# Towards Automating the Configuration of a Distributed Storage System

Lauro B. Costa, Matei Ripeanu
Department of Electrical and Computer Engineering
The University of British Columbia
Vancouver, BC, Canada
{lauroc,matei}@ece.ubc.ca

*Abstract*—**Versatile storage systems aim to maximize storage resource utilization by supporting the ability to 'morph' the storage system to best match the application's demands. To this end, versatile storage systems significantly extend the deployment- or run-time configurability of the storage system. This flexibility, however, introduces a new problem: a much larger, and potentially dynamic, configuration space makes manually configuring the storage system an undesirable if not unfeasible task.**

**This paper presents our initial progress towards answering the question: "*How can we configure a distributed storage system (i.e., enable/disable its various optimizations and configure their parameters) with minimal human intervention?*" We discuss why manually configuring the storage system is undesirable; present the success criteria for an automated configuration solution; propose a generic architecture that supports automated configuration; and, finally, instantiate this architecture into a first prototype, which controls the configuration of similarity detection optimizations in the MosaStore distributed storage system. Our evaluation results demonstrate that the prototype can provide performance close to the optimal configuration at the cost of minimal overhead.**

***Keywords: distributed storage systems; automated configuration; self-tuning; data deduplication***

## I. Introduction

Aggregating available storage space from network-connected nodes has several appealing properties: *low cost* – it is cheaper than a dedicated storage solution; *high efficiency* – it allows for good resource utilization; *high-performance* - applications benefit form a wider I/O channel by striping and/or replicating data across several nodes. One of the many instances [1][2][3][5][6][15][16] where this technique is used, yet relevant to Grid settings, is storage system 'glide-ins' [1][18][19][20]. In this scenario, the components of a storage system are submitted together with a batch application and the storage system is instantiated on the fly to aggregate the storage resources available on the nodes allocated to the application. Once instantiated, the storage system will provide a dedicated, high performance 'scratch' space co-located with the application. Multiple projects have validated the practical appeal of this approach [1][3][18][20].

Instantiating the storage system on the fly, however, raises a challenge: optimally designing and configuring the storage system is significantly more complex. For example, *replication* and *caching* may speed data access; yet, they may entail complex consistency protocols. *Online data deduplication*, i.e., data compression by detecting repeated chunks of data and storing them only once, may save storage space and bandwidth when there is high similarity between successive write operations, yet at the cost of increased computational overheads. Typically, to avoid such complexity, most storage system designs fix these decisions in order to make the system simpler to manage, providing, in effect, a *"one size fits all"* solution.

An alternative rarely explored to date is a *versatile storage system* approach [1][2]. Versatility in this case is the ability to provide a set of storage-system optimizations that can be activated and/or configured at deployment time or even at runtime. Versatility enables maximally harnessing the available storage resources by 'morphing' the storage system to match the workload at hand. For example, for a read-most workload with good locality, the administrator can configure the storage system with appropriate replication and caching levels to maximize read performance. Similarly, for a checkpointing workload, depending on the type of checkpointing used (e.g., application-level, process-level, or virtual machine–level) and on the frequency of the checkpointing operation, the administrator may enable similarity detection to reduce the storage footprint and the network effort [3].

Although *versatility* enables better harnessing the storage resources and ultimately increased application performance, it also requires the administrator to tune the storage system. Such manual configuration is undesirable for several reasons including: The administrator might lack the necessary knowledge about the application and its generated storage workload; temporal variations in the workload or new application versions might make one-time tuning meaningless; and, equally important, performance tuning is time-consuming due to the large size of potential configuration space that needs to be considered.

Our long-term research program aims to answer the following question: *How can we configure a distributed storage system (i.e., enable/disable various optimizations and configure their parameters) with minimal human intervention?* We aim for a solution that allows automated configuration of a versatile storage system.

In this paper, we make initial progress towards answering the above question. We start by providing an in-depth analysis of why manually configuring the storage system is undesirable. We then present the success criteria for a solution that automates configuration tuning. We propose a generic architecture that supports automated configuration tuning and, finally, we instantiate this architecture into a first prototype and integrate it with MosaStore distributed storage system [1].

Our prototype controls the configuration the data deduplication optimizations in MosaStore – more precisely it is able to enable deduplication when the workload warrants using such optimization, and to disable it, thus completely eliminating its overheads, when the workload presents low levels of similarity. Our results demonstrate that the prototype can automatically configure the storage system to deliver performance close to the optimal configuration at a minimal overhead cost.

The rest of this document is organized as follows. The next section presents the necessary background and related work. Section III presents the motivation for automated configuration and derives the requirements that a successful solution should meet. Section IV presents the high-level architecture based on the sense-control-actuate architectural pattern. Section V discusses our first instantiation of this architecture in the context of checkpointing systems: the opportunity to reduce the checkpointing storage footprint by detecting similarity between successive checkpoint images (Section V.A), the specific control loop we use to activate/deactivate similarity detection (V.B), and the integration of our prototype with the MosaStore storage system (V.C). Section VI presents our evaluation of this prototype, and Section VII concludes the paper.

## II. Background and Related Work

This section briefly presents the most advanced efforts, in the areas of database systems and e-services platforms, to address the problem of automating the system configuration. Then, this section describes versatile storage systems, i.e. storage systems that provide optimizations configurable at deployment or run-time, thus, benefiting from automated system tuning. Finally, this section discusses the use of utility functions in storage systems.

*Automating configuration tuning.* The difficulties of manually configuring complex computer systems have been the focus of past research efforts. For instance, in databases, the memory space allocated to execute different SQL queries has a significant impact on overall system performance and determining the optimal allocation is a challenging task for a database administrator. Dageville and Zait [8] describe the Oracle9i's approach to manage this 'SQL memory': a manager dynamically allocates an initial memory buffer for each SQL query, continuously monitors the system's performance, and adaptively shrinks or enlarges this buffer according to the memory pressure in the system.

To address a similar problem, namely automating the tuning of memory allocation for DB2 servers at runtime, DB2 uses an adaptive self-tuning memory mechanism [9]. Similarly to our work, the authors use the concept of 'utility' to guide the optimization decisions of the system. They apply a cost/benefit analysis to shift memory from the sub-systems that need it least to those that need it most.

These solutions differ from our work in three main points: *(i)* they address the problem of dividing just one resource (memory) among several modules, *(ii)* the systems are centralized, and *(iii)* they focus solely on response time as the metric to be optimized.

In the context of *e-services*, dynamic provisioning has been an active research area: the main goal is to use a minimum amount of resources (e.g., web-server nodes) while still meeting a predefined service level agreement [11][12][13]. The variability of the workload provides the opportunity to reduce the amount of resources used when the load decreases, and to reallocate these resources when the load increases. Based on application-specific performance models, dynamic provisioning solutions manage resources in two ways: vertically, by increasing the amount of resources (e.g., memory, CPU share) available to an application on each node; or horizontally, by simply allocating more nodes and running more application instances.

*Versatile storage systems.* Versatility is the ability to provide optimizations tunable at deployment or runtime allowing applications to improve their performance. MosaStore [1] and UrsaMinor [2] are examples of versatile storage systems. MosaStore [1] is an application-optimized storage system that harnesses node-local resources. MosaStore is configured and deployed at application deployment time, and has its lifetime coupled with the application lifetime. A pipeline-based architecture enables versatility: a set of configurable optimizations (e.g., in memory temporary storage, data compression) corresponding to stages in the system data-pipeline can be tuned by the administrator.

UrsaMinor [2] is a distributed storage system that grants clients direct access to storage nodes. To access data, clients ask the object manager for metadata and authorization. This enables the clients to interact with the storage nodes directly and to carry out some of the optimizations at runtime. Examples of optimizations provided are: data encoding, consistency model to provide fault tolerance, and data placement. Compared to MosaStore, one difference is that UrsaMinor intends to be a persistent storage layer shared by several applications (rather than an application-dedicated 'scratch' space). For both systems, the administrator is responsible for choosing the optimizations' parameters to reach desired values for the target metrics and no tool is provided to guide such task.

*Utility functions in storage systems*. Strunk *et al.* [22] provide a tool to help administrators in the task of designing a cost-effective storage solution. The administrator expresses a utility function that captures the financial impact of high-level characteristics (e.g., availability or throughput). The tool receives the datasets to be used in the future system and uses such utility function to explore data placement across storage nodes. Although this work helps the administrator to configure her storage system based on high level metrics, it is limited to optimizations that focus on data placement.

## III. Motivation and Requirements

This section, argues that manually tuning storage system is undesirable, and presents the success criteria for a solution that automates distributed storage system configuration.

### A. *Why Tuning is Hard*

As we mentioned in the introduction, in the context of batch applications, a 'glide-in' deployment scenario allows a per-application, dedicated storage system deployment collocated with the application. A *versatile storage system* unlocks the flexibility required by this approach by exposing all tuning knobs to facilitate per-deployment tuning (and, additionally, even tuning at the level of each stored object [14]), thus enabling applications to more effectively harness the existing storage resources.

Although such versatility can improve an application's performance, it introduces a new problem: the administrator has to manually decide the configuration of the storage system. This new task leads the administrator through a process that involves: *(i)* choosing the objective of the tuning operation – the performance metric(s) (e.g., write operation throughput, execution time or storage space), and the desired target level (e.g., an specific value) for each metric; *(ii)* identifying the optimization features to be enabled and/or their parameters to be tuned; *(iii)* enabling these optimizations and setting their parameters; *(iv)* measuring the performance of the system, and, *(v)* repeating the steps (ii) to (iv) until the desired performance level is attained.

This process is complex, time-consuming, and error prone for several reasons. First, the administrator may not have a good knowledge of distributed systems, the requirements of the running application(s), and their workload. Such knowledge is essential to identify the most frequent and costly operations in order to know which parameters should be changed to reach the desired performance level.

Second, defining the desired target performance level for the performance metric is complex. There are situations where the administrator does not have a specific target level in mind, but, in fact, she is interested in finding the configuration that can extract the best performance for a given application/storage platform combination. In other situations, the administrator has more than one target metric to consider and changing the desired level on one metric affects the others. For example, consider a case where the administrator wants the shortest possible execution time, but the application produces a large amount of data, and she also has constraints on the storage space used. Enabling deduplication can save storage space, but it introduces an additional computational overhead that may make the application slower.

Third, the workload characteristics can change. The administrator may go through the whole effort of configuring the system and have it perfectly tuned for a given application version and workload. However, the workload can change dynamically and in an unpredictable manner. In these cases, the configuration may not attend the desired performance and the administrator would need to keep tuning the system.

Finally, performance tuning is time-consuming. Administrators can spend a long time performing the tuning loop and handling the difficulties described above.

### B. *Solution Requirements*

Based on these difficulties the main question this work addresses is: How can we configure a storage system (i.e., enable/disable various optimizations and configure their parameters) with minimal human intervention?

The solution should meet the following requirements:

- *Be easy to use*. The main goal is to make the administrator's job easier. Thus, we intend to design a solution that requires minimal human intervention to enable/disable various optimizations and choose their configuration parameters. The ideal case is no human intervention.
- *Choose a satisfactory configuration*. The automated configuration should detect a set of parameters that can bring the performance of the system close to the administrator's intention. Note that we are not looking for an optimal configuration, as this might be too costly to determine, but for a configuration that brings the system 'close' to the administrator-defined success criteria.
- *Have a reasonable configuration cost*. The overheads of the automated solution should be low. In other words, the cost required to enable/disable and to configure various optimizations should be small when compared to the cost of the various operations performed by the application. For example, assuming the performance goal is to minimize the application execution time, if the cost to discover parameters that minimize runtime is greater than the time saved, and then automating configuration is useless.

## IV. ARCHITECTURE

Our architectural solution is predicated on one key assumption: we assume the existence of repetitive operations. This makes possible to compare the effects of various configuration options to detect an optimal (or, simply, a good enough) configuration. There is a wide spectrum of applications and deployment scenarios that meet this assumption: from a scenario where the same scientific application (e.g., a climate modeling application) is executed with different input multiple times; to an application that includes regular ancillary I/O operations (e.g., checkpointing); to applications that are inherently cyclical (e.g., a simulation application that regularly outputs its state to enable visualization).

This repetitive pattern enables the use of a monitor-control-actuate loop that continuously adjusts the configuration of the storage system to optimize its performance according to a user-specified utility function (Figure 1).

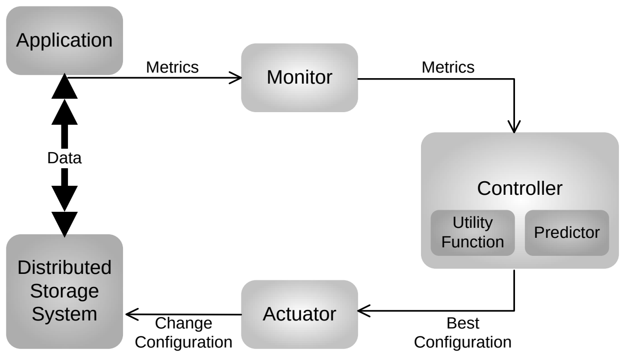

**Figure 1:** Overall system architecture. Control loop based on the monitor-control-actuate architectural pattern.

This closed loop consists of a *monitor*, which constantly monitors the behavior of the storage system. The performance metrics monitored depend on the target optimization goal and on the active configuration at the moment. For a distributed storage system, the monitored metrics can include: the amount of data sent through the network, the amount of data actually stored, the response time for an operation, the compute/memory overheads at the client, and the network latency.

The monitor passes these measurements to the *controller*, which analyzes them, infers the impact of its previous decisions, and may dispatch new actions to change the system configuration to the *actuator*. To perform its tasks, the controller needs: *(i)* a utility function that captures the administrator's intention, and *(ii)* a prediction mechanism that estimates the impact of its actions.

The *utility function* reduces the multidimensional space of the monitored metrics to one dimension that guides the optimization effort. For example, while in most cases the administrator will choose a single metric to optimize upon (e.g., minimize the execution time for I/O operations, minimize the storage footprint) more complex definitions are possible. For example, the administrator may define that she tolerates the execution time to be up to 20% longer than when using the default configuration if the amount of data stored is reduced by more than 30%.

The *prediction mechanism* estimates the impact of a specific configuration change on the target performance metrics. This mechanism can be built using various approaches including: performance model [13] or machine learning [21]. The predictor should provide an estimate for how different configuration changes impact the storage system's performance (given the current state of the system). For example, for a storage system that can save space by compressing data, the predictor receives estimates of the achievable compression rate, and predicts the impact of compression on the time required (and/or throughput) for write operations and the space saved.

Finally, the controller decides whether configuration changes are needed and communicates its decision to the *actuator*. Based on the current delivered utility and state, the accumulated past history of changes, and the estimates for delivered utility in new configurations, the controller can select which configuration provides the best utility and communicate the desired configuration changes needed to the actuator.

The *actuator* performs the configuration changes as instructed by the controller.

While this architecture and a number of its components (the controller, the means to express utility and performance models) are generic, some the other required components are specific for each storage system supported (the actuator, the monitor) or for each application context (the specific utility function and the predictor).

## V. A First Use Case: Checkpointing Applications

To better understand the issues involved by automating the configuration of a storage system, this section focuses on a specific, more constrained, scenario: automating storage system configuration to support checkpointing applications. The goal is to detect at runtime whether the checkpoint images produced by a checkpointing application offer high-enough similarity to warrant enabling online compression and to optimally configure the storage system for the similarity present in the data stream.

Studying this specific use-case allows better gauging the opportunities and challenges posed by automating distributed storage system tuning. After having a solution for this specific application, we plan to expand our work to support a wider array of applications and configuration options.

The rest of this section highlights: *(i)* how checkpointing applications can benefit from similarity detection (Section V.A); *(ii)* how the control loop can be designed to automate system tuning in this case (Section V.B); and *(iii)* the details of our implementation in the context of the MosaStore storage system [1] (Section V.C).

### *A. Similarity Detection for Checkpointing Applications*

Checkpointing is persistently storing a snapshot of the application state. These snapshots (or checkpoint images) may be used to restore the application's state in case of failure or as an aid to debugging.

Depending on the checkpointing technique used, the application characteristics, and the time interval between checkpoint operations, checkpointing may result in large amounts of data to be written to the storage system, and successive checkpoint images may have a high degree of similarity.

Al-Kiswany et al. [3] demonstrate that similarity can be effectively detected at the storage system level (i.e., without application support). Additionally Al-Kiswany et al. [3] analyze traces from applications that use checkpointing. The similarity between successive checkpoint images varies significantly: from almost no similarity when application-level checkpointing is used, to over 70% when process-level checkpointing is used for a popular bioinformatics application (BLAST [7]) and checkpoint images are taken at 15 minutes intervals..

This past work, however, leaves an important gap: it assumes that the storage system administrator has knowledge about the checkpointing technique used by the application, the application characteristics, and the checkpointing interval, and that the administrator configures the system accordingly. Our goal is to take the administrator out of this loop and automate storage system tuning.

Even when the administrator has all necessary information, manual, static tuning can be undesirable. In a scenario where checkpoint frequency varies (e.g., as used for debugging) we aim to dynamically (i.e., at runtime) enable data deduplication only for the times where the application frequently produces checkpoints (that, we assume, have high similarity) and dynamically disable deduplication otherwise.

In our prototype, we use a similarity detection technique based on hashing that works as follows: to detect which parts of two files are similar, the system splits them into fixed-size blocks (corresponding to storage units) and pass the content of each block through a cryptographically strong hash function (MD5, SHA1). If two blocks result in the same hash value the system considers them as similar (and stores only one of them).

This and similar techniques are used by many data deduplication systems based on content addressable storage [1][3][17]) to save storage space, reduce the pressure on the storage system (and possibly reduce the generated network traffic if the system is distributed), at the cost of increased computational cost to compute hash values. Thus, *it is important to estimate when the similarity detection can improve storage system performance,* which depends on how fast similarity detection can be performed, how much I/O bandwidth can be saved, and how fast the I/O operations are.

### B. Control Loop for Configuring Data Deduplication

This section describes an instantiation of the control loop presented in Section IV. We aim to automate the choice between *two* configuration options: data deduplication *on* or *off*.

To instantiate the control loop, we have defined: the metrics to be collected, the set of utility functions the administrator may use to guide the choice of a configuration, and the performance model used as the predictor.

The two success metrics we expose to the administrator to define the utility function are: (i) *the time consumed for checkpointing,* and (ii) *the amount of data stored.* The utility function that drives the optimization is a linear combination of these two metrics. For example the administrator may specify that, regardless of time, the storage footprint should be minimized, or conversely, she may specify that reducing the checkpointing time is primary optimization criteria.

For each write operation, the monitoring information collected is composed of: *(i)* operation timestamp; *(ii)* total duration of the operation (note that this includes the time spent to calculate the hash value and to send data over the network); *(iii)* the total number of blocks received by the write function; and *(iv)* the number of blocks that have already been saved (i.e., they overlap with previous checkpoints). From (iii) and (iv) we can extract the percentage of similarity to be used in the performance model.

The rest of this section describes the performance model used to estimate the impact of configuration changes on the system's performance. Note that an accurate model is the most important part in the control loop to determine that the system converges to a good configuration.

The performance model needs to predict the space used and operation duration metrics used by the utility function in two cases: with deduplication enabled and disabled.

*When deduplication is disabled,* predicting the space for a write operation is simple. The amount of data received by the write operation is the amount of data persisted. The time depends on the I/O operations, which typically present a higher variation and depend on several factors such as the operating system's operations, buffers, and competing I/O operations. The model predicts the write time by analyzing the history of measurements obtained. It calculates a moving average of time to write a block. The moving average is based on the last $N$ operations where $N$ can be tuned to improve the accuracy. Once the time to write a block is estimated, the total time for the write operation can be obtained by multiplying the time to write a block by the number of blocks to write.

*When deduplication is enabled,* the total time is the sum of the time required to calculate the hash values for every block and the time to write the new blocks (we ignore the time to compare hashes as, for this usage scenario, this is orders of magnitude smaller than the previous two). The model estimates the number of similar blocks based on the history of the last measurements. Then, the controller knows how much space is needed and can estimate the I/O operations cost using the same approach as described above when the deduplication is disabled.

The system can obtain a good approximation of hashing overheads by calculating the hash value for just one block and multiplying this value with the total number of blocks to be preserved. Note that the controller can also keep history of the time consumed to calculate hash and estimate the time cost of hashing one block based on such history.

Finally, the controller needs to choose the best configuration. It uses the model to estimate the metrics for the two possible configurations, then it applies the utility function on the estimated metrics, verifies which configuration provides the best utility, and may request the actuator to change the current configuration if it is not the one providing the best utility.

### C. Implementation

This section presents the architecture of the MosaStore storage system and our effort to integrate the control loop that enables/disables similarity detection.

#### 1) MosaStore Architecture

MosaStore aggregates storage space from network-connected nodes. Similar to other distributed storage systems (e.g. GoogleFS [15]), MosaStore uses a *metadata manager* and several *storage nodes* that offer their storage space to be managed by the distributed storage system. To speed up storage and retrieval, MosaStore employs striping [6]: files are split into chunks stored across several storage nodes.

*The metadata manager* is centralized. It is responsible for knowing the state of the storage nodes and storing the metadata (e.g., file's attributes and file's chunk mapping to storage nodes). The storage nodes are responsible for informing the manager when they are online and how much space they have.

To create a file, the client application requests a new file id from the manager and asks the manager to reserve the needed space. The manager returns a list of storage nodes that have space. The client, then, starts writing the chunks of the file to the storage nodes and assembles a chunk-map that maps the chunks to storage nodes for future retrieval. Finally, when the client finishes writing, the chunk-map is sent to the manager to be persistently stored. To detect similarity between the content of successive write operations, a content addressable storage scheme (as described in Section V.A) is used.

#### 2) Implementing the Control Loop in MosaStore

This section describes how each of the components of the control loop presented in Section IV is implemented.

*Monitor*. To collect measurements, we changed MosaStore default write operation flow to capture the metrics as described in Section V.B. The monitor also captures whether deduplication was enabled (by saving the naming scheme used for the chunk-map).

*Controller*. The controller receives measurements from the monitor and archives them. Once the archived history reaches a specific size (minimum default value is five estimates in our initial version), the controller starts to predict the amount of storage space used and time consumed by write operations according to the performance model described in Section V.B. The utility function that drives the optimization is specified by the administrator through a configuration file. The function specifies the weights of each metric (storage space and operation duration) in the total utility. Finally, the controller uses the utility function on the estimate metrics, determines which configuration provides the best utility, and informs the actuator if some configuration change is needed.

*Actuator*. Originally MosaStore did not support configuration changes during runtime. We added a new feature that supports enabling/disabling deduplication at runtime

*Details:* The initial configuration has deduplication activated since the controller needs the level of similarity between writes. Note that if the controller deactivates deduplication, the system cannot provide information regarding the similarity level. To evaluate it, deduplication is turned on again after a given amount of write operations (100 by default in our prototype). The overhead of this approach and alternative solutions are described in Section VI.C.

## VI. Evaluation

This section presents our evaluation of the prototype according to the success criteria listed in Section III.B: effort to configure (Section VI.A), performance delivered by the configuration detected by automated tuning (Section VI.B), and configuration overheads (Section VI.C).

*Experimental setup:* The experimental environment consists of a MosaStore deployment with 10 benefactor nodes. The metadata manager runs on a different machine. All nodes have 2.33GHz Quad-core CPUs, 4GB memory, and 1Gbps NICs.

*Methodology:* Each point in the plots is the average over all write operations performed during the application's execution. It guarantees 95% confidence intervals with ±5% accuracy according to the procedure described in [10].

### A. *Effort to Configure*

The proposed solution requires a minimal configuration effort: the administrator only needs to specify her intention. In our prototype this is expressed by specifying the weights of runtime and storage footprint size in the utility function. Note that, the default configuration of the utility function assumes that the administrator's intention is to minimize runtime which we assume is the most frequent situation.

### B. *System Performance*

To analyze how satisfactory the automated configuration is, we compare the performance of the system using automated tuning and the performance using the two manual configurations available: similarity detection always on and always off.

We use a synthetic workload modeled based on our past experience analyzing checkpoint workloads [3]. The synthetic application writes 100 snapshots of around 256 MB each, with similarity of 70% between writes.

We first analyze the case where the administrator is interested in minimizing runtime (i.e., the lower the runtime, the higher the utility). Figure 2 presents the average time per write for the three different configurations. The automated solution is able to detect that similarity is high enough to offset the computational overheads generated by hash-based similarity detection. Indeed, the automatic solution performs as good as a configuration for which similarity detection is always activated, which is the best configuration in this case.

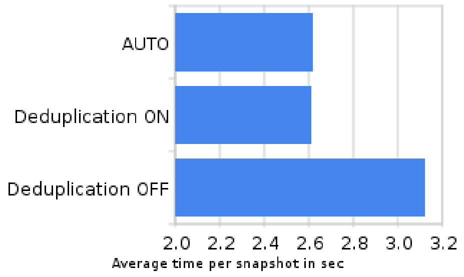

**Figure 2:** Average time to write a snapshot of 256MB and similarity of 70% using different configurations.

To analyze system's performance under different scenarios, we executed the same synthetic application yet varying the level of similarity present in the data (this is equivalent to varying the frequency of the checkpointing operation). Figure 3 presents the average time required for checkpointing. For up to 50% of similarity, on our platform similarity detection should be deactivated as the hashing overheads are not compensated by savings in I/O operations. The results confirm that automated tuning makes the correct configuration choice.

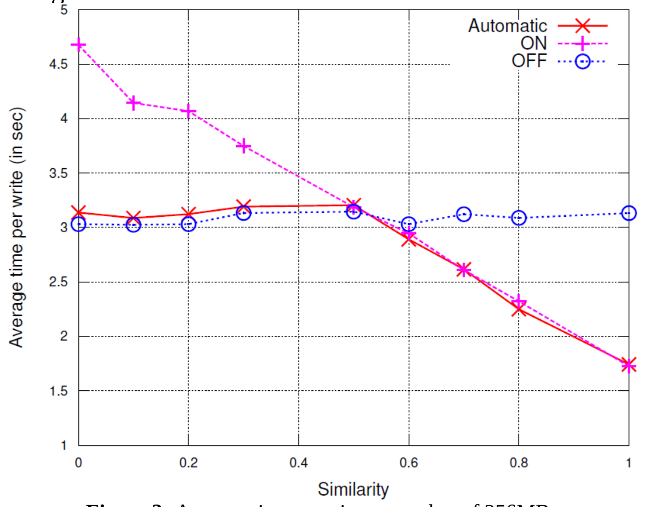

**Figure 3:** Average time to write a snapshot of 256MB.

The slightly lower performance when using automated tuning at low similarity levels is the result of the needs to estimate the similarity level in the data stream before making a decision (these overheads are discussed in Section VI.C).

The experiments also consider write operations for different data sizes (32 MB, 64 MB, 128 MB, and 512 MB) while varying the similarity level. Although the amount of time required for each operation varies, automated tuning always makes the correct decision (and the results are similar to those presented in Figure 3).

To analyze a case where the administrator is interested in more than one metric, the controller input is set to have a utility function where the time and storage space as equally important (i.e., each of the two terms receives 50% weight).

Figure 4 shows the relative utility gained when writing checkpoint images of 256MB with 70% similarity. The results are normalized by the value of the lowest utility among different configurations (deduplication off in this case) to provide a clear comparison. Similarly to the case when the system aims to minimize runtime, the automated solution is able to detect that the similarity is high enough to provide a better utility. Indeed, the automatic solution provides the same utility (around 66% higher than the worst configuration) in cases where similarity detection is always activated, which is the best configuration for this case.

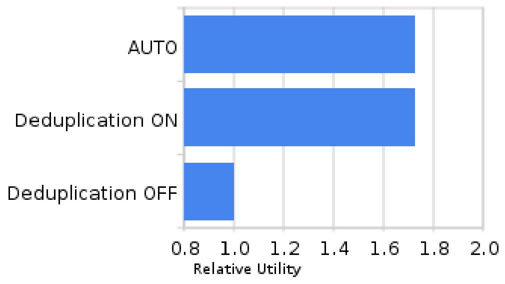

**Figure 4:** Average relative utility to write a snapshot of 256MB and similarity of 70% using different configurations.

We analyze the system when using this utility function while varying the similarity rate in the data stream. Figure 5 shows the average relative utility for these experiments. For up to 20% similarity, the best configuration for the execution platform is having the deduplication deactivated as the time needed to hash the data is not enough to balance the storage space saved and the savings in I/O operations. As similarity increases, the detection cost is paid off by the time saved for the I/O operations and the amount of storage space saved. As the figure shows, the automated configuration choice always matches the optimal manual configuration.

### *C. Cost of Automated Tuning*

To evaluate the configuration cost, we analyze the amount of extra resources used for the automated configuration: CPU time and memory.

Memory overheads are minimal: the controller only keeps a history of past measurements. Such history is short since the controller applies the moving average for the last *N* measurements. In the experiments, we use *N*=5 which results in less than a kilobyte of extra memory. Older data can be discarded or persisted on disk for future analysis.

As the experiments presented in the previous section suggest (see Figure 3) the performance of the configuration chosen by auto-tuning (that includes the auto-tuning overheads) is within 5% of the performance of the best configuration.

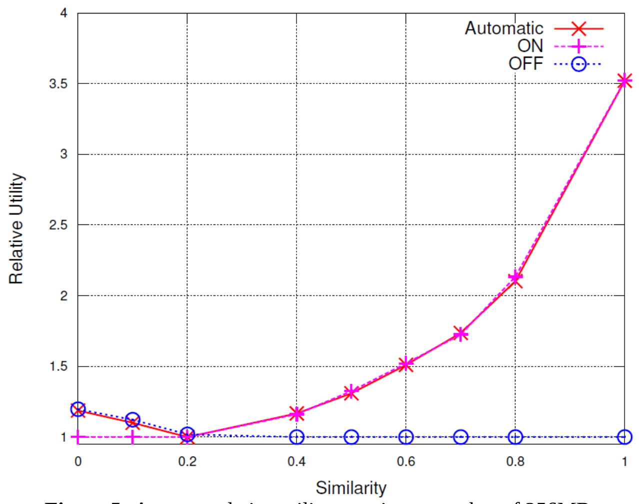

**Figure 5:** Average relative utility to write a snapshot of 256MB.

The compute overhead generated depends on the state of the system. If deduplication is activated, the overhead to estimate the similarity level in the data stream (the single largest component of the overhead) is zero. However, if deduplication is deactivated, the system still needs to estimate the similarity level. Thus, it starts hashing the data, which slows down the write operation by up to a factor of two for cases where there is no similarity among writes. Note that such additional cost exists only until the automated solution detects that hashing the data is not worth. From this moment, this cost is amortized over the following write operations.

To reduce this cost, we plan to evaluate two options: *(i)* using sampling to reduce the volume of data analyzed when detecting the level of similarity, and *(ii)* moving similarity estimation offline, outside the critical performance path.

## VII. SUMMARY AND FUTURE WORK

Versatile storage systems aim to maximize storage resource utilization by supporting the ability to 'morph' the storage system to best match the application demands. To this end, versatile storage systems significantly extend the deployment- or run-time configurability of the storage system. This flexibility, however, introduces a new problem: A much larger, and potentially dynamic, configuration space makes manually configuring the storage system an undesirable if not unfeasible task.

This paper presents our initial progress towards developing an automated solution to configure a distributed storage system, that is, to enable/disable its various optimizations and configure their parameters with minimal human intervention. To the best of our knowledge, no previous work addressed the question of automated tuning the context of distributed storage systems. In fact, our bibliography review revealed a large number of studies that highlight the sensitivity of the storage system performance to workload characteristics, and/or deployed optimizations and their specific configuration. Yet, none of these works attempt to automate the choice of the storage system configuration, even for specific optimizations.

To better understand the challenges entailed by automated storage system configuration, we study one optimization, namely online data compression through similarity detection, in the context of checkpointing applications. We note that progress in this direction not only sheds lights on the challenges related to automate tuning, but also directly applied in practice: Current operational practice when running checkpointing applications requires a wealth of information about the checkpointing characteristics (e.g., checkpointing technology, frequency) to configure the storage system. Automating the choice of enabling/disabling deduplication allows decoupling the concerns of the application's scientist/developer from those the storage system operator.

Our experiments demonstrate that the prototype we have developed correctly configures the storage system to optimally enable/disable deduplication at a minimal overhead. While we are encouraged by our results so far we acknowledge that, even for this optimization, we have not yet explored the full space of configuration options that may require automated tuning. To fully explore this space, we are currently extending the prototype in a number of directions: (i) reducing the cost of quantifying data similarity while the controller verifies whether similarity detection optimization should be active or not; (ii) using different block's sizes to tune the detection, and (iii) automated control of a more complex approach to detect similarity that determines block boundaries based on data content [3].

Additionally, in the medium term, we plan to explore support for a wealth of other optimizations and more powerful utility functions. We anticipate that more optimizations not only will bring new challenges to automate the configuration of each of these optimizations but also the challenge of dealing with the interference between these optimizations. Richer utility function will allow the administrators to define metric constraints to metrics.

## ACKNOWLEDGMENTS

We would like to thank Steven Liu and Keven Chen for their participation in the first version of the prototype. This research was supported in part by the DOE ASCR XStack program under grant DE-SC0005115.

## REFERENCES

[1] S. Al-Kiswany, A. Gharaibeh, M. Ripeanu, "The Case for a Versatile Storage System", Workshop on Hot Topics in Storage and File Systems (HotStorage'09), Oct. 2009.

[2] M. Abd-El-Malek, W. V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad *et al.*, "UrsaMinor: versatile cluster-based storage", Proc. of the 4th USENIX Conf. on File and Storage Technologies, Dec. 2005.

[3] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing", Proc. of the 28th International Conf. on Distributed Computing Systems, Jun. 2008.

[4] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan, M. Ripeanu, "Enabling cross-layer optimizations in storage systems with custom metadata", Proc. of the 17th International Symp. on High Performance Distributed Computing, Jun. 2008.

[5] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose and R. Hawkes, "Jumbo store: providing efficient incremental upload and versioning for a utility rendering service" Proc. of the 5th USENIX Conf. on File and Storage Technologies, Feb. 2007.

[6] J. H. Hartman and J. K. Ousterhout, "The Zebra striped network file system", Proc. of the 14th ACM Symp. on Operating Systems Principles, Dec. 1993.

[7] S. F. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool. Molecular Biology", 1990. 215: pp. 403–410.

[8] B. Dageville and M. Zait, "SQL memory management in Oracle9i", Proc. of the 28th International Conf. on Very Large Data Bases. Aug. 2002.

[9] A.J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, M. Surendra, "Adaptive self-tuning memory in DB2", Proc. 32th International Conf. on Very Large Data Bases, Sep. 2006

[10] R. Jain, "The Art of Computer Systems Performance Analysis". Wiley-Interscience, New York, NY, Apr. 1991

[11] S. Ranjan , J. Rolia , H. Fu , E. Knightly, "QoS-Driven Server Migration for Internet Data Centers", 2002.

[12] J. Chase, D. Anderson, P. Thakar, A. Vahdat, R. Doyle, "Managing Energy and Server Resources in Hosting Centers", Proc. of the 18th ACM Symp. on Operating System Principles, Oct. 2001.

[13] R. P. Doyle, J.S Chase, O. M. Asad, W. Jin, and A. M. Vahdat, "Model-based resource provisioning in a web service utility", Proc. of the 4th Conf. on USENIX Symp. on Internet Technologies and Systems, Mar. 2003

[14] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan and M. Ripeanu, "Enabling cross-layer optimizations in storage systems with custom metadata", Proc. of the 17th international Symp. on High Performance Distributed Computing (HPDC '08), Jun. 2008.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System" Proc. 19th ACM Symp. on Operating Systems Principles, Oct. 2003.

[16] S. A. Weil, S A Brandt, E. L. Miller, D.D. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system", Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI'06), Nov. 2006.

[17] S. Rhea, E. Cox, and A. Pesterev. "Fast, inexpensive content-addressed storage in foundation", Proc. Of *USENIX Technical Conf.*, Jun. 2008.

[18] D. Bradley, O. Gutsche, K. Hahn, B. Holzman, S. Padhi, H. Pi, D. Spiga, I. Sfiligoi, E. Vaandering ,and F. Würthwein. "Use of glide-ins in CMS for production and analysis". In Journal of Physics: Conf. Series. Vol. 219. Part 7. 2010

[19] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. "Explicit Control in a Batch-Aware Distributed File System", In Proc. of Symp. on Networked Systems Design and Implementation NSDI'04, 2004

[20] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, M. Wilde. "Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming", Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), Nov. 2008

[21] D. Michie, D. J. Spiegelhalter, C. C. Taylor, and J. Campbell, Eds. "Machine Learning, Neural and Statistical Classification". Prentice Hall. 1994.

[22] J. D. Strunk, E. Thereska, C. Faloutsos, G. R. and Ganger, "Using utility to provision storage systems", Proc. of the 6th USENIX Conf. on File and Storage Technologies , Feb 2008.