Automatic Configuration of Internet Services

Wei Zheng
Dept. of Computer Science
Rutgers University
wzheng@cs.rutgers.edu

Ricardo Bianchini
Dept. of Computer Science
Rutgers University
ricardob@cs.rutgers.edu

Thu D. Nguyen
Dept. of Computer Science
Rutgers University
tdnguyen@cs.rutgers.edu

ABSTRACT

Recent research has found that operators frequently misconfigure Internet services, causing various availability and performance problems. In this paper, we propose a software infrastructure that eliminates several types of misconfiguration by automating the generation of configuration files in Internet services, even as the services evolve. The infrastructure comprises a custom scripting language, configuration file templates, communicating runtime monitors, and heuristic algorithms to detect dependencies between configuration parameters and select ideal configurations. To demonstrate our infrastructure experimentally, we apply it to a realistic online auction service. Our results show that the infrastructure can simplify operation significantly while eliminating 58% of the misconfigurations found in a previous study of the same service. Furthermore, our results show that the infrastructure can efficiently determine the configuration parameters that lead to high performance as the service evolves through a hardware upgrade and the scheduled maintenance of a few nodes.

Categories and Subject Descriptors

K.6 [**Management of computing and information systems**]: System management

General Terms

Algorithms, experimentation, performance

Keywords

Configuration, manageability, operator mistakes, Internet services

1. INTRODUCTION

Recent research has found that operator mistakes are a significant source of availability, performance, and security problems in cluster-based Internet services [13, 16]. Moreover, these studies found that misconfigurations are the most common type of operator mistake. For example, Oppenheimer and Patterson [16] studied three commercial Internet services and found that more than 50% of the operator mistakes that led to service unavailability were misconfigurations. As another example, Nagaraja *et al.* [13] asked 21 volunteer operators to perform different management tasks on a three-tier online auction service. During 43 experiments with these volunteers, they found that misconfigurations were the most common type of operator mistake, occurring 24 times out of a total of 42 mistakes. Even expert operators misconfigured the service in the experiments.

Nagaraja *et al.* detailed the misconfigurations that they observed. For example, some operators misconfigured first-tier servers when a new second-tier server was added to the service, misconfigured second-tier servers after upgrading the database machine, misconfigured an upgraded first-tier server, and improperly assigned root-user passwords in database configuration files. Although the operators did not have access to the supporting software that is available to the operators of commercial services, we believe that these types of misconfiguration also occur in commercial services. The reason is that, even in these services, software configuration is often done manually or semi-automatically for one or a few servers and then deployed widely to the remaining servers [17], as in [6, 11, 15].

In essence, misconfigurations occur frequently for two main reasons. First, multi-tier services and the servers that implement them are becoming increasingly complex. For example, services consisting of Web, application, and database tiers require operators to master the operation of three different classes of servers. Mastering even one of these classes may be a challenge, as illustrated by the Apache Web server. The main configuration file for Apache has 240+ uncommented lines setting parameters that relate to performance, support files, service structure, and required modules, among other things. Second, operators have to manually modify the servers' configuration files often over the service's lifetime. In particular, services constantly evolve through software and hardware upgrades, as well as node additions and removals. Again taking Apache as an example, its configuration files have to be changed every time there is a node addition or removal in the second tier.

In this paper, we propose a software infrastructure that can eliminate a wide range of misconfigurations by automating the configuration of cluster-based Internet services. In particular, we focus on the generation of configuration files for the servers that implement the service. The infrastructure is motivated by three key observations we have made in managing prototype services for our research: (1) the vast majority of the configuration of each server does not change over the lifetime of the service; (2) some of the most tedious and mistake-prone operator activities involve reconfiguring the service as a result of node additions and removals (or non-transient failures); and (3) changes made to a configuration parameter may affect the best value for only a few of the other pa-

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

rameters. Based on observation (1), we create configuration file templates that are similar to the configuration files themselves and specify how to perform the small set of changes that they require. Based on observation (2), we include a network of membership daemons that initiates reconfigurations. Based on observation (3), we introduce the explicit representation of relationships between configuration parameters.

More specifically, the infrastructure comprises a custom scripting language, configuration file templates, per-node communicating daemons, and heuristic algorithms to detect dependencies between configuration parameters and select ideal configurations. Using the scripting language, the service designer can write *configuration scripts* that procedurally specify how the templates should be modified to generate each configuration file. The daemons collect information about the membership of the different service tiers and can readily provide this information to the configuration scripts. The automatic configuration is started when the daemons detect changes to the membership (or when the operator requests it explicitly). The daemons also interpret the scripts to effect the changes to the configuration files. Finally, one of our heuristic algorithms defines a *parameter dependency graph* for the service. During the automatic generation of files, this graph is used by another heuristic algorithm to explore the space of parameter values, seeking the best possible configuration. Multiple criteria, such as performance, availability, or performability, can be used to determine the best configuration.

We have developed a prototype implementation of the infrastructure, including configuration scripts and templates for Apache, Tomcat, and MySQL. Our evaluation uses the online auction service studied by Nagaraja *et al.* in [13]. Using their operator action logs and a previously proposed approach to quantifying configuration complexity [4], we find that our infrastructure can simplify the service operation significantly while eliminating 58% of the misconfigurations found in [13]. Furthermore, our results show that the infrastructure can efficiently determine the configuration parameters that lead to high performance, as the service evolves through a hardware upgrade and the scheduled maintenance (i.e., removal) of a few nodes, by leveraging the parameter dependency graph.

Other infrastructures for automatic configuration have been proposed, e.g. [1, 2, 5]. However, they focus on configuring the operating system and installing the proper user-level software on each node; the configuration of the user-level server software itself is not addressed. Furthermore, none of the previous infrastructures explicitly represents or leverages the dependencies between configuration parameters. Thus, our infrastructure can be combined with these previous systems to produce services that are manageable and efficient.

We conclude that our infrastructure can be useful for both existing and new services, as it reduces complexity, eliminates operator mistakes, and can express a range of configuration parameter relationships and optimize the finding of good configurations.

In summary, our main contributions are:

- We propose an infrastructure for generating configuration files automatically. The most novel aspect of the infrastructure is the notion of a parameter dependency graph. We propose a heuristic algorithm for creating the graph and a heuristic algorithm that uses the graph to optimize the service configuration; and
- We implement and quantitatively evaluate our infrastructure and algorithms for a realistic service.

The remainder of this paper is organized as follows. The next section overviews the most important related work. Sections 3 and 4 describe our infrastructure in detail. Section 5 describes our experimental setup and discusses our main results. Section 6 summarizes our findings and draws our conclusions.

2. RELATED WORK

We make contributions related to configuration parameter relationships, strategies for configuration management, and performance tuning, all in the context of Internet services. We believe that our work is the first to represent and leverage the relationships between parameters explicitly. Next, we discuss the related works in configuration management and performance tuning.

2.1 Configuration Management

Most previous works in configuration management seek to automatically detect and correct misconfigurations. Validation [13] detects many classes of operator mistakes, including misconfigurations, in Internet services. The Chronus system [22] seeks to identify misconfigurations in stand-alone computers that go undetected for a period of time. Also targeting stand-alone computers, the Glean system [9] infers correctness constraints for configuration entries contained in the Windows Registry. Along similar lines, the Strider system [21] compares the registry of a misconfigured computer with a "healthy" registry snapshot to detect potential misconfigurations. Later, a database of known misconfigurations support the user in correcting the misconfiguration. (Interestingly, [21] finds that between 80% and 95% of the configuration of desktop machines does not change over time, which is along the same lines as our observation about Internet services.) To eliminate the need for manually selecting correct snapshots, PeerPressure [20] compares the registry of a misconfigured computer to those of a large sample of computers and uses statistical techniques to correct misconfigurations to conform to the majority of samples.

Our infrastructure differs significantly from these previous systems as it generates the proper configuration files whenever necessary, obviating the need to detect and correct misconfigurations. Furthermore, these previous systems do not reduce the amount of work required of operators to configure each machine. As we discuss in detail later, our infrastructure (almost) completely eliminates operator involvement in many common configuration tasks.

Like our infrastructure, several other systems have proposed to generate and manage configurations automatically, e.g. [1, 2, 5]. Cfengine [5] is perhaps the most widely used configuration management tool. It provides high-level language directives that describe how classes of machines should be configured in a large installation. An agent on each machine contacts a central configuration repository to collect the specific directives for the machine. With similar goals and approach, LCFG [2] holds declarative descriptions of the "aspects" of the installation's configuration. These aspects are compiled into "profiles" for each machine. A script on each machine creates the configuration based on its profile.

A key difference between our infrastructure and these systems is that they do not view the machines in a cluster as forming a single service. As a result, they make it hard or impossible to represent configuration changes that are prompted by remote components of the service. Furthermore, these systems provide support for machine installation and startup, whereas our infrastructure assumes that machines are already running the operating system and focuses solely on service management and evolution. In these respects, our infrastructure is more closely related to SmartFrog [1].

Like Cfengine, LCFG, and our infrastructure, SmartFrog also includes a custom interpreted language. The language is used to describe, in a declarative manner, the software components that form the distributed application or service, their configuration parame-

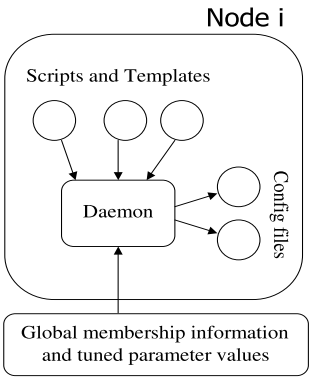

Figure 1: Overview of each node.

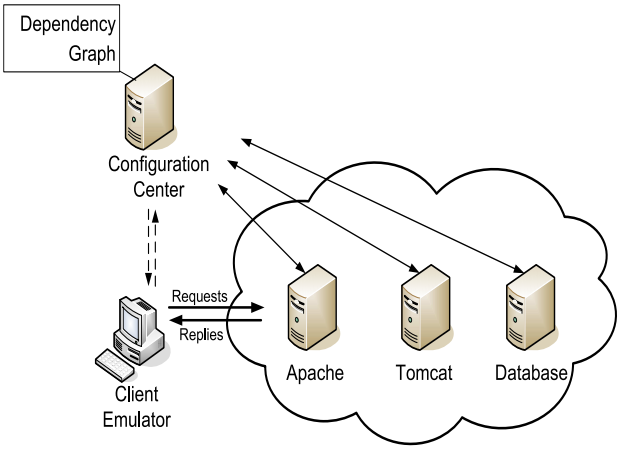

Figure 2: Overview of the entire system. For simplicity, we only show one server node per service tier.

ters, and how they should connect to each other. SmartFrog also includes a runtime system to activate and maintain the desired configuration, and a component model defining the interfaces that each component should implement to allow SmartFrog to manage it.

A basic difference between SmartFrog and our infrastructure is that we rely heavily on templates that are similar to the actual configuration files, using procedural generation scripts to modify these templates. This difference is largely philosophical; we feel that it is easier to describe (changes to) configurations in a procedural manner, as in regular systems programming. More importantly, we quantitatively evaluate the manageability benefits of our infrastructure for a realistic service. We are not aware of any quantitative evaluation of SmartFrog. Furthermore, our infrastructure generates configurations that maximize a specific metric, relying on explicit configuration parameter relationships. SmartFrog does not represent or leverage such relationships.

2.2 Performance Tuning

Several papers, e.g. [7, 8, 19], have considered performance tuning of Internet services. Diao *et al.* [8] studied the performance tuning of a Web server, using an agent-based feedback control system. Chung and Hollingsworth [7] considered a three-tiered service and demonstrated that no single assignment of machines to tiers performs well for all types of workloads. Using the Simplex algorithm [14], they find the ideal machine-tier assignment for each workload. Again considering multi-tier services, Stewart and Shen [19] developed profile-based models to predict service throughput and response time. They also explored how to accomplish typical system management tasks, such as placing software components for high performance, using their models and offline Simulated Annealing [10].

Compared to these and other previous approaches to performance tuning of Internet services, our infrastructure has two key distinguishing features: it starts the tuning process as a result of service changes that require the regeneration of configuration files, and it reduces the search space by creating and leveraging a parameter dependency graph.

3. OUR INFRASTRUCTURE

Overview. The key goal of our infrastructure is to obviate the need for the operator to specify values for the configuration parameters that should change dynamically. A good example of such a parameter is the list of application servers that a Web server may contact to service requests for dynamic content. Another is the maximum number of worker threads that a Tomcat server should use, which may depend on the hardware configuration of the machine hosting the Tomcat server. The maximum number of worker threads is also an example of a tunable numeric parameter. Rather than forcing the operator to manually maintain these parameters as the service evolves, we have designed a simple infrastructure for determining values for these parameters dynamically and inserting them into the appropriate configuration files at their appropriate places.

As shown in Figures 1 and 2, our infrastructure has four main components: (1) a set of configuration file templates, which are similar to the configuration files themselves, but include place-holders for dynamic parameter values; (2) a simple language that can be used to write configuration scripts (or simply scripts) for accessing the runtime information and formatting it for insertion into configuration files, according to the templates; (3) per-node daemons that monitor the service and regenerate configuration files as needed by interpreting the scripts; and (4) a configuration center that directs the tuning of parameter values and disseminates copies of the templates and scripts to machines that are added to the service. The client emulator exercises the service during the tuning process.

To tune parameters without conflicting with actual user loads, our infrastructure assumes that the service is provided by at least two independent data centers. (Note that real services always comprise multiple independent and geographically distributed data centers to guarantee high availability.) During periods of light overall load, the data center to be tuned can be dedicated to the tuning process by redirecting its load to the other data center(s).

The process of automatically configuring the service works as follows. At service-installation time, the service designer creates the configuration file templates and the scripts for transforming the templates into the actual configuration files. At run time, each per-node daemon monitors the state of its node, including what service component is running on that node, and periodically broadcasts this information so that all daemons have a complete picture of the service. The automatic configuration is started when the daemons detect changes to the service membership, i.e. a node is added/removed or fails, or the operator requests it explicitly (discussed below). During automatic configuration, the daemons interpret the scripts to effect the changes to the configuration files. Because parameter tuning may take long to complete, the daemons continue using the current values for the tunable parameters but

```
1. # File worker.properties.tmp
2.
3. # List workers by name
4. worker.list = [WORKER_LIST]loadbalancer
5.
6. # Describe properties of each worker
7. [WORKER_PROPERTIES]
8.
9. # Describe properties of load balancer
10. worker.loadbalancer.type=lb
11. worker.loadbalancer.balanced_workers=[WORKERS]
```

Figure 3: Template for worker.properties file.

inform the configuration center that the tuning process should be started at the next opportunity, i.e. the next time the service enters a period of light load. When the configuration files have been generated, they are installed by each local daemon, which restarts its local server. When the opportunity arises to tune parameters, the configuration center starts the execution of the heuristic algorithms and eventually provides the best values to the local daemons. At that point, the local daemons again generate their configuration files and restart their servers.

Thus, throughout the lifetime of the service, the participation of the service operator can be dramatically reduced with our infrastructure. However, the operator remains responsible for physically changing the service, e.g. by replacing broken disk drives or adding more network bandwidth to the system, and starting the automatic configuration explicitly as a result of application-level evolution, e.g. a new database schema is defined or new service features are added. Note that such application-level evolutions are not detectable by an external, operating system-level monitoring infrastructure like ours.

Templates and scripts. The generation of each configuration file requires two components: (1) a template containing the values for the static configuration parameters and macro names in place of the values for the dynamic configuration parameters, and (2) a script for generating the text that will be substituted in place of the appropriate macro names.

To discuss these components concretely, consider figures 3 and 4. Figure 3 shows a very simple example template for the worker.properties file used by the `mod_jk` module of the Apache Web server. Worker.properties describes the names and properties of the servers responsible for generating dynamic content. In the template, [WORKER_LIST], [WORKER_PROPERTIES] and [WORKERS] are macros to be defined dynamically by the per-node daemons, as the service evolves.

Figure 4 shows an example script for generating the actual worker.properties file. Like all scripts, this example has two sections, a global section and a program section. The global section defines the name of the template, as well as useful constants such as the comment character. The program section contains instructions for formatting the information retrieved from the runtime system for the configuration file.

The programming language for this scripting is quite simple and has only a few features: macros, strings, variables, control statements (if-then and for loops), and output to files. Macros are named by a string inside a pair of square brackets, e.g. [WORKER_LIST]. Variables are named by a $ character followed by a string. All variables are of type string and are allocated and assigned the null string on first reference. A few variables have pre-defined meanings, such as the variable $TEMPLATE which should store the name of the template to use in generating the configuration file. The macros

```
1. # Global section
2.
3. $TEMPLATE = "worker.properties.tmp";
4. $COMMENT_CHAR = "#";
5.
6. # Program section
7.
8. # Generate list of tier-2 application servers
9. ITER($TIER2.COUNT, $index)
10. {
11.   $name1 = $name1 + "tomcat" + $index + ",";
12. };
13. [WORKER_LIST] = "worker.list=" + $name1;
14.
15. # Generate properties of each application server
16. ITER($TIER2.COUNT, $index)
17. {
18.   $port = "worker.tomcat" + $index + ".port=8009\n";
19.   $host = "worker.tomcat" + $index + ".host=" +
20.           $TIER2.NAMELIST.EACH + "\n";
21.   $lb = "worker.tomcat" + $index + ".lbfactor=1\n";
22.   $worker_props = $worker_props + $port + $host + $lb;
23. };
24. [WORKER_PROPERTIES] = $worker_props;
25.
26. # Generate properties of the load balancer
27. CHOP($name1);
28. [WORKERS] = $name1;
```

Figure 4: Script for generating worker.properties.

and some pre-defined variables hide communication with the local daemon and the configuration center. In the example, the $TIER2 pre-defined variables contain information about the tier of application servers. Specifically, $TIER2.COUNT contains the number of application servers in the tier and $TIER2.NAMELIST contains the names of nodes hosting these application servers. Values for these macros are received from the local daemon. The control statements are also illustrated in the example: lines 9–13 show the iterative construction of the list of application servers, whereas lines 16–24 show the iterative construction of the properties of each of these servers. Finally, lines 27–28 specify the properties of the load balancer. When the script terminates, an interpreter (part of the local daemon) generates the configuration file by replacing each occurrence of a macro name in the template with the string that the corresponding macro was last assigned.

Obviously, we could have simply used Perl or some other scripting language to write such scripts. The reason we opted for a new language is that we wanted to investigate whether we could effectively tailor the language to configuration tasks and integrate it tightly with the rest of the infrastructure (the local daemon and the configuration center). These characteristics actually make writing scripts in the language extremely easy and clean. Despite these benefits, we are fully aware that some users would possibly prefer a more familiar language. In fact, we may have made different design decisions if we had been building a commercial product rather than a research testbed.

Network of per-node daemons. Each daemon tracks the service components that run on its node. Periodically, it broadcasts information about these service components to its peers. For example, a daemon running on a node that hosts an application server would periodically broadcast the following record:

```
tier:      ''AppServer''
component: ''Tomcat''
ip:        the node's IP address
hostname:  node's host name
```

The periodic broadcasts also serve as heartbeat messages for a simple membership protocol. If a daemon does not receive a heart-

beat from a peer within 3 heartbeat periods, it assumes that the peer has crashed, removes it (and the service components hosted on that node) from its list of active hosts, and regenerates its configuration files accordingly. The daemon regenerates the configuration files by parsing and interpreting the corresponding scripts.

Configuration center. The configuration center controls the tuning of the numeric parameters and provides their best values to the local daemons for inclusion in the configuration files. Note that, because the configuration center needs to be involved in the generation of the configuration files and the heartbeat messages are broadcast to the entire cluster, we could have assigned the responsibility for generating the configuration files to the configuration center. However, this design could hide network partitions between servers that do not affect the communication of the servers with the configuration center. For this reason, we did not pursue it.

The next section details the parameter-tuning process.

4. TUNING PARAMETERS

One of the main goals of our infrastructure is to generate configuration files that optimize the service with respect to a pre-defined metric of interest, e.g. performance, availability, performability [12], as the system evolves. In particular, common changes that are made to services over their lifetimes, such as increasing the amount of memory per node, adding new nodes, or replacing a pair of expensive database systems (a primary and a hot backup) with multiple cheaper computers, may cause the current configuration settings to behave poorly with respect to the metric. Settings that are tailored to the changed service may be more appropriate.

To see this more clearly, suppose that we have a three-tier service comprising Web, application, and database servers. Given throughput performance as the metric, increasing the amount of memory per node may allow the servers to create more threads concurrently and, as a result, achieve higher throughput. Adding nodes at the application-server tier may force a reduction in the maximum number of concurrent threads per application server to avoid exceeding the corresponding number at the database tier. The maximum number of concurrent threads is a common configuration parameter of servers, like Apache, Tomcat, and MySQL. Given availability or performability as the metric, replacing expensive database systems with more numerous cheaper ones may require a new setting for the failover timeouts at the application-server tier.

It is difficult to know exactly what parameters have to be changed as the system evolves in different ways. It is even harder to select new, efficient values for the parameters to be changed, especially when there are dependencies between parameters. Both of these decisions are typically based on the operator's intuition and/or a few exploratory experiments. This is clearly not ideal.

A better approach is to cast these decisions as an optimization problem and have the system automatically search the parameter space, i.e. tune the parameter values, for the operator. The traditional approaches to automate this tuning process are brute-force and heuristic algorithms. The brute-force algorithm tries all possible combinations of values for all possible parameters. Clearly, it only works when the total number of parameters is extremely small. Heuristic algorithms are better in that they try to approximate the optimal value settings or simply produce very good settings, without actually going through all combinations. However, these algorithms are still inefficient because, every time they are run, they still have to consider the space formed by all parameters in the system.

Our infrastructure proposes that *the key to improving efficiency further is to constrain the search space to only the subset of parameters that are affected by the change to the service*. It is computationally intensive to define this subset, but this overhead is amortized over multiple parameter tunings in which only the affected parameters are considered.

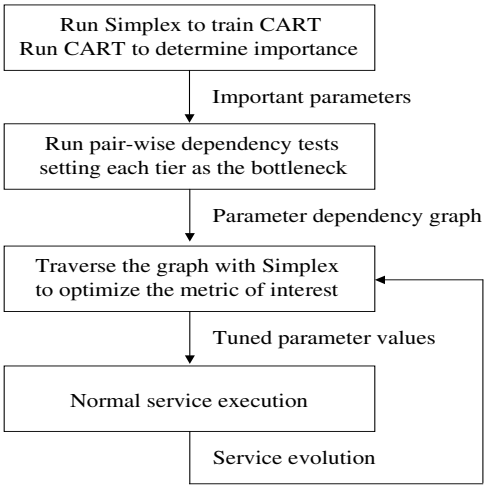

Figure 5: Steps involved in tuning parameters.

We represent the affected parameters in a *parameter dependency graph*. For each particular change made to the service, we can traverse the graph to find all the affected parameters and tune them. Our approach to defining the graph and tuning parameters is experimental. More specifically, we accomplish these tasks by exercising the service multiples times with a single representative trace but different parameter values. As mentioned above, these tasks must be performed during periods of light overall load.

Figure 5 lists the set of steps involved in defining the dependency graph and tuning parameters. In the next two subsections, we detail our heuristic algorithms to define the graph and leverage it to limit the number of parameter-tuning experiments.

4.1 Parameter Dependency Graph

As its name suggests, our parameter dependency graph represents the dependencies between parameters explicitly as a directed graph. Each vertex in the graph represents a parameter, whereas a directed edge represents a dependency between two vertices. Specifically, if a vertex B depends on another vertex A (there is an edge from A to B), the value for the parameter that corresponds to B has to be recomputed every time the value for the parameter that corresponds to A changes.

When the service evolves, the parameters corresponding to the tier directly affected by the change become "source" vertices in the graph. For example, if an application-server node is added to the service, we would make all configuration parameters of application servers source vertices. The parameters that are affected by the change are those reachable from the source vertices. The values for these reachable parameters should be tuned experimentally for each possible value of each source parameter.

The challenge at this point is to find the dependencies between parameters in an efficient manner. Next, we describe the two steps we take to do so.

First step: Finding important parameters. Our first step in determining the dependencies between parameters reduces the search space to just the "important" parameters of each service tier, i.e. those parameters that have the greatest effect on the metric of inter-

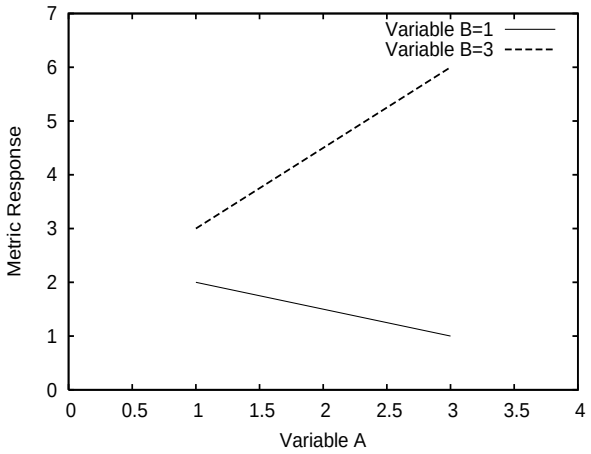

Figure 6: A depends on B.

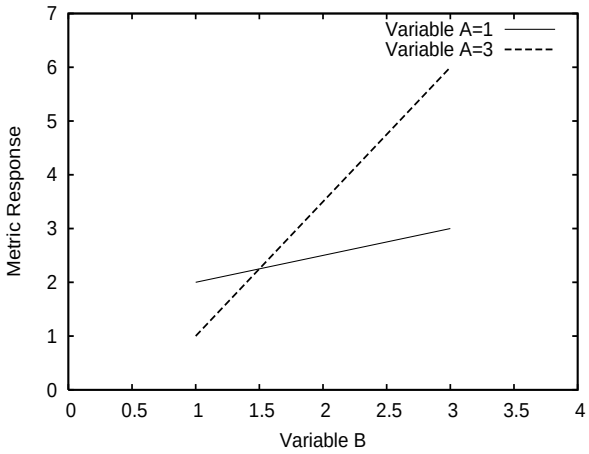

Figure 7: B does not depend on A.

est. To accomplish this reduction, we apply the Classification And Regression Tree (CART) algorithm [3], which is particularly effective at determining the importance of its input parameters, even when the distribution of their values is unknown. CART has three main components: an outcome variable (e.g., throughput or performability), a set of predictor variables (e.g., configuration parameters or hardware characteristics), and a learning dataset. Each data point in the learning dataset must include the values for the predictor variables and the value of the corresponding outcome variable. Using the learning dataset, CART constructs a binary classification tree by recursively splitting the dataset into partitions defined by values of the predictor variables. The purpose of each partition split is to reduce the diversity of the classifications in the partition. Partitions that become homogeneous are not split further. CART evaluates the importance of each predictor variable by its contribution to reducing diversity, i.e. the more important variables produce a larger reduction in diversity.

In our evaluation (Section 5), we define service throughput as the outcome variable. The set of configuration parameters defines our predictor variables. The learning dataset is collected from a set of runs of the service, as determined by the Simplex algorithm [14], which we describe below. For our purposes, the key output of CART is the importance factor it assigns to each parameter.

Second step: Finding dependencies between important parameters. Knowing which parameters are important, we now need to determine whether there are dependencies between them. However, the dependencies between even this smaller set of parameters may be complex and difficult to isolate as a single problem. Thus, we break it into pair-wise dependency tests between all pairs of important parameters. For a group of N important parameters, the number of pairs of parameters is $C_2^N = N * (N-1)/2$.

Our dependency test is as follows: a parameter A depends on parameter B, if and only if different settings of B lead to different best values for A. To understand our definition, consider figures 6 and 7. Figure 6 shows the values of a metric of interest as a function of parameters A and B. It is clear from this figure that the best value for A depends on the value of B, regardless of whether we seek to maximize or minimize the metric of interest. For example, if B=1, the value of A that minimizes the metric is 3. If B=3, the same value is 1. The same effect does not appear in Figure 7. In this figure, we can see that B does not depend on A according to our dependency test.

To restrict the set of dependencies to those that affect the metric of interest more substantially (and, thus, reduce parameter tuning overheads), we establish a "dependency threshold" below which the metric is assumed not to have changed. So, for example, if the difference between the best values for A when B=1 and B=3 in Figure 6 were smaller than the dependency threshold, we would have assumed that A does not depend on B. Thus, the dependency threshold has to be selected carefully; an excessively high threshold may miss real dependencies, whereas an excessively low threshold may find false dependencies.

Unfortunately, experimentally determining the behavior of the metric of interest with respect to all possible values for each pair of parameters, as in these figures, is not typically feasible. For this reason, our infrastructure requires service designers or operators to specify the range of reasonable values {*min value, max value*} for each numeric parameter. For designers and operators, doing so is obviously much simpler than having to specify the actual dependencies between a potentially large number of parameters. Our infrastructure experiments with all combinations of min, medium (the average between min and max), and max values for each pair of parameters. Thus, for each pair-wise comparison, we need $3 * 3 = 9$ experiments.

Finally, the parameter dependencies may be affected by the provisioning of the service tiers. For example, if throughput is the metric of interest and one tier is overprovisioned, it is likely that parameters from other tiers will not depend on the parameters of the overprovisioned tier. However, these dependencies may appear if the overprovisioned tier eventually becomes a performance bottleneck, as the service evolves. Thus, we run dependency checking once per tier, each time forcing a different tier to become the bottleneck by removing one or more of its nodes. To avoid checking dependencies between parameters in the same tier repeatedly, we only perform those checks when the tier is the bottleneck. The set of dependencies we encode in the graph is the union of those found in the provisioning experiments.

Number of experiments. Overall, for a set of N important parameters, our dependency-finding algorithm translates into $(\sum_{i=1}^{M} C_2^{N_i} + \sum_{i=1}^{M} \sum_{j=1, j \neq i}^{M} N_i * N_j) * 9$ experiments for each tier i with N_i important parameters and M is the number of tiers, i.e. $O(N^2)$ experiments. Considering that the total number of possible combinations of parameter values is $O(R^T)$, where T is the total number of parameters ($T >> N$) and R is the number of possible values

for each parameter ($R >> 3$), our algorithm provides a dramatic reduction of the dependency space. However, because our algorithm may fail to find dependencies that do exist, we call it a heuristic. Nevertheless, our understanding of the servers we study in this paper and their parameters suggests that our heuristic dependency-finding algorithm works well in practice.

As we mentioned above, the only information that our algorithm requires is the range of possible values for each parameter. This information can be communicated to our infrastructure using a simple interface. In fact, the same interface allows designers or operators to specify the parameter dependencies manually.

4.2 Traversing the Graph

Tuning parameters involves traversing the dependency graph and using the dependencies to limit the number of experiments. The traversal starts at the source vertices. If there are cycles in the graph, all vertices that form each cycle need to be tuned at the same time, i.e. we need to consider all combinations of values for all parameters in the cycle. In acyclic dependency chains, parameters that come earlier in the chain are tuned before those that come later. In detail, the source parameter is tuned first assuming the current values for all other parameters. After this step, all parameters reachable from the source in one edge traversal can be tuned independently, each of them assuming the best value for the source and the current values for all parameters. The next step involves the parameters that can be reached in two edge traversals and so on. If a vertex cannot reach any other vertices, all we need to do is explore the possible values for the corresponding parameter, keeping all other parameters fixed at their current best values. To explore the possible parameter values, we use Simplex. In particular, Simplex considers values in the $\{$*min value, max value*$\}$ range.

Number of experiments. The key advantage of using the dependency graph to guide the tuning of parameters is a major reduction in the number of experiments. Recall that the total number of possible combinations of parameter values is $O(R^T)$. The dependency graph transforms certain multiplicative factors in this large number into additive factors. To see this more clearly, consider the simple (and very pessimistic) scenario where all parameters are found to be important and each half of them form a dependency cycle. The number of possible combinations in this scenario would only be $R^{T/2} + R^{T/2}$ or $O(R^{T/2})$, which represents a significant reduction of the tuning space. In practice, reductions are even more substantial than in this case.

4.3 Simplex Algorithm

The Simplex algorithm, as extended by Nelder and Mead [14], is an efficient method for nonlinear, unconstrained optimization problems. The algorithm optimizes (maximizes or minimizes) unknown functions $f(x)$ for $x \in I\!R^n$. A *simplex* is a set of $n + 1$ points in $I\!R^n$, i.e. a triangle in $I\!R^2$, a tetrahedron in $I\!R^3$, and so on. The algorithm starts by selecting a random simplex and evaluating the function at each vertex of the simplex. Each iteration involves reflecting one of the vertices, but may also include expanding and contracting the simplex. These three operations are illustrated in Figure 8.

In more detail, each iteration involves the following steps: (1) Ordering – ordering the function values according to the optimization goal (e.g., descending order if the goal is to maximize the function); (2) Reflection – replace the vertex that leads to the worst function value (e.g., the smallest value when we are maximizing the function) by its mirror image in the centroid of the remaining n vertices. If the reflected value is better than the old value but not better than the best value currently, accept the reflected vertex and terminate the iteration; (3) Expansion – if the reflected value is better than the current best value, expand the reflected vertex away from the centroid. If the expanded value is better than the reflected value, take the former and terminate the iteration. Otherwise, take the latter and terminate; (4) Contraction – if the reflected value is worse than the next to worst value, perform a contraction of the worst vertex. If the contracted value is better than the worst value, take the former and terminate the iteration; and (5) Shrinking – shrink the simplex around the centroid and start another iteration.

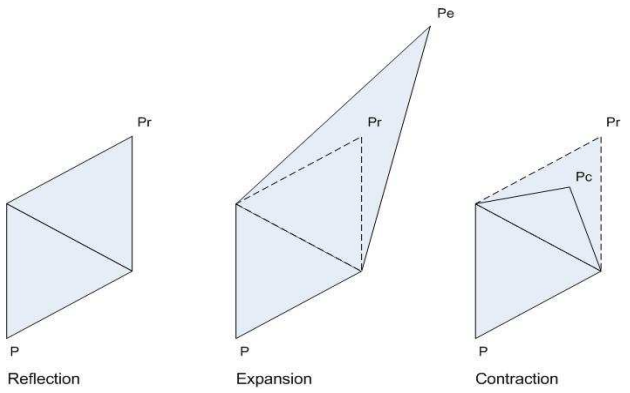

Figure 8: Main operations in the Simplex algorithm.

Number of experiments. The number of experiments required by each Simplex execution depends on the landscape being searched. In our evaluation, the parameter tuning of a three-tier service using Simplex required 299 experiments. With the support of our dependency graph, only 73 experiments were required.

5. EVALUATION

In this section, we present our experimental setup and results. We first evaluate our infrastructure's ability to simplify service management and eliminate misconfigurations. Next, we evaluate our infrastructure's ability to find good configurations efficiently.

5.1 Experimental Setup

For comparison purposes, we experiment with the same online auction service used in [13]. The service is organized into three tiers of servers: Web, application, and database tiers. We use two machines in the first tier running the Apache Web server (version 2.0.54), three machines running the Tomcat servlet server (version 4.1.18) in the second tier and, in the third tier, one machine running the MySQL relational database (version 4.12). The service requests are received by the Web servers and may flow towards the second and third tiers. The replies flow through the same path in the reverse direction.

We exercise the service using a client emulator. The workload consists of a "bidding mix" of requests (94% of the database requests are reads) issued by a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, "thinks" for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. The code for the service, the workload, and the client emulator are from the DynaServer project [18].

The machines that run the service and the clients are Nexcom blade servers with 512 MB of memory, 5400 rpm IDE disks, and 1.2 GHz Celeron CPUs running the Linux kernel (version 2.4.27)

	Measure	Node addition		Apache upgrade		Apache diag		DB upgrade	
		Without	With	Without	With	Without	With	Without	With
Exec.	NumActions	8	0	12	12	7	1	20	11
	ContextSwitchSum	6	0	16	16	2	0	44	22
Param.	ParamCount	7	0	6	6	0	0	4	2
	ParamUseCount	13	0	12	8	0	0	10	4
	ParamCrossContext	24	0	8	6	0	0	18	2
	ParamAdaptCount	0	0	4	4	0	0	0	0
	ParamSourceScore	25	0	24	24	0	0	12	6
Memory	MemSizeMax	7	0	4	2	$\geq$1	0	4	2
	MemSizeAvg	2.4	0	2	1.2	NA	0	1.33	0.67
	MemDepthMax	7	0	3	2	$\geq$1	0	2	2
	MemDepthAvg	2.3	0	1.4	1.2	NA	0	1	0.67
	MemLatMax	3	0	3	0	$\geq$1	0	1	1
	MemLatAvg	0.5	0	0.6	0	NA	0	0.44	0.33

Table 1: Management complexity with and without our infrastructure.

connected by a Fast Ethernet switch.

We have implemented our infrastructure (6K lines of C and Perl code) and the generation scripts and configuration templates for Apache, Tomcat, and MySQL.

5.2 Complexity and Misconfigurations

Some of the central goals of our infrastructure are to generate configuration files automatically, simplifying service management and eliminating most misconfigurations. To quantify management complexity with and without our infrastructure, we use the measures recently proposed by Brown *et al.* [4]. Although subjective in some cases, their measures do provide insight into complexity and represent the only comprehensive set of complexity measures of which we know for service management.

The measures define management complexity according to three aspects of operator tasks: the actions that they need to execute (execution complexity), the parameters values that they need to select (parameter complexity), and the requirements on their minds (memory complexity). In more detail, the *execution complexity* of a task measures the number of actions (NumActions) and the number of "context switches" (ContextSwitches) it involves. By their definition, a context switch occurs between any two consecutive actions that act upon two different servers or subsystems. The *parameter complexity* measures the number of parameters (ParamCount), the number of times parameters are supplied (ParamUseCount), the number of times parameters are used in more than one context weighted by the distance between contexts (ParamCrossContext), the number of times parameters are used in a different syntactic form (ParamAdaptCount), and the sum across all parameters of a score from 0 to 6 based on how hard (0 = easy, 6 = hard) it is to obtain the value of each parameter (ParamSourceScore). Finally, the *memory complexity* measures the number of parameters that must be remembered (MemSize), the length of time they must be retained in memory (MemDepth), and how many intervening items are stored in memory between uses of a parameter (MemLatency). We consider the maximum and average of these memory complexity measures.

To demonstrate the complexity benefits of our infrastructure, we study the operator tasks and mistakes described in [13]. Table 1 lists the complexity results with and without our infrastructure for the four tasks that involve configuration: node addition in the application-server tier, Apache upgrade, database node upgrade, and diagnosing and fixing an Apache misconfiguration. Note that the results in the table underestimate the benefits of our infrastructure, as they do not account for the complexity of manual parameter tuning after the first three tasks when our infrastructure is not used.

Despite this underestimation and the fact that our infrastructure only helps with the server configuration part of these tasks, the table shows that it can still reduce complexity significantly. The execution complexity is reduced because our infrastructure automates most operator tasks related to the servers, e.g. modifying worker.properties as a result of application server additions. With automation, the operator does not need to perform most actions and, thus, does not context switch between actions. Along the same lines, the automation brought about by our infrastructure also reduces the parameter and memory complexities, as it relieves the operator from having to supply or remember most parameters, e.g. the IP address of an added node or the hostname of the node running the database server.

The exception to these general trends is the Apache upgrade task for which our infrastructure reduces the parameter and memory complexities somewhat, keeping the execution complexity unaffected. The reason for this behavior is that the Apache upgrade task is dominated by non-configuration executions: installing the new version in a different directory, copying content to the new directory, setting up the heartbeat service, shutting down the old version, and starting the new version.

To evaluate the ability of our infrastructure to eliminate misconfigurations, we repeat the operator-emulation experiments from [13]. In particular, we use the operator action traces that they collected for the four tasks above. Table 2 shows the misconfigurations that our infrastructure eliminates (left) and those that it does not (right). Overall, our infrastructure eliminates 14 out of a total of 24 observed misconfigurations, i.e. 58%.

Note that some misconfigurations remain because, for certain tasks, the operator is required to change a few parameters in the configuration file templates. Specifically, the operator needs to supply the htdocs directory (the root directory of the content to be served), set the correct translation between URLs and Tomcat servlets, and point to the correct implementation of the membership protocol (the heartbeat service) for the Apache upgrade task. These changes account for 6 of the remaining misconfigurations. The other 4 remaining misconfigurations do not involve configuration files, so our infrastructure could not have eliminated them. Specifically, they occurred during the upgrade of the database node (3) and when diagnosing and repairing an application server hang (1).

Eliminated Misconfigurations		Remaining Misconfigurations	
Description	Instances	Description	Instances
Duplicated entry in Apache worker.properties	4	Wrong Apache htdocs directory	3
Unmodified last line of Apache worker.properties	3	Apache pointing to wrong heartbeat service	2
Extra space in last line of Apache worker.properties	2	URLs not mapped to servlets in Apache httpd.conf	1
Non-existing Tomcat server in Apache worker.properties	1	MySQL access rights not given to Tomcat servers	2
Unmodified Apache worker.properties	1	No password for MySQL root	1
Missing listen port in Apache httpd.conf	1	MySQL writes forbidden	1
Wrong listen port in Tomcat configuration file	1		
Unmodified Tomcat configuration file	1		

Table 2: Misconfigurations with our infrastructure.

Apache	`TimeOut`, `MaxKeepAliveRequests`, `KeepAliveTimeout`, `StartServers`, `MinSpareServers`, `MaxSpareServers`, `MaxClients`, `MaxRequestsPerChild`
Tomcat	`maxProcessors`, `MinProcessors`, `acceptCount`
MySQL	`key_buffer`, `max_allowed_packet`, `table_cache`, `sort_buffer`, `read_buffer_size`, `record_buffer`, `myisam_sort_buffer_size`, `thread_cache`, `query_cache_size`, `thread_concurrency`, `innodb_buffer_pool_size`, `innodb_additional_mem_pool_size`, `max_connections`

Table 3: Performance-relevant parameters.

5.3 Parameter Tuning

The other key goal of our infrastructure is to generate configuration files with optimized parameter values. In this section, we assume that the metric of interest is throughput performance, which we want to maximize. Because of our focus on performance, we consider only the 24 parameters that may impact it, which are listed in Table 3. Each performance experiment was run for 2 minutes with warm caches.

Generating the parameter dependency graph. The first step in generating the dependency graph for our auction service is to find the important parameters using CART. To do so, we create a learning dataset for CART using Simplex to find good settings for the above parameters for each tier independently. We start with a random configuration. The first Simplex run tries to find good parameter values for the database tier, the throughput bottleneck. We then set the database server parameters to the values found by Simplex, make the application-server tier the bottleneck (by decreasing the number of nodes in this tier), and run Simplex again to find good parameter values for the application servers. Finally, we set the parameter values of the application servers and of the database server to those found by Simplex, make the Web-server tier the bottleneck (by bringing back the application server nodes and removing Web server nodes), and run Simplex again to find good parameter values for the Web servers. The total number of experiments for these three Simplex runs is 254.

CART can use the results of these experiments to compute the relative importance of the parameters. Table 4 lists the 9 parameters with relative importance greater than 20. We initially picked this threshold value based on our intuitive understanding of the servers involved. Later, we realized that a threshold of 30 or 35 could have produced the same results using many fewer experiments. Nevertheless, we decided to stay with 20 to study our approach in a more challenging scenario.

Tier	Parameter	Relative Importance
Apache	MaxRequestsPerChild	100
	MaxClients	44
	StartServers	24
Tomcat	acceptCount	100
	MinProcessors	38
	maxProcessors	26
MySQL	innodb_buffer_pool_size	100
	max_connections	47
	query_cache_size	32

Table 4: Parameters with relative importance greater than 20 as computed by CART.

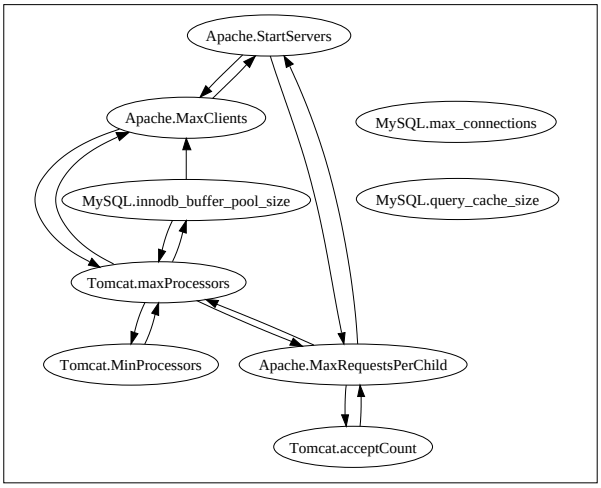

Figure 9: Parameter dependencies.

The second step in generating the graph involves finding dependencies between parameters. Following the procedure described in Section 4.1, we performed 567 experiments. To determine the dependencies, we set the dependency threshold to 10%, leading to the relationships portrayed in Figure 9.

Performance tuning. We now evaluate the use of our parameter dependency graph for automatic performance tuning as the service evolves. The first evolution is the migration of the bottleneck MySQL server to a more powerful machine, which has a 2.8 GHz Xeon CPU, 2 GB of memory, and a RAID-5 SCSI disk array. Starting from the upgraded service, the second evolution is the removal of two application-server nodes to mimic a scheduled maintenance event or a brownout.

The tuning of the parameters after the evolutions can be performed using several methods:

Evolution	Tuning Approach	Throughput (reqs/sec)	Number of Experiments
DB node upgrade	Best values prior to DB node upgrade	343	–
	Simplex search	372	299
	Dependency graph + Simplex	377	821 + 73
Removal of 2 appl. servers	Best values prior to DB node upgrade	91	–
	Simplex search	114	220
	Dependency graph + Simplex	110	0 + 35

Table 5: Comparing performance tuning approaches for two service evolutions.

- Exhaustive search: Explores all combinations of parameter values. Overall, $O(R^N)$ experiments, where R = range and N = total number of parameters = 24. Obviously, exploring this entire space is infeasible.
- Simplex search: Uses the algorithm described in Section 4.3 to tune the 24 parameters. The number of experiments here depends chiefly on the landscape being searched.
- Dependency graph + Exhaustive search: Explores all combinations of values for the groups of dependent parameters. For our service's dependency graph (Figure 9), this means an exhaustive search of the combinations of values for 7 important parameters and an exhaustive search of the other 2 important parameters independently. Again, exploring this space is infeasible.
- Our approach = Dependency graph + Simplex search: Run Simplex on the groups of dependent parameters. For our service's dependency graph, this means running Simplex on 7 important parameters and exhaustive search for the 2 remaining ones independently.

Table 5 compares the number of experiments and tuned throughput entailed by the approaches that are feasible for each of the evolutions. Each number of experiments in our approach is presented as the sum of the number of experiments for generating the dependency graph and for performance tuning based on the graph. For the results in the table, the Simplex searches were set to terminate when the standard deviation of the throughputs forming the simplex was lower than 3 reqs/sec. The first row of each group lists the throughput of the best parameter values prior to the database node upgrade when applied to the service after the evolutions. (We used these values as the starting point of the Simplex searches.)

It is interesting to observe that our infrastructure achieves high throughputs with the smallest number of performance-tuning experiments. In fact, its throughput is about 10% (database node upgrade) and 21% (removal of two application-server nodes) higher than that of the best parameter values prior to the first evolution. However, our approach also involves a large number of experiments (254 + 567 = 821) for generating the dependency graph. The effect of this extra overhead is that our infrastructure needs a few evolutions to become cheaper than the Simplex approach. In particular, after these two evolutions, our approach has executed 929 (821 + 73 + 35) experiments, whereas the Simplex approach has involved 519 (299 + 220) experiments. Thus, we have been able to amortize 411 (299-73 + 220-35) experiments from the overhead over the two evolutions. Assuming that this amortization rate would persist, it would only take two more evolutions for our approach to break even with the Simplex approach.

Another interesting observation is that the actual values ultimately chosen by the tuning approaches are different for all tuned parameters, as a result of how Simplex searches the space. The tuned parameter values are also consistently different than the best values prior to the DB node upgrade.

Amortizing the overhead of the dependency graph. A key remaining question is whether the dependency graph can indeed be amortized over numerous service evolutions. To answer this question, we compared the dependency graphs generated by our heuristic algorithm before and after service evolutions. Finding the same (or a very similar) graph after an evolution suggests that the graph does not have to be regenerated as a result of the evolution.

We considered three types of evolutions: the database node upgrade, the removal of the two application-server nodes, and the upgrade of MySQL server (version 4.12) to version 5. Our results show that the database node upgrade leads to the same dependency graph as with the slower node. We expect that system software upgrades, e.g. upgrades of the operating system, for improving performance should lead to the same results as this evolution.

Removing two application-server nodes also led to the same graph as before the upgrade. Because of the way we generate the dependency graph, we expect other membership evolutions, e.g. node additions, to behave similarly.

The MySQL upgrade was more interesting. We first had to verify that the performance parameters in the new version of MySQL are the same as in the old version, which is indeed the case. (In fact, the same is true of the most recent versions of Apache and Tomcat.) After generating the dependency graph for the upgraded service, we found that it is a subset of the original dependency graph, suggesting that the original graph would still be useful.

Despite our ability to amortize the overhead of generating the dependency graph over many types of evolutions, we may eventually need to regenerate it. For example, the dependency graph may become inaccurate after a large number of evolutions, even if each evolution affects actual dependencies only slightly. To deal with these situations, operators can periodically start the regeneration of the dependency graph; the frequency of regenerations should depend on the overheads involved and on the frequency of evolutions.

6. CONCLUSIONS

In this paper, we proposed a software infrastructure for automatically generating configuration files for cluster-based Internet services. The infrastructure introduces the notion of a parameter dependency graph and algorithms to generate the graph and optimize the service with it. Our evaluation showed that the infrastructure can simplify the service operation, eliminate operator mistakes, and generate high-performance configurations efficiently.

Limitations and future work. Although our experience and results are clearly positive, we plan to extend the work presented here in a few different ways. In particular, we are studying the sensitivity of our results to the parameter-importance threshold in more detail. Our results for the first evolution above suggest that

a threshold of 35 would have been a better choice than 20, since it produces essentially the same throughput but with many fewer experiments. We plan to study approaches to extend our infrastructure to deal with application-level evolutions. Furthermore, we plan to extend our infrastructure to automatically detect significant changes in workload characteristics, which should also prompt parameter tuning. As a longer-term goal, we plan to study performability (performance + availability), rather than performance alone, as the metric to be optimized by our infrastructure.

In closing, it is important to mention four limitations of our work. First, our current infrastructure only works for numerical parameters; categorical or symbolic parameters are not handled. Second, our current experimental methodology is not very robust for systems that are prone to experimental noise (e.g., performance degradation due to unstable hardware, non-deterministic software, or operating system daemons that are activated during experiments). For these systems, a more robust methodology could be to run each experiment twice and use the best value of the metric of interest, for example. Although our system is indeed prone to noise, we opted for lower overheads. Our positive results demonstrate our current methodology worked well in practice. Third, our sample Internet service, the online auction, is substantially simpler than real commercial services. Configuring these real services may pose challenges that we are not addressing. For example, a real service may involve a larger number of different servers and configuration parameters. It is possible that the search space would have to be reduced further to make automatic tuning practical for such a service. Finally, we do not have access to statistics on the frequency and type of evolutions that real services experience. Note however that these limitations plague the vast majority of academic studies of Internet services, since real services rarely divulge information about their internal structure and evolution. Despite these limitations, we believe that real services can certainly leverage the principles and ideas introduced here in their more complex environments.

7. ACKNOWLEDGEMENTS

We would like to thank Gustavo Alonso, Thomas Gross, Richard Martin, Willy Zwaenepoel, and the members of the Vivo group at Rutgers and LABOS group at EPFL for their many helpful comments on the topic of this paper. We would also like to thank the anonymous reviewers for helping us improve the paper. Finally, the research presented here was partially supported by NSF grants #EIA-0103722 and #CSR-0509007.

8. REFERENCES

[1] ANDERSON, P., GOLDSACK, P., AND PATERSON, J. SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control. In *Proceedings of the 17th Systems Administration Conference (LISA 2003)* (San Diego, CA, Oct. 2003).

[2] ANDERSON, P., AND SCOBIE, A. LCFG: The Next Generation. In *UKUUG Winter Conference* (2002), UKUUG.

[3] BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A., AND STONE, C. J. *Classfication and Regression Trees*. 1984.

[4] BROWN, A. B., KELLER, A., AND HELLERSTEIN, J. L. A Model of Configuration Complexity and its Application to a Change Management System. In *Proc. 9th IFIP/IEEE Int'l Symposium on Integrated Network Management (IM 2005)* (2005).

[5] BURGESS, M. Cfengine: A site configuration engine. *USENIX Computing systems 8*, 3 (1995).

[6] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

[7] CHUNG, I.-H., AND HOLLINGSWORTH, J. K. Automated Cluster-Based Web Service Performance Tuning. In *The Thirteenth IEEE International Symposium on High-Performance Distributed Computing* (Honolulu, Hawaii USA, June 2004).

[8] DIAO, Y., HELLERSTEIN, J., PAREKH, S., AND BIGUS, J. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal 42*, 1 (2003).

[9] KICIMAN, E., AND WANG, Y.-M. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (May 2004).

[10] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by Simulated Annealing. *Science 220*, 4598 (May 1983).

[11] Levanta. http://www.levanta.com.

[12] NAGARAJA, K., GAMA, G. M. C., BIANCHINI, R., MARTIN, R. P., JR., W. M., AND NGUYEN, T. D. Quantifying the Performability of Cluster-Based Services. *IEEE Transactions on Parallel and Distributed Systems 16*, 5 (2005).

[13] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[14] NELDER, J. A., AND MEAD, R. A Simplex Method for Function Minimization. *Computer Journal 7*, 4 (1965).

[15] OLIVEIRA, F., PATEL, J., HENSBERGEN, E. V., GHEITH, A., AND RAJAMONY, R. Blutopia: Cluster Life-Cycle Management. Tech. Rep. RC23784, IBM Austin, November 2005.

[16] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[17] OPPENHEIMER, D., AND PATTERSON, D. Architecture and Dependability of Large-Scale Internet Services. *IEEE Internet Computing 6*, 5 (2002).

[18] RICE UNIVERSITY. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[19] STEWART, C., AND SHEN, K. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation* (May 2005).

[20] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[21] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Systems Administration Conference (LISA 2003)* (San Diego, CA, Oct. 2003).

[22] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).