand the impact of the data size, for each benchmark, we use

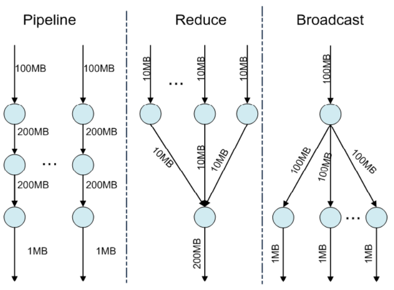

Figure 2: Pipeline, Reduce, and Broadcast benchmarks. Nodes represent workflow stages and arrows represent data transfers through files. The file sizes represent the *medium* workload. The part of the flow that is repeated, ran over 19 machines in this e

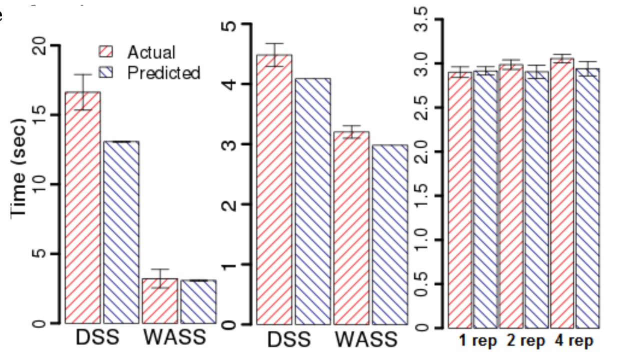

Figure 3: Actual and predicted performance for the pipeline, reduce, and broadcast (1, 2 , and 4 replicas) benchmarks for the medium workload. The plots consider the average turnaround time and for 15 trials, proving a 95% confidence level with maximum error of 5%.

three workloads labeled as *medium* (data sizes indicated in Figure 2), the *small* (10x smaller than medium), and a 10x larger, *large* workload. Results for the *small* workload exhibit similar performance among different configurations and the predictions are inside the confidence interval, thus, we do not discuss them further. For the *large* workload, which overall results are similar to medium, can be found in a technical report [7] with a longer analysis of the results in this section.

Pipeline pattern. A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task (Figure 2 - left). WASS configuration places intermediate pipeline files locally such that 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.

Evaluation results. The predictions match the actual performance for WASS (Figure 3). For DSS, the prediction is 15% far from the average ± standard error. Note that for a case with default data placement policy, all clients stripe (write) data to all machines in the system; similarly, all machines read from all others. This creates, a complex interaction among all components in the system and some retries due to connection timeouts caused by network congestion which, we believe, is the main source of the prediction inaccuracy.

Reduce or Gather pattern. A single compute task uses input files produced by multiple tasks. A data placement optimization is collocation - placing input files on one node and expose their location, which is later used by the scheduler to run the reduce task on that machine. In the benchmark, 19 processes run in parallel on different nodes, consume an input file, and produce an intermediate file each. The next stage consists of a single process that reads all intermediate files, and produces the reduce-file. WASS configuration uses collocation optimization for the files used in the reduce stage, and the locality optimization for the remaining files.

Evaluation results. Figure 3 (middle) shows that the predictions for the reduce benchmark are within 10% of the average ± standard error. More important, they capture the relative improvements that pattern-specific data placement policies bring.

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 (one performing the reduce phase).

Broadcast pattern. 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 an 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 results. Broadcast predictions were inside or very close to the interval of mean of actual ± confidence interval (Figure 3 - right). This experiment highlights an interesting case for the predictor. According to the structure of the pattern and the results reported in [18], creating replicas would improve the performance of the broadcast pattern. The results, however, show that creating replicas does not really help. Although replication can alleviate concurrent access to a given machine, this gain is not paid off by the overhead of creating replicas. More importantly, 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.

Response time. Workflow benchmarks predicted the results in tens of milliseconds, using 2000-5000× less resources (machines × time) than running the actual benchmark, it can also scale while sustaining or improving this ratio [7].

3.2 Supporting decisions for a real application

This section targets a more complex scenario where the user has to deal with a real application, allocation decisions, and the storage system configuration. Further, this section demonstrates prediction accuracy when the number of clients and storage nodes change, and storage uses separate nodes (sometimes a needed approach for in-memory intermediate storage when the storage puts pressure on the application memory or there is not enough memory for both storage and application as in some supercomputers).

To this end, this section shows the predictor's ability to guide the user or a search algorithm to the desired configuration, specifically focusing on the following scenario: given a fixed size cluster, *how should the nodes be partitioned between the application and the intermediate storage, and what should be the intermediate storage system configuration to yield best application performance?*

Workload. We explore this scenario with a real workflow

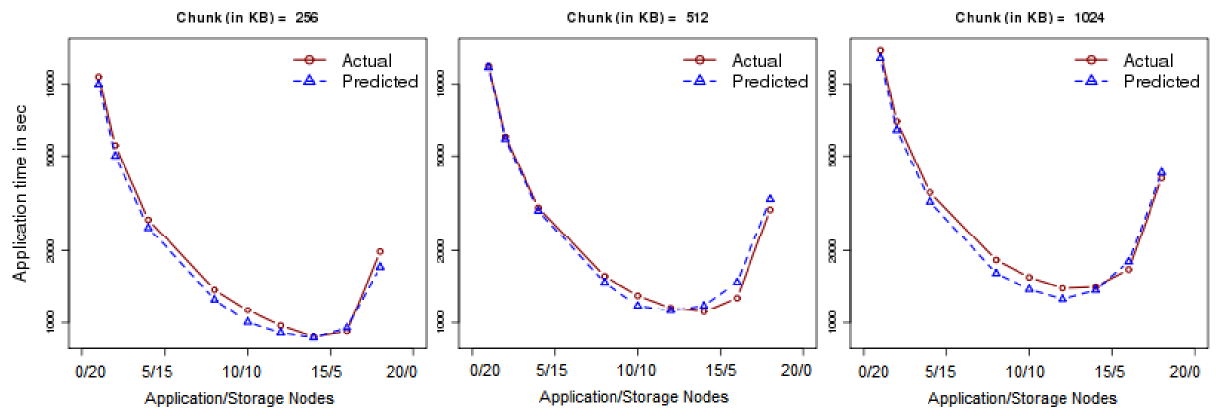

Figure 4: Application runtime (log-scale) for a fixed-size cluster of 20 nodes. One node coordinates BLAST tasks' execution and runs the storage system manager. The remaining nodes can either execute tasks from the workload or act as storage nodes X-axis represents number of nodes allocated for the application/storage. The three plots represent runtime for different storage configurations (chunk sizes). The plots report the average of at least 20 runs, obtaining 95% confidence intervals - since they are small (less than 5%), they are omitted to reduce clutter.

application: BLAST [4] a DNA search tool for finding similarities between DNA sequences. In the BLAST workflow, each node receives a set of DNA sequences as input (a file for each node) and searches the same database file. The workload includes 200 queries using the RefSeq database [4]. Each machine produces one output file, which is combined in the end of the application (reduce pattern).

Evaluation results. Figure 4 shows the actual and predicted application execution time with different partitioning and storage system configurations. For this application, chunk size is the configuration parameter (not covered in §3.1) that has the highest impact on the BLAST workflow. To limit the number of possible configurations in the figure, we focus on it - *predictions capture the lack of impact of the other parameters.*

Figure 4 highlights several important points. First, the 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 a chunk size of 256KB (4x smaller than the default) a non-obvious configuration beforehand. Third, the experiment shows that the predictor accurately captures the system performance given changes in the allocated partition, and on the storage system configuration. Actually, the overall error of the predictions are small (up to 10% of the average and 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.

The technical report [7] presents a second scenario that explores the provisioning problem with cost constraints.

4. RELATED WORK

Past work used model-driven analysis or prediction to estimate the storage system performance. For instance, Ergastulum [6] targets centralized storage solution to recommend an initial system configuration, and Hippodrome [5] relies on Ergastulum to improve the configuration based on online monitoring. Similar to this work, Thereska et al. [17] proposed a prediction mechanism for a distributed storage system, but for a different class of applications and with a detailed model. A detailed monitoring system provides such information by the cost of changes to the storage system and kernel modules to add monitoring points. This approach enabled their predictor to achieve prediction within 15% of the actual performance depending on the workload. Our approach have achieved similar accuracy on our target workload, however with a simpler model and lightweight approach to seed the model.

Our work is different from the previous effort in three ways: First, we target distributed storage system, a harder to predict system than centralized system due to the interaction among the system components and the more configuration options. Second, we rely on a lightweight seeding process that does not require changes to the system or kernel modules. Finally, we target the problem in the context of workflow applications, where we also consider task dependency and scheduling, and, in addition to system configuration, resource partitioning (between application and storage).

5. SUMMARY AND DISCUSSION

This paper makes the case for a performance prediction mechanism to support automating configuration choices for workflow applications when running on top of an intermediate object-based storage system. Our solution has 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 accuracy that allowed proper decisions for the cases we study.

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 functions [16] - including energy models [10], and (iv) explore different optimization solvers to search the configuration space.

A technical report [7] presents a longer analysis and discussion to clarify our understanding of the limitations of this work and the lessons we have learned. We discuss how a performance predictor can support the *development process* of a storage system [11] in another report.

6. REFERENCES

[1] The Network Simulator NS2. http://www.isi.edu/nsnam/ns/, 2012.

[2] 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 *Proceedings of the Conference on File and Storage Technologies*, December 2005.

[3] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. The Case for Versatile Storage System. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2009.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

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

[6] 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.

[7] L. B. ao Costa, A. Barros, S. Al-Kiswany, E. Vairavanathan, and M. Ripeanu. Predicting intermediate storage performance for workflow applications. Technical report, UBC/ECE/NetSysLab, September 2013. http://www.ece.ubc.ca/~lauroc/tr/tech.pdf.

[8] 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. Third Workshop on*, pages 1–10, 2008.

[9] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *J. Parallel Distrib. Comput.*, 62(10):1571–1601, 2002.

[10] 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 International Green Computing Conference and Workshops*, pages 1–6, 2011.

[11] 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, September 2013. http://www.ece.ubc.ca/~lauroc/tr/tech2.pdf.

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

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

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

[15] T. Shibata, S. Choi, and K. Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 746–755, New York, NY, USA, 2010. ACM.

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

[17] 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 *Proceedings of the 3rd International Conference on Autonomic Computing*, pages 187–198, 2006.

[18] 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 International Symposium on*, pages 326–334, 2012.

[19] 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.

[20] J. M. Wozniak and M. Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 16–20, New York, NY, USA, 2009. ACM.

[21] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.