ABSTRACT
Developing a distributed system is a complex and error-prone task. Properly handling the interaction of a potentially large number of distributed components while keeping resource usage low and performance high is challenging. The state-of-the-practice on performance evaluation focuses on employing profilers to detect and fix potential performance problems, but lacks decision support information on when profiling effort should stop.
To help address this issue, we propose the use of a prediction tool to estimate the expected performance of a complex system, and describe our experiences with employing this tool to support the development of a distributed storage system.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications; D.4.8 [Performance]: Modeling and prediction

1. INTRODUCTION
Efficiency, in terms of resource usage to deliver better performance, is an important systems success criterion. In particular, for large scale systems that are likely to be used by several clients and handle a variety of workloads, this non-functional requirement is crucial to achieve success and leverage their adoption. High performance must be reached while keeping the cost low, by using the least possible amount of resources, while delivering the fastest time-to-solution. Because of its importance, performance should be addressed in the early stages of software development [4], given that maintenance costs increases over time [21].

The state-of-the-practice in addressing performance consists of analyzing a system by employing profilers to monitor its behavior [6]. During the analysis, these profilers should pinpoint the regions of the code where execution takes longer and, therefore, should receive more attention and be optimized. Although profiling-based optimization is undoubtedly a key part of the development of complex system, deciding when the efficiency of a system has reached a “good enough” level where extra effort is unlikely to render extra benefits is still a challenge.

In this context, where optimal performance is crucial for the functioning of the system, we believe that developers should be able to rely on tools that provide an estimate of the expected performance (i.e., a baseline) and compare it to the actual performance while developing the system. This process would be conceptually similar to the use of automated tests to check functional requirements, but with the tool being a system’s performance predictor for a specific efficiency. Ideally, the tool should also provide information about where the system is losing performance, similar to automated tests identifying the part of the code that is failing for a specific requirement.

Note that the usage of a performance predictor in this scenario is different from its common usage. Typically, performance predictors rely on monitoring information of an already deployed system to support configuration decisions, capacity planning, or online provisioning adaptation during a deployment-planning or post-deployment phase. The scenario that we describe advocates for the usage of a performance predictor as a tool incorporated in the software development phase to provide a performance baseline to guide the effort of performance improvements.

In this work, we advocate the importance of using performance predictors to develop complex software systems by narrating our experience in applying a specific performance predictor in the development of a complex and error-prone distributed storage system. We demonstrate how the performance predictor helped developers to properly handle the interaction of a potentially large number of distributed components in a storage system while avoiding to compromise efficiency due to complex interactions of these components.

By describing our experience, our goal is to investigate the following two questions: How can a performance predictor bring benefits to the software development process?, and What are the limitations and challenges of using a performance predictor as part of the development process?

The predictor was useful to set goals for the system’s performance despite the difficulties of proper seeding the model and mimicking the used benchmarks to evaluate its performance. Specifically, this approach gave confidence in the implementation for some scenarios and pointed out situations that needed further improvement. The latter made possible an improvement of the system’s performance by up to 30% and a decrease of 10x in the response time variance.

2. CASE STUDY
The focus of this paper is to investigate the following questions:

• How can a performance predictor bring improvements to the software development process?

• What are the limitations and challenges of using a performance predictor as part of the development process?

We used these questions to drive our study, rather than testing specific hypotheses. To that end, we employed our
performance prediction approach during the development of a distributed storage system. In this section we describe both the system used in this study, MosaStore (2.1), and the design of the performance predictor (2.2).

2.1 Object System

Our study focuses on MosaStore [3], a distributed storage system that can be configured to enable workload-specific optimizations. It is mainly implemented in C, and has approximately 10,000 lines of code. To this date, more than fifteen developers were involved in different versions of its implementation. Additionally, it has been used in several different projects in different laboratories (e.g., [4,5,9,10,20]).

2.1.1 MosaStore Architecture

MosaStore employs a widely-adopted object-based storage system architecture (such as that adopted by GoogleFS [11], PVFS [12], and Ursa Minor [2]). This architecture includes three main components: a centralized metadata manager, storage nodes, and a client-side System Access Interface (SAI), which uses a virtual filesystem (VFS) via the FUSE (Filesystem in Userspace) [1] kernel module to implement a user-level file-system that provides a POSIX API in MosaStore (see Figure 1). The manager maintains the stored files’ metadata and system state. To speed up data storage and retrieval, the architecture employs striping ([13]: files are split into chunks stored across several storage nodes.

![MosaStore Architecture](image)

Figure 1: MosaStore Architecture as described by Al-Kiswany et al. [5]

2.1.2 MosaStore Modus Operandi

More relevant to this study, MosaStore can be configured to employ several different data placement policies, a variable number of storage nodes, and a variable number of clients that can be collocated with the storage nodes or not.

These possible configurations can result in a number of different deployments to the system, where storage nodes and SAIs interact in different ways, with different overheads, and leading to different performances. In this context, the developers of MosaStore initially applied profiling analysis during its development, but in an ad hoc manner: there was no baseline of the best possible performance, and no indicator of when they should have started or stopped the profiling process. This scenario brought a challenge to MosaStore’s developers, which we believe is present in many other systems: for a given configuration, what performance should an implementation deliver to be considered efficient?

2.2 Performance Predictor

To support the development in the performance improvement phase, we decided to design and implement a performance predictor to be applied as part of the development process. This section describes the general idea behind the performance predictor as a mechanism to provide a performance goal for the system.

2.2.1 Local Deployment: A Simple Scenario

In its simplest configuration, MosaStore uses a single machine. By deploying all three components locally, the system should not face efficiency loss related to the coordination of distributed components neither due to contention on the network substrate.

The expected ideal performance for such a scenario can be set by external tools that serve as benchmarks for local filesystems, or even by simple analytical models. MosaStore developers already used this approach that served as simple “sanity” checks for the system’s performance. For example, a scenario where a single SAI uses a manager and a storage node deployed on the same machine should provide a sequential write throughput similar to throughput of a write to a local filesystem. For our system, the performance baseline to be achieved was set based on the performance of a different local filesystem.

2.2.2 Distributing the System Components

As the number of components in the system grows, predicting performance of a distributed system accurately and in a scalable manner is challenging. Analytical models for the contention of the distributed environment on the storage nodes and on the manager, or the network performance that can capture different applications’ workload are increasingly more complex. Moreover, there are no tools that can be used as a baseline, and a similar system would play the role of a competitor, which is not necessarily the best performance that the system can deliver in a given setup. One could indeed use a competitor system to set the goal for the performance of a system, but this approach still misses the point of informing the developers that there is more to be extracted from the system.

Our approach for the distributed configurations builds on top of the single-machine scenario. We consider that some overhead of the distributed system is inherent from the distributed environment (e.g., data transfers, network sharing) while another part is directly related to the implementation and deployment environment (e.g., resource locking, network packet loss). To capture the inherent interactions of the distributed environment, we use a queue-based storage system model [3], which is not affected by performance anomalies caused by the implementation.

The performance from the single machine scenario seeds the different components of the model (SAI, storage module, and manager), capturing the performance of the system components when there is no efficiency loss caused by the distributed environment. Finally, a utility tool seeds the performance of the network part of the model.

2.2.3 Predictor Input/Output

The predictor requires three inputs: the performance characteristics of system components, the storage system configuration, and a workload description. The performance characteristics of system components used to seed the model are
based on the single-machine scenario. Not present in the single-machine case is the network substrate, which is characterized via a utility tool to measure network throughput and latency (e.g., iperf [16]). We have implemented the above model as a discrete-event simulator in Java. The simulator receives as input: a summarized description of the application workload and a description of the deployed system:

- The workload description contains per application task I/O traces (i.e., open, read, write, close calls with the call details timestamp, operation type, size, offset, and client id) captured after just one execution.
- The deployed system’s description contains two parts:
  - The first part describes the system-wide configuration parameters (currently, replication level, stripe-width, chunk size, and data-placement system-wide) and the details of the system: number of hosts, number of storage nodes and clients, whether storage and clients nodes are collocated in the same hosts. The second part characterizes the performance of system services: service times for network, client, storage, and manager services (the process of identifying these values can be found in our technical report [8]).

A simulator instantiates the storage system model with the specific component characteristics and configuration, and simulates the application run as described by the workload description. Once the simulation ends, it provides an estimate of the performance of the system for a given setup.

The developers of the system actually deploy and run the system in the same setup used to describe the simulation and measure the actual performance. They then compare the estimates of the actual performance and decide whether they are close enough to the goal. If they are not close enough, they pursue more optimizations by debugging the system, and follow this process iteratively as part of the development phase.

A technical report [8] presents a complete discussion of the predictor design, its input, the model seeding process, and an evaluation for its response time, scalability, and accuracy.

3. USING THE PREDICTOR DURING DEVELOPMENT

As part of the development cycle of MosaStore, synthetic benchmarks run on the top of the system to mimic real workflows’ access patterns [20] and real applications to evaluate the performance of the system. These synthetic benchmarks represent worst situations in terms of the storage system performance as they are composed exclusively of I/O operations, which are intended to create contention and expose the overheads of the system.

The performance predictor was included in the development process to produce performance estimates of the same environments used to evaluate the performance of the storage system.

In a number of cases, the predicted and actual performance were close, providing confidence that the implementation was efficient. There were cases however in which the actual and predicted performance varied significantly, and they highlighted complex performance-related anomalies, leading to debugging effort. This section presents three of these performance anomalies cases by describing how they affected the system. Specifically, it describes how the predictor identified performance anomalies, how the predictor was useful to address them, and how the fixes impacted the system performance.

The cases presented here focus on two benchmarks:

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. 19 application pipelines, one for each machine, run in parallel and go through three processing tasks.

Reduce benchmark. A single compute task uses input files produced by multiple tasks. In the benchmark, 19 processes run in parallel on different nodes, each consumes an input file, and produces an intermediate file. The next stage consists of a single task reading all intermediate files, and producing the reduce-file.

Experimental Setup. The benchmarks run on a testbed of 20 machines each with an Intel Xeon E5345 2.33GHz CPU, 4GB RAM, and 1Gbps NIC. MosaStore’s manager runs on one machine while the other 19 machines each run both a storage node and a client access module. The plots show the average turnaround time and standard deviation for at least 15 trials guaranteeing a 95% confidence interval with relative error of 5%.

To make the impact of these improvements clear, Figure 2 and Figure 3 show the average time for the two benchmarks. Figure 2 has four bars: one representing the time obtained from the initial version of the storage system, two showing the versions of the system after fixing the performance anomalies (two anomalies are grouped together) described in this section and the last showing the predicted time. Figure 3 has only one of the performance anomalies since the other one does not impact this benchmark.

Accuracy of the predictor is not the focus of this work, which rather focuses on how the tool can be useful for supporting the development process. A technical report [8] presents a deep evaluation of the accuracy of the predictor using synthetic benchmarks and a real application, including a detailed presentation of each benchmark.

![Figure 2: Impact of fixing performance issues for Pipeline benchmark.](image)

3.1 Lack of Randomness

Context. Whenever a client creates a file, it contacts the manager to obtain a list of storage nodes that will be used
The overall gain was around 2.5 seconds in Figure 2, and combined with the lack of randomness case, determined by the manager and is shuffled according to a random seed.

The order that the storage nodes will be used to write is determined by the stripe-width configuration parameter. The number of storage nodes will be used to write is determined by the manager and is shuffled according to a random seed.

**Problem.** The manager used the same seed to shuffle the list of storage nodes returned. Therefore, when the system was set to use the maximum stripe-width, all clients obtained the list in the same order, issuing write operations to the storage nodes in the same order. This created temporal hot-spots where the first storage node received connections from all clients while other storage nodes were idle.

**Detection.** When developers are interested in obtaining the response time for a given component of the system, they can simply obtain this by turning on a log option that measures the time from the reception of a request until its response at the component acting as a server for that request. In fact, this functionality was key to detect the problem for the next two cases. By studying the behavior, the developers stated a hypothesis for the problem, fixed it, and executed the benchmarks again.

**Fix.** The fix was to change the algorithm that shuffles the list of storage nodes to use a different seed every time it is invoked, providing the needed randomness of node allocation.

**Impact.** The performance improvement can be seen in Figure 2 which combines this and the next case.

### 3.2 Lock Overhead

**Context.** Whenever a client reads or updates metadata information (e.g., open or close a file), it contacts the manager that needs to lock multiple updates over the metadata to avoid race conditions.

**Problem.** The developers of the initial version of the system chose a conservative approach by opting to lock large code blocks that are called during the client invocation rather than locking only the critical regions.

**Detection.** Based on the manager’s service times, the developers could verify that a few requests to the manager took reasonably longer than the average. Once they spotted the problem in the manager, they started a debugging process to spot the parts of the code that took longer and detected large and unnecessary lock scopes.

**Fix.** Reduce the lock scope.

**Impact.** The performance improvement can be seen in Figure 2 and combined with the lack of randomness case, in Figure 3. The overall gain was around 2.5 seconds in average. Note that the variance also decreased.

### 3.3 Connection Timeout

**Context.** Similarly to the way the predictor helped us revisit assumptions about the implementation of the system, it also pointed out issues related to the middleware stack that the storage system relies upon.

If a client contacts a server to establish a TCP connection, it waits for a specific timeout before trying it again. Note that this is to establish the connection and not the TCP window management that happens once the connection is established.

**Problem.** In cases where too many clients tried to send data to a storage node at the same time, the storage node dropped some SYN packets (packet used to establish a TCP connection) because there were too many packets in its queue. The client then waited for a timeout defined by the TCP-implementation which is 3 seconds, consequently taking the average time to write the data in the benchmark to a much higher value than expected.

**Detection.** The developers also used the logging of service time of each component for this case. The simulator provides similar information, which was compared against the system’s logs to identify when there was a mismatch of the time predicted for the storage nodes to finish a request and the predicted time. Moreover, system logs showed an actual longer waiting time.

In this case, however, obtaining service times for each request did not help much because the problem actually prevented the request to reach the storage component in the first place. To find the gap between predicted and actual time, the developers added a new functionality that logged service time at the client side. By collecting these times and analyzing them, the developers could see that most of the requests were processed around the time predicted, and the other requests three seconds later. By debugging the code, the developers found that this gap happened in the network connection phase of the request and discovered how the TCP implementation could lead to this case.

**Fix.** The developers decided to define their own timeout instead of relying on the system’s default. To do so, they needed to change the implementation from blocking to non-blocking sockets during the connection phase.

**Impact.** The performance improvement can be seen in Figure 3 and Figure 4. In this case, the gain affected mostly Reduce benchmark (Figure 3) due to its data flow, i.e. several machines write to just one.

### 4. PROBLEMS FACED

The limitations of using a separate model to capture the actual implementation behavior are well-known and well captured in a sentence by George Box: “Essentially, all models are wrong, some are useful”. Indeed, properly capturing the system’s behaviour to provide useful performance estimates or correctly define the deployment to be simulated can be challenging.

In some cases, the system analysis was affected by simplifications in the model, shortcomings of the seeding process, or incorrect assumptions about the deployment platform. We highlight here four of the problems we faced:

**Workload description mismatch.** The predictor receives a collection of I/O operations based on the log of a
benchmark’s execution. These benchmarks launch processes on the specific nodes via ssh. The actual time to launch one round of processes in the cluster varies from 0.1s to 0.3s. This variance is not related to the storage system, but it affected the comparison between actual and the predicted time. After realizing this variance, we added it to the workload description to be simulated.

Platform Mismatch. During the execution of the benchmarks, we realized that one of the machines used in the experiments was reasonably faster than the others. In this case, we had specified a deployment environment that was not the one actually used. Running the seeding process on the fast machine, and specifying the deployment environment properly fixed the problem.

Lack of read priority in the simulator. In cases where there are several replicas of a chunk in the system, the client selects one randomly. However, the storage system gives priority to chunk replicas located in the same machine. The simulator did not capture this priority and it led to mismatch prediction in cases with higher replication levels. It happened in the initial stage of the simulator implementation and unit tests captured the problem.

Modeling connection timeout. The connection timeout problem described in Section 3.3 shows a case where the implementation did not handle the connection establishments properly. The implementation changed to avoid long waits. Nevertheless, a remote machine receiving more requests to open connections than it can handle would still drop some of the requests, which may increase response time.

This situation describes a scenario where the model does not capture the actual behavior of the deployed system. We consider this problem to be a result of the environment. Therefore, we did not extend the model to include this behavior and adjust the prediction since the predictor should provide the performance in absence of implementation or environment issues.

5. DISCUSSION

In this section, we tackled the two research questions on applying a performance predictor as a tool to leverage the development of complex systems with the goal of calling the community’s attention on the value of producing such a tool.

How can a performance predictor bring benefits to the software development process?

Our case study showed the usefulness of having a performance prediction tool that can set a performance target of a system in deployments with different configurations and scale. The predictor brought confidence in the results obtained in several scenarios, was successful in pointing out scenarios that needed improvement, and could support the improvement effort. In fact, the performance of the system improved by up to 30% and the response time variance decreased by almost 10x in some scenarios (e.g., 9) as a result of applying this approach.

We believe that applying a performance predictor in other complex systems would also be beneficial. Similar to back of the envelope calculations, the predictor evaluates expected performance bounds for a given system. A richer predictor can however take back of the envelope calculations a step further, because the model it uses provides building blocks to guarantee its usefulness in complex scenarios where back of the envelope estimates could be intractable. Not only developers can use it to obtain a baseline to detect performance anomalies, but also to evaluate the potential gains of implementing new complex optimizations or to study the impact of faster network and nodes to a system.

What are the limitations and challenges of using a performance predictor as part of the development process?

Overall, we could split the problems that we face with this approach into two classes: (i) on having a good performance predictor, and (ii) on its usage during the development phase.

The first class covers problems related to performance predictors in general [6], such as accurate model and proper seeding. We describe the problems we encountered in our case study in [4]. Additionally, the approach we apply has one main challenging point: define the baseline to set the performance goal. Since we were interested in a distributed system, we targeted this challenge in two phases: a local scenario goal set by an external tool (2.2.1), and a distributed scenario goal captured by a queue based model (2.2.2). Other approaches, such as microbenchmarks and analytical models, can be used to define these baselines. The proper choice, however, is system-dependent.

The second class is related to how developers use such a tool. In our experience, after the initial month of use, the developers started skipping the prediction tool as part of the cycle. During the initial phase of applying the tool to development process, the developers could discover several problems and this led to several improvements of the system. After this initial phase, the gains of frequently running the tool and actual benchmarks decreased since most of the problems were addressed.

We believe that this problem is similar to having a suite of automated tests that take too long to execute. In these situations, the test suite tends to be split in different suites, one to be used often by the developers and another (or several) to be used as pre-commit or during nightly builds. Hence, we advocate that the predictor tool should be used as part of a large suite of automated tests. In fact, we believe that the developers should also specify a percentage of tolerated overhead over the predicted performance to define acceptance tests as part of this large suite of automated tests.

6. RELATED WORK

This section relates our work with other attempts to integrate performance prediction/analysis into the software development process. Our goal is to discuss how prediction techniques are used to improve software development rather than presenting the state of the art on predicting performance.

Balsamo et al. [6] conducted a survey of model-based performance prediction at software development time. According to them, the first approach to integrate performance analysis into the software development process was conducted by Smith [18] and named Software Performance Engineering (SPE) methodology. SPE is a generic methodology that relies on software and system execution models to specify resource requirements and conduct performance analysis, respectively. Our main goal is to have a predictor to set a goal
for the performance, although it can also assist performance analysis and debugging.

Over the years, some approaches were proposed based on the SPE methodology\textsuperscript{[7,19,22]}. Most of them analyze UML diagrams (e.g., Class, Sequence, Deployment diagrams, etc.) to build software execution models to achieve performance prediction. For example, Williams and Smith\textsuperscript{[22]} employed Class, Sequence, and Deployment diagrams enriched with Message Sequence Chart to evaluate performance of an interactive system designed to support computer aided design activities. They described their experience in using these architectural models to verify performance requirements during software design in order to support architecture trade-off decisions. Cortellessa and Miranda\textsuperscript{[7]} also used UML diagrams to generate a queuing network-based performance model. However, they prioritized State transition and Interaction diagrams as source of information.

There are two main differences between these approaches and ours: First, our main goal is to have a predictor to set a goal for the performance instead of only understanding how the performance of a given implementation will be. Second, these approaches require UML diagram analysis to build Queuing Network-Based (QN) models, while we build them from scratch from a coarser granularity (main system components).

On the one hand, the lack of UML prevent us from automating some steps of model specification, such as execution paths specification. On the other hand, system documentation (e.g., UML diagrams) tends to be neglected over time\textsuperscript{[15,17]}. By focusing on the main components, we reduce the effort of keeping the UML model updated and accurate.

Heger et al.\textsuperscript{[4]} proposed an approach to: i) detect performance regressions during software development and ii) isolate the root cause of such regression. The former is achieved by employing unit testing over the history of the software under analysis and comparing the performance results to uncover possible performance regression introductions. The latter is achieved by applying systematic performance measurements based on call tree information extracted from unit tests’ execution. Besides controlled experiments, the authors also present their successful experience while investigating a performance regression introduced by developers of Apache Commons Math library.

This work relates to ours in the sense that the authors advocate for the introduction of performance evaluation into the software development cycle. However, our work focuses on performance prediction rather than detecting possible bottlenecks introduced by code changes. In summary, we are concerned about how the system will perform given a specific workload scenario before the feature development, while Heger et al. try to detect performance regression as soon as they are introduced into the code.

7. CONCLUDING REMARKS

In this paper, we presented an experience where the use of a performance predictor was useful to support the development process of a distributed storage system. This approach could detect cases where performance anomalies were present, using the predictor to address such anomalies. It also increased the confidence in the implementation when the actual performance matched the predicted one\textsuperscript{[8]}.

We proposed a performance predictor 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; and with a low runtime.

Based on the experience described in this paper, we suggest that the use of performance predictors during software development can help developers to deal with this non-functional requirement similar to how automated tests verify functional requirements. In particular, when developing a large scale and high performance system, in which performance is a key concern, we believe that a predictor is useful to early detect potential performance problems and, consequently, reduce the effort of removing performance bottlenecks.

Acknowledgements

We would like to thank Samer Al-Kiswany, Emalayavan Vairavanathan, and Hao Yang for their collaboration in MosaStore development; Abmar Barros for his support in the development of the performance predictor; Elizeu Santos-Neto and Abdullah Gharibech for their collaboration in keeping the quality of the code via code reviews. This research was supported in part by the DOE ASCR XStack program under grant DE-SC0005115.

8. REFERENCES

\begin{thebibliography}{10}
\bibitem{FUSE1} FUSE: Filesystem in userspace.
\url{http://fuse.sourceforge.net/}
\end{thebibliography}


