OpTune: Multi-Point Performance Engineering in Server Systems

Cheng Li†  Íñigo Goiri‡  Abhishek Bhattacharjee†  Ricardo Bianchini‡†  Thu D. Nguyen†

†Rutgers University
{chenglii, abhib, tdnguyen}@cs.rutgers.edu

‡Microsoft Research
{inigog, ricardob}@microsoft.com


Abstract
Modern server systems encompass multiple components and/or layers containing configuration parameters that can affect performance. Managing the interplay of these parameters is becoming increasingly challenging as the complexity of server systems continue to grow. In this paper, we propose a framework called OpTune to help system administrators tune these parameters. OpTune asks administrators to specify objectives to shape the performance CDFs of systems; e.g., minimize the median response time while keeping the 99th percentile below a target value. In fact, administrators can specify entire target CDFs. OpTune then uses a graphical representation of the system, performance instrumentation and profiling, and manipulations of the profiled CDFs of components to configure the system. We demonstrate the broad utility of OpTune by integrating it into three different, widely-used systems: a Web server, a filesystem emulator, and a MapReduce server cluster. Evaluation results demonstrate that OpTune successfully helps administrators to quickly identify configuration parameter values to best achieve the desired performance behaviors.

1. Introduction
Modern server systems encompass multiple components and/or layers containing configuration parameters that can affect performance. Examples include parameters controlling the amount of parallelism (e.g., number of threads), the size and replacement policy used for memory caches, and the scheduling policies for processing workloads. As the complexity of server systems continues to increase, managing the interplay between these configuration parameters to precisely tune performance becomes a challenging task.

This challenge is exacerbated by the need of many service providers to meet multiple performance objectives. For example, reducing the tail latencies of on-line services has received much attention (e.g., [17, 23, 45]). However, techniques and configuration parameter values for reducing tail latencies can often negatively impact performance at other percentiles in the performance cumulative distribution function (CDF). Figure 1 shows an example of one such trade-off in a Web server, where setting the configuration parameter to a small value (i.e., value = 0.1 leading to the purple CDF) can significantly reduce the tail but gives much worse performance for a large part of the space (difference in the purple and blue CDFs between the 15th to 60th percentiles).

Thus, administrators must often consider multiple points on the performance CDF when configuring server systems.

Given the challenge of tuning system performance to meet a single performance objective (e.g., minimize median response time), it is not surprising that tuning for multiple performance targets is a challenging, error-prone, and time-consuming exercise for server system administrators. In this paper, we propose OpTune, a framework for guiding administrators to configure server systems to meet specified performance objectives. Examples of performance objectives that can be specified in OpTune include: (1) minimize the

Figure 1: Impact of a cache configuration parameter on the response time of a Web server.
average response time; (2) minimize the median response time while keeping the 99th percentile response time below a target value; (3) minimize the 99th percentile response time while keeping the median response time below a target value; and (4) find the “closest” achievable performance CDF for a specified target CDF.

OpTune assumes that administrators can identify a subset of important parameters, which it then carefully calibrates to best achieve the performance objectives. OpTune relies on a graphical representation of the system, performance instrumentation and profiling, and manipulation of performance CDFs to perform its function. The graphical representation describes the main system components and their interactions, and how they aggregate to determine overall system performance. OpTune collects transition probabilities and performance CDFs of the software components at different points in time while keeping the 99th percentile while maintaining the median response time below a target value.

In summary, our contributions are:

- Proposing and developing the OpTune framework for guiding administrators when configuring server systems to meet a set of performance objectives;
- Implementing OpTune in three diverse server systems to demonstrate its wide applicability; and
- Presenting results from a large set of case studies to show how OpTune can ease the task of performance tuning, particularly when this process involves tradeoffs between multiple points on the performance CDF (e.g., average and/or median vs. tail latencies).

2. Related Work

Full-range performance tuning has recently garnered interest. For example, recent work [46] uses a similar approach to OpTune—i.e., graphical representation of system together with composition of components’ performance CDFs—to predict the overall performance of compound Web services. Other work leverages graph representation to diagnose network problems [15, 29]. Unlike these study however, we seek to tune configuration parameters to actually achieve performance goals rather than only estimating system performance. We also implement and evaluate OpTune in three real systems.

Several additional projects have studied the problem of performance variability in different ways. For example, the real-time systems community has considered mechanisms to ensure that groups of tasks meet their deadline targets [9, 16], while the high performance computing community has studied mechanisms to remove performance variability or jitter [39]. OpTune is more flexible in that it helps administrators to shape the entire performance CDFs. Thus, it can be used to find appropriate configurations for different desired tradeoffs between performance (e.g., average response time) and performance variability (e.g., tail latencies).

Our work is also partly inspired by prior work on quality of service and resource allocation fairness studies [7, 12, 19, 22, 28], which attempt to guarantee a minimum level of performance in server and networked systems. While these approaches do use optimization techniques to suggest system configuration settings, they incentivize users to share resources, leaving other hardware idle. As pointed out in [37, 38] however, this may increase performance variability. Similarly, online services seek to guarantee that a high percentile of their requests complete within an acceptable amount of time [18, 28, 43]. Some of these works [30, 32, 33, 41, 44] specifically optimize median performance; we go beyond all this prior work, by considering full-range performance that also accounts for long-latency tails.

Implementing OpTune in three diverse server systems to
Finally, a few prior works have proposed to manage performance by adjusting system configurations, e.g., [47, 48]. These systems either had no performance target or needed to satisfy a single-point service-level agreement (e.g., 99th percentile performance lower than 100ms). OpTune differs from these efforts as it allows administrators to specify multiple performance targets.

3. OpTune Methodology

3.1 Overview

OpTune represents and tunes performance using the entire performance CDF of the system. To use OpTune, the designers of a system must build a service graph representation, where each node in the graph corresponds to the sequential execution of some code, and each directed edge represents control flow. The graph must contain a root node (where the computation starts) and one or more end nodes (where the computation ends). Each directed edge in the graph is labeled with the probability with which execution will pass from the source node to the destination node. Thus, each execution path from the root to an end node represents a potentially different performance behavior, which happens with different probability. Figure 2 shows the service graph for a Web server that serves only static content.

OpTune then works as follows (illustrated in Figure 3). OpTune begins with running the server system through a warm up phase so that subsequent profiling of the system will accurately represent steady state behaviors. After warm up, OpTune will enter a profiling phase (which should take place under a real or realistic workload). In this phase, OpTune gathers performance data (e.g., request service times) and transition probabilities for the components and edges in the service graph, respectively, as it sets the configuration parameters to different values within their defined domains.

Administrators are expected to specify the configuration parameters and the values that OpTune should explore. We assume that a server system making use of OpTune is modified to include the necessary calls into OpTune for this profiling.

Once OpTune has gathered the necessary profiling data, it can be directed to enter the configuration phase. In this phase, OpTune sets up an optimization problem to find configuration settings that will best meet a set of performance objectives. OpTune assumes that time will be divided into discrete accounting periods (e.g., 10 minutes), and that its goal is to configure the system to best meet the performance objectives within each accounting period. OpTune then further divides each accounting period into multiple equal length epochs (e.g., 2 minutes) and solves for a set of configurations, one per epoch, that together gives the optimal solution. This subdivision allows OpTune to achieve performance CDFs for an accounting period that would be impossible with a static configuration that should be used throughout the period.

To solve the optimization problem, OpTune must be able to predict the server’s performance for different configuration settings. It does this by using the service graph to compose the profiled performance of components and transition probabilities at specific configuration settings. As an example, suppose the Web server whose service graph is shown in Figure 2 has two parameters, $V_1$ and $V_2$. Further, suppose that $V_1$ only impacts $C_2$ and $V_2$ only impacts $C_3$. In this case, OpTune would profile $C_2$’s performance across different settings of $V_1$ (at a default setting of $V_2$), and $C_3$’s performance across different settings of $V_2$. Then, to predict the server’s performance for a particular configuration $V_1 = x, V_2 = y$, OpTune would compose $C_1$’s performance CDF, $C_2$’s CDF for $V_1 = x, C_3$’s CDF for $V_2 = y$, and $C_4$’s CDF.

Finally, OpTune enters the run phase. If only one configuration setting was chosen, then OpTune configures the system just once. If multiple configuration settings were chosen, then OpTune keeps track of time epochs, and reconfigures the system as appropriate at the beginning of each epoch.

Over time, system performance may deviate from the expected behavior. For example, this can happen when the characteristics of the workload changes. Thus, OpTune continuously monitors performance and compares the observed and expected performance for each accounting period. It will alert administrators and reinitiate the entire tuning process if it detects sufficiently large deviations. Note that such automatic detection of deviation may not be appropriate in some cases; e.g., in workloads with well-known diurnal patterns, but whose characteristics may change significantly throughout the day. In these cases, it is possible for OpTune to “remember” different profiling data and configurations appropriate for different periods during the day. One of our evaluation systems, a Hadoop MapReduce cluster, has some of these characteristics. The current implementation uses a mix...
In Section 3.4, we explain how we implement the composition operationally. This assumption does not always hold. The composition depends on the fact that service times at different nodes are independent (i.e., not correlated). The distribution function that describes the execution time of this pattern is given by the convolution (⊙) of the distributions of the components: \( F_{\text{par}}(t) = F_1(t) \circ F_2(t) \circ \ldots \circ F_n(t) \), which can be computed as \( F_{\text{par}}(t) = \prod_{i=1}^n F_i(t) \).

**Parallel with synchronized merge.** In this pattern (Figure 4(c)), execution passes from a node to the parallel execution of the set of following nodes, with a barrier at the end. The distribution function that describes the execution time of the set of parallel nodes is given by the product (⊙) of the distributions of the components: \( F_{\text{par}}(t) = F_1(t) \circ F_2(t) \circ \ldots \circ F_n(t) \), which can be computed as \( F_{\text{par}}(t) = \prod_{i=1}^n F_i(t) \).

**Loop.** In this pattern (Figure 4(d)), the execution loops through a number of states for a number of iterations before exiting the pattern. The distribution of the execution time of this pattern depends both on the distribution of the sub-graph within the loop, as well as the probability for the execution of another iteration vs. that of exiting the loop. The computation of this distribution is more involved, and so we refer the reader to [46] for the details. We denote this composition as: \( F_{\text{loop}}(t) = \oplus(F_1, F_2, \ldots, F_n) \) where \( F_i \) is the performance CDF of component \( C_i \) in the loop. Similar to the \( \circ \) operation, the transition probabilities are important to the computation but left out of the notation for simplicity.

**Computing the composed CDF of a graph.** Given a graph, we first compute the composed CDFs of loops; for nested loops, we start from the innermost loop and proceed outward. We then apply the remaining operations from the root (left) to the end nodes (right).

### 3.2 Performance Composition

Given the service graph of a system, we view the system’s service time as a random variable with a distribution that can be calculated if we know the transition probabilities of the edges and the distribution of the service time at each node. In general, most service graphs can be defined using a small number of patterns. We briefly discuss operations used to compose the CDFs for the basic patterns that are used to define the service graphs for the OpTune systems that we have built (Section 4).\(^1\) We refer the reader to [46] for a more detailed discussion of a similar approach for predicting the performance of composed Web services.

**Sequential.** In a sequential pattern, execution always passes from a node to the one following it as shown in Figure 4(a). The distribution function that describes the performance of both components is then given by the convolution (⊙) of the distributions of the components: \( F_{\text{seq}}(t) = F_1(t) \circ F_2(t) \), where \( F_i \) is the CDF for component \( C_i \) (equivalently \( F_{\text{seq}} = \int_0^t F_1(t-x) f_2(x) dx \), where \( f_2(x) \) is the probability density function of \( F_2(t) \)).

**Conditional.** In a conditional pattern (Figure 4(b)), execution passes from a node to one of a set of following nodes according to the probability associated with each outgoing edge. The distribution function that describes the execution time of the set of following nodes is then given by a weighted sum (\( \oplus \)) of the distributions of the components: \( F_{\text{cond}}(t) = F_1(t) \oplus F_2(t) \oplus \ldots \oplus F_n(t) \), which can be computed as \( F_{\text{cond}}(t) = \sum_{i=1}^n P_i f_i(t) dx \), where \( P_i \) is the probability for transitioning to node \( C_i \). Note that the transition probabilities are required for this operation, but are left out of the notation for simplicity.

### 3.3 Performance Decomposition

OpTune must also address the reverse problem in order to find suitable configurations. That is, given a service graph and different transition probabilities and performance CDFs for each component corresponding to different settings of configuration parameters, how can OpTune choose the appropriate configuration parameter values to best meet a set of performance objectives. Our approach is to pose this as an optimization problem and then solve it.

Specifically, administrators can specify performance objectives as a set of points on a target performance CDF \( F_\text{tar} \), and optionally constraints such as \( F_\text{tar}^{-1}(99) < 200\text{ms} \). The optimization problem is then posed as the minimization of the mean squared error (MSE) between a solution CDF \( F_s \) and the target \( F_\text{tar} \):

\[
\min \frac{1}{|P|} \sum_{p \in P} (F_\text{tar}^{-1}(p) - F_s^{-1}(p))^2
\]

subject to the specified constraints, where \( P \) is the range (percentages) specified for \( F_\text{tar} \), and \( F^{-1} \) is the inverse of \( F \) (typically called the quantile function).

Alternatively, administrators can specify performance objectives as the minimization of a function \( g() \) applied to the solution CDF \( F_s \), and optionally constraints such as:

---

\(^1\)The composition depends on the fact that service times at different nodes are independent (i.e., not correlated). This assumption does not always hold. In Section 3.4, we explain how we implement the composition operationally so that we can account for correlations when needed.

---

Figure 4: Sequential (a), conditional (b), parallel (c), and loop patterns.
\[ F_{g^{-1}}(99) < 200ms. \] In this case, the optimization problem is posed as:

\[
\min g_{p \in P}(F_{g^{-1}}(p))
\] (2)

where \( g() \) can be an arbitrary function that produces a single value and \( P \) is a set of percentages specified by administrators. Average is a commonly used function; e.g., minimize the average response time, while ensuring that the 99th percentile is less than 200ms.

As already mentioned, to solve for \( F_s \), we divide each accounting period into \( E \) epochs of equal length (e.g., 2 minutes). We assume that the offered load is stable over the accounting period; note that this typically implies that the accounting period should be relatively short (since the offered load is more likely to change over longer periods of time). We then search for a performance CDF \( F_e \) for each epoch \( e \) such that:

\[
F_e(t) = \frac{1}{E} \sum_{c=1}^{E} F_c(t)
\] (3)

where each \( F_e \) is the system performance CDF given by a particular setting of configuration parameters \( C_e \). Each of the CDFs \( F_e \) is computed by composing the component CDFs corresponding to \( C_e \) using the system service graph as explained in Section 3.2. The goal of course is to find the set of \( F_e \) that leads to an overall \( F_s \) giving the minimum for the posed optimization problem.

### 3.4 Implementing OpTune

We have built a prototype OpTune framework that can be integrated with different server systems. Some relevant aspects of the implementation are as follows.

**Configuration parameters independence.** Our approach of profiling components’ performance at different settings of their configuration parameters, and then composing components’ CDFs to predict overall service performance, is most efficient if the impact of parameters is localized and relatively independent. As an example, consider a system with \( n \) components \( C_1, C_2, \ldots, C_n \), and \( m \) parameters \( p_1, p_2, \ldots, p_m \). If all \( m \) parameters significantly impact all transition probabilities and/or the performance of all components, then OpTune would need to profile the system for \( O(\prod_{i=1}^{m} v_i) \) different configurations, where \( v_i \) is the number of values that \( p_i \) can take on. On the other hand, if each parameter mostly impacts the transition probabilities and performance of a non-overlapping sub-graph of the service graph, then OpTune would need to profile the system for only \( O(\sum_{i=1}^{m} v_i) \).

Currently, OpTune relies on administrators to identify the edges and components that are significantly impacted by each parameter. It uses these relationships to minimize the number of collected profiles. In future work, we will explore techniques for automatically detecting these relationships.

**Composing CDFs.** Section 3.2 describes a mathematic for manipulating CDFs that is convenient for discussing how the transition probabilities and performance CDFs of components can be composed to predict the overall service performance. However, our implementation uses a sampling approach to implement the composition operations. This approach works as follows. When profiling the performance of a component, OpTune records a large number of execution times across a large number of a component’s execution. This defines the component’s performance CDF at a particular configuration setting. Then, to compute \( F_1 \circ F_2 \), we would repeatedly compute a value of the resulting CDF by adding two values \( v_1 \) and \( v_2 \), chosen randomly from the set of execution times comprising \( F_1 \) and \( F_2 \), respectively. Other operations are implemented in a similar manner. Our profiling runs need to track the probabilities for different numbers of iterations across loop executions to support this implementation.

The above approach allows us to account for correlation between the performance of different components. For example, supposed the execution times of two components \( C_1 \) and \( C_2 \) are correlated. Then, when computing \( F_1 \circ F_2 \) (\( F_1 \) is \( C_1 \)’s performance CDF), we will choose \( v_1 \) and \( v_2 \) in a manner that respects the correlation. This was not needed in any of the three systems we implemented. When composing to predict the performance of a service, we ensure that there are enough sampling points so that compositions are always statistically significant [27, 34].

**Solving the OpTune optimization problem.** Currently, our implementation performs a complete search over the possible configuration settings to find the best solution to the optimization problem. This approach works well for a small number of configuration parameters; for example, solving the optimization problem for the three server systems that we implemented and evaluated, each with two parameters exposed to OpTune, always took less than 40 seconds. In the future, we intend to explore a more scalable approach based on a search algorithm (e.g., simulated annealing).

**User interface.** It is possible that no solution exists that satisfy the constraints specified by administrators, or that the best fitting solution is very different from the target CDF. In this case, administrators can iterate through multiple runs of OpTune, modifying the performance objectives. To ease this task, OpTune can show different CDFs that can be achieved using different configuration settings.

### 4. OpTune Systems

We have built three OpTune systems using the above framework, including a Web server, a filesystem emulator, and a MapReduce scheduler. We describe the design and implementation of these systems in this section.
4.1 Web Server

We have modified the Mongoose Web server [2] to work with OpTune. The primary performance metric for this server is request processing time. Figure 2 shows the service graph for the Web server comprising four components. We modified Mongoose to measure the execution time of the code corresponding to these components. The Web server’s CDF of request processing time \( F_{WS} \) can then be computed as: \( F_{WS} = F_1 \oplus (F_2 \odot F_3) \oplus F_4 \), where \( F_i \) is the performance CDF of component \( C_i \).

Two configuration parameters are exposed to OpTune for performance tuning. As shall be seen, we choose these two parameters because their settings can strongly shape the entire performance CDF of the server. The first is the \( \text{idle}_{-\text{time}} \) parameter in the disk I/O layer (inside the Linux kernel). This parameter specifies the length of time that the CFQ I/O scheduler will wait for another request from a thread it is currently servicing before switching to servicing I/O requests from another thread. It has been observed that in many cases, a stream of I/O requests from a single thread will correspond to sequential accesses to files [31]. This is especially true for a Web server. Thus, increasing \( \text{idle}_{-\text{time}} \) can reduce disk head movement, improving both throughput and average response time [6, 24]. However, it can increase the tail response time if a request is delayed while the I/O scheduler switches through several other threads. As shall be shown below, this parameter’s setting can have a significant impact on the Web server’s performance CDF.

The second configuration parameter is in the caching subsystem. Caching inside Web servers and proxies have been studied extensively [3, 10, 11, 14], and it has been shown that accounting for factors such as temporal locality, popularity, and size is important for maximizing performance. However, accounting for these factors represent tradeoffs. For example, in some cases, it may be desirable to achieve the highest hit rate, since this corresponds to the highest percentage of clients experiencing low response time. On the other hand, high hit rates disproportionately favors the caching of small files. Thus, in other cases, it may be more appropriate to maximize the byte hit ratio, which trades off more misses for small files to get the benefits from caching larger files.

In our Web server, we use the GDSF algorithm, which has been shown to work well [10, 14]. This replacement algorithm considers three different metrics for choosing victims for eviction when there is a miss and the cache is full. These metrics include aging based on time of last access, frequency of access, and object size. Briefly, GDSF works as follows. Each cached file is assigned a priority \( P(f) \) computed as:

\[
P(f) = \text{clock} + \text{Freq}(f) \frac{\text{Cost}(f)}{\text{Size}(f)}
\]

Figure 5: File server composition graph. \( C_1 \): process file request, \( C_2 \): get requested block from buffer cache, \( C_3 \), service missed block from disk, \( C_4 \): wait for all disk accesses to complete, complete processing of request, and return.

When there is a miss, and the cache does not have enough free space to cache the referenced file, the set of files with the lowest priorities are evicted to make space. When a file is first brought into the cache, \( \text{Freq}(f) = 1 \) and \( \text{Size}(f) \) is a function of \( f \)’s size in bytes. Whenever there is a hit for \( f \) in the cache, \( \text{Freq}(f) = \text{Freq}(f) + 1 \), and \( f \)’s priority is recomputed. Whenever a set of files \( f_1, f_2, \ldots, f_n \) are evicted, \( \text{clock} = \max_{i=1}^{n} P(f_i) \).

There are actually four possible configuration parameters for tuning GDSF’s performance, the function for increasing clock, the \( \text{Size}(f) \) function, the \( \text{Cost}(f) \) function, and the function for increasing \( \text{Freq}(f) \). For simplicity, we focus on \( \text{Size}(f) \) as the performance tuning knob because we find that it gives the largest trade-off between performance and variability; others have found size to be a critically important parameter as well, e.g., [4].

4.2 Filesystem Emulator

We have also integrated OpTune into a filesystem emulator. This emulator takes a trace of I/O requests and a file-to-disk-block mapping, and emulates the servicing of the I/O requests. The emulator emulates the operation of a filesystem by implementing a block-based buffer cache on top of a disk partition. It accesses the disk for cache misses using direct I/O to bypass the OS buffer cache. It services each I/O request by computing the set of blocks needed using the file-to-disk-block mapping, and then retrieve the blocks from the buffer cache or disk as appropriate. Writes are buffered in the buffer cache, and written back to disk by a background flusher thread. Note that the Linux kernel writes back dirty blocks based on several conditions (current load, flush threshold, etc.). For simplicity, our implementation flushes dirty blocks periodically (every 30 seconds) or when the number of dirty block reaches a high watermark.
Finally, we have integrated OpTune with the Hadoop MapReduce scheduler to explore its behavior in a system that is drastically different from the Web and filesystem servers. In this system, we use OpTune to tune the CDF of job completion times. Figure 6 shows the service graph for completing a Hadoop MapReduce job. Each job has two phases, a Map phase where all map tasks \((C_{M1}, C_{M2}, \ldots, C_{Mm})\) are executed and a Reduce phase where all reduce tasks \((C_{R1}, C_{R2}, \ldots, C_{Rn})\) are executed. The job is initiated in \(C_s\). \(C_b\) transitions between the Map and Reduce phases, and \(C_e\) saves the output and completes the job.

Given this service graph, the performance CDF of the system \((F_{MR})\) can be computed as:

\[
F_{MR} = F_s \oplus (F_{M1} \otimes \ldots \otimes F_{Mm}) \oplus F_b \oplus (F_{R1} \otimes \ldots \otimes F_{Rn}) \oplus F_e
\]

We tune the performance of this system by adjusting the scheduling policy and dropping the execution of a subset of map tasks. For the scheduling policy, we implemented a parameter \(\text{prob}_{\text{SJF}}\) that moves the scheduling policy between FIFO, which gives better fairness since jobs are executed in order of arrival, and Shortest-Job-First (SJF), which reduces average waiting time but may starve large jobs. \(\text{prob}_{\text{SJF}}\) allows a mix of the two scheduling policies, allowing the administrator to favor one over the other by sliding the parameter. Note that Hadoop has been specifically implemented to allow configuration with different pluggable schedulers. In this case, we are introducing/studying a scheduler that allows dynamic tuning, rather than the static a priori selection of a single scheduler.

We implement the mixed scheduling policy using Hadoop’s job priorities 1-5, with 1 being lowest and 5 being highest. When a job arrives, we randomly determine whether FIFO or SJF should be used based on \(\text{prob}_{\text{SJF}}\). If FIFO, then the job is given priority 3. If SJF, then the job is given a priority based on the number of reduce tasks (which we use as a rough estimate of job size). The partitioning between priorities is such that jobs with sizes around the median are given priority 3, the largest priority 1, and the smallest priority 5.

The second configuration parameter \(\text{drop}_{\text{p}}\text{maps}\) allows the administrator to trade precision for reduced completion times. In particular, this parameter controls whether map tasks can be dropped from the execution of a MapReduce job. When non-zero, \(\text{drop}_{\text{p}}\text{maps}\) percent of map tasks are randomly chosen and dropped from the execution of each job. While we are introducing this parameter, we note that dropping tasks has been used to enable approximations in MapReduce with small inaccuracy bounds, as shown in [21, 40]. Thus, we hypothesize that parameters for controlling approximation similar to \(\text{drop}_{\text{p}}\text{maps}\) will be introduced into future approximation-enabled MapReduce frameworks.

### 5. Evaluation

We now turn to explore and evaluate OpTune’s efficacy at helping administrators to tune their systems to achieve specific performance goals. We present mostly results from the Web server, although we also show some results for the MapReduce system and filesystem emulator toward the end of the section.
5.1 Experimental Setup

Experimental platform. Experiments for the Web and filesystem servers were run on a server machine equipped with a 2.4 GHz 4-core, each with 2 hyper-threads, Xeon CPU, 8 GB of RAM, and a 160 GB 7200 RPM SATA hard disk. The server was running Linux 3.2.54, with the scheduling policy of the disk I/O subsystem set to Completely Fair Queuing (CFQ).

Experiments for the MapReduce system were run on a 10-machine cluster, where each machine is equipped with a 1.8 GHz 2-core, each with 2 hyper-threads, Opteron CPU, 8 GB of RAM, and a 750 GB 7200 RPM SATA hard disk. The servers were running Hadoop 1.1.2 on top of Linux 2.6.18.

Web server workload. We use ProWGen [8] to generate a Web access trace. We use the default Zipf distribution with parameter 0.9 and Pareto distribution with tail index of 1.2 to model object popularity and object size, respectively [8]. The median object size is set to 60 KB with standard deviation of 10 MB based on studies of previous Web server workloads [20, 25, 36]. Finally, requests arrive according to a Poisson process with mean inter-arrival time of 72ms, leading to an average utilization of approximately 50%. We generate a trace lasting 4 hours, using the first 2 hours for profiling and system warmup and the last 2 hours for our experiments.

Filesystem workload. We use a trace from the Microsoft Production Server Traces [26, 42]. Traces in this set were collected from a number of different Microsoft production servers, and include information such as process ID, operation type, file descriptor, offset, and size. These traces contain sufficient information for us to build the needed file-to-disk-block mapping. We use the 6-hour MSN Storage File Server trace to study the behavior of typical file servers. We use the first 4 hours for profiling and system warmup and the last 2 hours for our experiments. The trace was collected from a more powerful server than our, so we slowed the trace down such that average throughput is approximately 60% of saturation.

MapReduce workload. We use the Statistical Workload Injector for MapReduce (SWIM) [13] to generate a scaled-down 6-hour workload from a larger Facebook trace collected from May to October 2009. In the resulting workload, each job comprises 2-120 map tasks and 1-20 reduce tasks. There are ~700 jobs with ~8000 tasks. The map phase of each job takes 50-300 seconds, and the reduce phase takes 15-100 seconds. Jobs have inputs of 64MB-9GB and outputs of up to 1GB. Figure 7 plots this workload, which gives an average cluster utilization of 64%. We use the first 3 hours for profiling and the last 3 hours for our experiments.

5.2 Impact of Configuration Parameters

Impact of caching size priority parameter. We begin our study by exploring the impact that different values of configuration parameters can have on the performance of a system. Figure 8 plots the performance CDFs for the Web server when we set the I/O idle-time parameter to 8ms and the function \( Size(f) \) to \(|f|^\beta, |f|^{1/2}, |f|, |f|^2\), where \(|f| = \text{number of bytes in } f\).\(^3\) These functions lead to 18%, 24%, 41% and 44% item-wise hit-ratios, respectively. It is easy to observe that lowering the caching priority of larger objects (e.g., \( Size(f) = |f|^2 \)) can substantially improve response time for a fraction of the requests (differences in CDFs from ~15-45%). This is because larger objects will push out smaller objects less frequently, leading to more effective caching for the smaller objects. On the other hand, favoring the smaller objects can significantly increase the tail response time, as requests for the largest objects will most likely lead to cache misses.

Impact of I/O idle-time parameter. We next explore the impact of the I/O idle-time parameter on server performance. Specifically, we set the caching \( Size(f) \) parameter to \(|f|\), which gives a hit ratio of 41%, and we set the idle-time \( (I) \) in

\( ^3\) We also adopt the common approach of not caching files larger than 2MB to avoid polluting the cache with very large objects [1, 3, 11].
CFQ to 0ms, 4ms, 8ms, 16ms, and 24ms. Figure 9 plots the results. Again, it is easy to observe that the parameter value offers tradeoffs between the response time for a significant portion of the CDF (between ~50%-90%) against the tail (>95%).

**Configuration parameters independence.** We specify to OpTune that \( \text{Size}(f) \) affect \( P_{\text{hit}}, P_{\text{miss}} \), and the performance of \( C_2 \), while \( \text{idle\_time} \) affect the performance of \( C_3 \). In Figure 10, we study the accuracy of this information. Specifically, Figure 10(a) and (b) show that \( C_2 \)'s performance CDF remains almost the same even when \( \text{idle\_time} \) is set to two very different values. Thus, \( \text{idle\_time} \) indeed does not impact \( C_2 \)'s performance. On the other hand, Figure 10(c) and (d) show that the tail of \( C_3 \)'s performance CDF is affected somewhat by \( \text{Size}(f) \) when \( \text{idle\_time} = 24\text{ms} \). We deemed the inaccuracies introduced to be small enough that it was acceptable to assume \( C_3 \) is independent of \( \text{Size}(f) \) to keep profiling overheads low.

Figure 11 shows an example of the potential inaccuracies arising from OpTune’s various assumptions. The figure shows a target CDF chosen to highlight inaccuracies introduced by the specified independence assumptions (the long tail), the CDF predicted by OpTune for its chosen configuration(s), and the resulting observed CDF when the server was run with OpTune’s chosen configuration(s). Observe that while there are some inaccuracies, the fit between predicted and actual is quite good.

**5.3 Performance Tuning**

**Single point performance optimization.** We begin by studying the impact of optimizing for a single point on the CDF. Figure 12a shows this single point optimization when OpTune optimizes for the smallest average, median, 90\(^{\text{th}}\) percentile, and 99\(^{\text{th}}\) percentile. Table 1 lists the average, median, 90\(^{\text{th}}\) percentile, and 99\(^{\text{th}}\) percentile response times for
each of these optimization goal. Observe that depending on whether the user is more concerned with average/median performance or the tail, the overall performance CDF differs significantly. Specifically, when we are interested in minimizing the 99th percentile, OpTune sets \( \text{Size}(f) \) to \(|f|^2\) and \( \text{idle} \) – \( \text{time} \) to 0. These settings lead to the smallest 99th percentile response time (46ms), but degrades performance significantly for lower percentiles (e.g., median response time of 11.3ms). In contrast, optimizing the median leads OpTune to set \( \text{Size}(f) \) to \(|f|^2\) and \( \text{idle} \) – \( \text{time} \) to 24ms. These parameters lead to a much lower median response time (0.1ms), but significantly degrades the 99th percentile response time (281ms). Minimizing the average leads to longer response time for shorter requests (median time of 10.4ms), but significantly smaller 99th percentile time (102.2ms).

<table>
<thead>
<tr>
<th>Goal</th>
<th>Avg.</th>
<th>Median</th>
<th>90th-%ile</th>
<th>99th-%ile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min Avg.</td>
<td>12.1</td>
<td>10.4</td>
<td>24.2</td>
<td>102.2</td>
</tr>
<tr>
<td>Min Median</td>
<td>22.4</td>
<td>0.1</td>
<td>49.6</td>
<td>281.3</td>
</tr>
<tr>
<td>Min 90th-%ile</td>
<td>13.4</td>
<td>3.2</td>
<td>19.5</td>
<td>143.7</td>
</tr>
<tr>
<td>Min 99th-%ile</td>
<td>13.9</td>
<td>11.3</td>
<td>30.1</td>
<td>46.0</td>
</tr>
</tbody>
</table>

Table 1: Detailed results for web server single point optimization. Times are given in ms.

Multi-point performance optimization. As discussed previously, it is frequently not desirable to optimize for a single point of performance. Rather, the user may want to realize multiple performance goals, such as minimizing the 99th percentile response time, while maintaining a target median response time. Meeting such performance goals is exactly what we set out to do with OpTune.

Figure 12b shows the results when the user wants to minimize the 99th percentile response time while constraining median response time to be no worse than 10ms and 5ms. Observe that as the bound for the median response time becomes tighter (10ms \( \rightarrow \) 5ms), OpTune has to trade off a progressively “longer” tail for the desired median performance.

Figure 12c shows that OpTune is also effective when the constraint and optimization goal are exchanged; that is, in these cases, OpTune is seeking to minimize the median performance while observing a constraint for the 99th percentile.

Figure 13a shows the results when minimizing the 99th percentile performance with bounds on the median and the 90th percentile performance. Interestingly, when we introduce the bound for the 90th percentile performance, the tail becomes much worse, while the median becomes much better compared to when we only bound the median. This demonstrates OpTune’s ability for full-range performance tuning, and the difficulty facing administrators without OpTune when performance objectives require thinking about multiple points on the performance CDF.

Figures 13b and 13c show the results when minimizing the 99th percentile performance with bounds on the median and the 90th percentile performance for the filesystem emulator and MapReduce system. Results are similar in that adding performance objectives (constraints in these cases) can lead to significant changes over the entire performance CDF. Such full-range tuning would be very difficult for administrators to manage manually.

Full performance CDF target. We previously showed results in Figure 11 for OpTune seeking to meet an achievable performance objectives specified as a full CDF curve (100 points). Next, we show the behavior of OpTune when user’s performance requirement is hard to achieve.

Figure 14 plots the target CDF that the user provides and two closest CDFs that OpTune can achieve. Figure 14a shows the target CDF requires the 80th percentile to be below 4ms. However, the predicted CDF only achieves \(~20\)ms. Figure 14b plots the QQ-plot to illustrate the difference between the target CDF and predicted CDFs.

As described in Section 3.4, it is possible that no solution exists that satisfy the constraints specified by administrators, or that the best fitting solution is very different from the target CDF. In this case, administrators can iterate through multiple runs of OpTune, modifying the performance objectives.
5.4 Sensitivity Analysis
We have studied the sensitivity of our results for the Web server to different workload characteristics, including load intensity, correlation between object size and popularity, and the distribution of object sizes. As these characteristics change, the tradeoffs embodied in the configuration parameters can increase or decrease. For example, Figure 15 shows that a higher load intensity can significantly increase the tradeoffs given by different settings of idle_time. Correspondingly, results for a lower load intensity (not shown here) shows less tradeoffs. Overall, we find that the parameters exposed to OpTune for the three systems continue to embody significant tradeoffs across a wide range of different workload characteristics. Thus, we conclude that OpTune should be widely applicable to full-range performance tuning of many server systems and many different workloads.

6. Conclusions
In this paper, we proposed and evaluated OpTune, a framework for helping administrators to configure server systems to best achieve a set of performance objectives. Administrators can use OpTune to find settings for multiple interacting configuration parameters in order to shape the entire performance CDF of a server system. Administrators can also use OpTune to ask what-if questions. For example, what will happen to the performance CDF of a system if its configuration is modified to meet an additional constraint such as “make the 90th percentile response time less than X.” Such tasks are extremely difficult to perform manually, and will only become more difficult as server systems becomes ever more complex.

We have integrated OpTune into three different server systems: a Web server, a file system emulator, and a MapReduce scheduler. Our evaluation results show that configuration parameters embody significant tradeoffs between different parts of the performance CDF—e.g., configuring to reduce tail response times can significantly worsen response times for shorter requests—and that OpTune is a powerful tool for helping administrators explore such tradeoffs and configure server systems appropriately for performance objectives driven by different needs.

References


