EXPERIMENT-BASED MANAGEMENT OF DATA CENTERS

BY WEI ZHENG

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Ricardo Bianchini

and approved by

New Brunswick, New Jersey

May, 2012

ABSTRACT OF THE DISSERTATION

Experiment-based Management of Data Centers

by Wei Zheng

Dissertation Director: Ricardo Bianchini

Our daily lives depend on data centers as they host popular Internet services and critical business applications. Managing these large and complex data centers is a challenging endeavor. We have designed and implemented three systems that simplify the management of data centers. The unifying characteristic of these systems is that they all rely on a limited number of experiments with real servers and workloads.

The first system, called JustRunIt, replaces analytical modeling in assessing the performance, availability, and/or energy implications of potential management decisions or system configurations. The second system, called ACI, efficiently optimizes configurations as services evolve, by detecting and leveraging dependencies between configuration parameters. The last system, called MassConf, automatically configures server software for new users by leveraging configuration information from the existing users of the software.

The evaluation shows that our systems significantly reduce the resources and time required to accomplish many management tasks. Given our experience and positive results with these three systems, we conclude that experiment-based management has the potential to be very useful in practice.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Ricardo Bianchini, whose guidance, encouragement, and support made the thesis possible. Not only you worked closely with me on all research topics, but also your passion towards work and life greatly influenced me and served me as a role model. The problem solving skills, creative thinking, and can-do attitude that I learned through the research process built solid foundations for the future of my career.

I would like to thank my thesis committee members, Professor Thu D. Nguyen, Rich Martin, and Jason N. Flinn for valuable feedback on the thesis. I also would like to thank my internship mentors Alan Bivens at IBM Research; John Janakiraman, José Renato Santos, and Yoshio Turner at HP Labs; and YaYunn Su at NEC Labs. The internships and mentorships helped me to understand real world problems and to identify the topic of my thesis.

It is an honor to be a member of DARK lab, where hard work and fun coexist. It is a great pleasure to have worked with my labmates, Fabio Oliveira, Rekha Bachwani, Qingyuan Deng, Inigo Goiri, Kien Le, Cheng Li, Luiz Ramos, and Ana Paula. I enjoyed open discussion, brainstorming, and friendship over the course of seven years. Most importantly, you made the challenging, sometimes frustrating process much easier to go through.

Finally, I would like to thank deeply my lovely wife Fang and son Ryan, my parents, and parents-in-law. Their spiritual and financial support enabled me to devote time to the work I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Data centers are part of our daily lives. Some data centers host popular Internet services, such as Google, Yahoo, and Amazon, while others support cloud computing systems, such as Amazon EC2 and Microsoft Azure. These data centers can easily host thousands of computers and their supporting infrastructure. Moreover, these services usually comprise multiple tiers that interact in complex ways. A typical three-tier service has a Web server tier, an application server tier, and a database tier. A client request may pass through all tiers in one direction, whereas the reply flows through the same tiers in the opposite direction.

The server software in each tier can also be complex. For example, designers introduce a large number of configuration parameters into server software to accommodate various end-user needs. Among many other examples, there may be parameters specifying the number of threads to create, the timeout before an idle network connection is destroyed, and/or the amount of memory to use for caching disk data. The configuration parameters affect the functionality, performance, availability, and energy consumption of data centers. Thus, selecting proper values for them is critical. Unfortunately, doing so is often difficult.

Worse, configuration setting/tuning is only one of many data center management tasks. Other tasks include managing resources to satisfy service-level agreements (SLAs), adding or removing nodes, upgrading hardware and/or software, and capacity planning. The advent of virtualization simplified certain tasks, but also added an extra layer of software to be managed. Evaluating each possible management decision often requires understanding its performance, availability, and energy implications.

The complexity of the management tasks in large data centers represents a major burden on their operators. Previous studies [48] found that operator mistakes are a common source of availability, performance, and security problems in Internet services. Among the mistakes, misconfigurations were the most common type. Another study [84] also observed that misconfigurations are one of the dominant causes of system issues through analysis of extensive real world misconfiguration cases, including commercial storage systems and other open source systems.

These studies suggest that, as much as possible, management tasks should be automated. The aim of this thesis is to create software that enables automated, accurate, and efficient data center management.

1.2 State of the Art

Many researchers have realized the problem and proposed various approaches to manage data centers: (1) [9, 19, 25, 27, 32, 64, 65, 71, 72, 75] use analytical modeling to automate data center management; (2) [52, 53] leverage feedback control theory for resource management; (3) [1, 78, 79] use coercion to troubleshoot misconfigurations; (4) [2, 3, 12] build infrastructures for automatic configuration management; and (5) [41, 45, 46, 74] validate configuration changes or operator actions in an isolated environment before deployment.

Even though the previous works are useful in many cases, they have some important limitations. Out of the approaches listed above, analytical modeling is the only one that can be used to automate all of the management tasks we consider. Specifically, modeling can be used to predict the impact of the possible management decisions or system configurations on performance, availability, and/or energy consumption. Performance models are often based on queuing theory, whereas availability models are often based on Markovian formalisms. Energy models are typically based on simple (but potentially inaccurate) models of power consumption, as a function of CPU utilization or CPU voltage/frequency. On the bright side, these models are useful in data center management as they provide insight into the systems' behaviors, can be solved quickly, and allow for large parameter space explorations. Essentially, the models provide an efficient way of answering "what-if" questions during management tasks.

Unfortunately, modeling has a few serious shortcomings. First, modeling consumes a very expensive resource: highly skilled human labor to produce, calibrate, and validate the models. The cost of this labor is built into the management systems and is always disregarded in their evaluations. Second, the models typically rely on simplifying assumptions. For example, memoryless arrivals is a common assumption of queuing models for Internet services [65]. However, this assumption is invalid when requests come mostly from existing sessions with the service. Another common simplifying assumption is the cubic relationship between CPU frequency and power consumption [19]. With advances in CPU power management, such as clock gating, the exact power behavior of the CPU is becoming more complex and, thus, more difficult to model accurately. Third, the models need to be re-calibrated and re-validated as the systems evolve. For example, the addition of new machines to a service requires queuing models to be calibrated and validated for them.

The other approaches can only be applied to a limited number of management tasks. Feedback control is useful for resource management, but cannot be used for tasks that do not involve repeated action adjustments. Capacity planning and software or hardware upgrades are obvious examples of the latter type of tasks. Coercion does not prevent misconfigurations from occurring in the first place and still leaves users trying to tune performance clueless. Configuration management systems mainly provide support for machine installation and startup configuration. Lastly, validation presumes stable system behavior, while often management tasks intend to change the systems.

1.3 Challenges and Contributions

In the dissertation, we argue that the best approach to automation is one that is based on experiments with the managed systems themselves. Experiments can produce results realistically and transparently, enabling automated management systems to perform their tasks effectively. Actual experiments exchange an expensive resource (human labor) for much cheaper ones (the time and energy consumed by a few machines in running the experiments). Thus, actual experiments are cheaper, simpler, and more accurate than models in their ability to answer “what-if” questions. Moreover, experiments can be easily integrated with many automated management tasks.

Despite the advantages of experiment-based management, its key challenges are to produce accurate experiments and limit the amount of resources and/or time used in the experimentation.

To address these challenges, we design and build three experiment-based management systems for data centers, namely JustRunIt, Automatic Configuration Infrastructure (ACI), and MassConf. We apply our systems across different management tasks, including resource management, configuration setting, performance tuning, and hardware upgrades. The evaluation shows that our systems can always automate management with a limited number of experiments (time). In some cases, we can also restrict the number of machines used for the experiments (resources). In the next few subsections, we overview our three systems.

1.3.1 JustRunIt

JustRunIt is a software infrastructure for experiment-based management of virtualized data centers, such as those of cloud computing providers. The idea behind JustRunIt is to utilize a small fraction of the resources of the data center to create a sandboxed environment where experiments can be performed away from the production system.

JustRunIt can be used by higher level automated management systems or by the operators themselves to answer ”what-if” questions during management tasks. To demonstrate JustRunIt, we combine it with a system that performs server consolidation/expansion and a system that evaluates the benefits of hardware upgrades.

Our evaluation demonstrates that JustRunIt can help automate many tasks, and produce accurate results despite using a very limited amount of resources.

1.3.2 ACI

ACI is a software infrastructure for experiment-based performance tuning of Internet services. Its goal is to select settings for the service’s configuration parameters every time it evolves, e.g. new machines are added or removed. ACI limits the number of experiments by creating and leveraging a dependency graph between the configuration parameters. The graph identifies the parameters that may need new values as a result of each possible change to the service. To select the values, ACI uses an optimization algorithm that is driven by the dependency graph.

Our evaluation of ACI demonstrates that it is capable of producing high-performance configurations while limiting the number of tuning experiments. Our results also show that ACI prevents a large number of misconfigurations reported by Nagaraja *et al.* [41].

1.3.3 MassConf

Finally, MassConf simplifies the configuration process for new users of server software, by leveraging the existing users’ configurations. MassConf dynamically collects and ranks users’ configurations. At each new user’s site, MassConf then evaluates the ranked configurations, selecting the first one that produces the desired behavior. The two key observations behind the approach are that (1) a ”good” configuration may work well for many different users, and (2) multiple good configurations may work well for each user.

To evaluate MassConf, we use it to configure the Apache Web server to achieve a response-time target. Our evaluation confirms our observations and shows that MassConf successfully reaches the new users' performance targets in fewer experiments than existing approaches.

1.4 Overview of the Dissertation

The dissertation comprises six chapters. Chapter 2 describes the design, implementation, and evaluation of JustRunIt. Chapter 3 details the concept of configuration dependency and describes ACI. Chapter 4 introduces MassConf. Chapter 5 discusses related works. Finally, Chapter 6 concludes the dissertation.

Chapter 2

JustRunIt

2.1 Introduction

Managing data centers is a challenging endeavor, especially when done manually by system administrators. One of the main challenges is that performing many management tasks involves selecting a proper resource allocation or system configuration out of a potentially large number of possible alternatives. Even worse, evaluating each possible management decision often requires understanding its performance, availability, and energy consumption implications. For example, a common management task is to partition the system's resources across applications to optimize performance and/or energy consumption, as is done in server consolidation and virtual machine (VM) placement. Another example is the evaluation of software or hardware upgrades, which involves determining whether application or system behavior will benefit from the candidate upgrades and by how much. Along the same lines, capacity planning is a common management task that involves selecting a proper system configuration for a set of applications.

Previous efforts have automated resource-partitioning tasks using simple heuristics and/or feedback control, e.g. [4, 15, 53, 60, 76, 83]. These policies repeatedly adjust the resource allocation to a change in system behavior, until their performance and/or energy goals are again met. Unfortunately, when this react-and-observe approach is not possible, e.g. when evaluating software or hardware upgrades, these policies cannot be applied.

In contrast, analytical modeling can be used to automate all of these management tasks. For example, researchers have built resource-partitioning systems for hosting centers that use models to predict throughput and response time, e.g. [25, 75]. In addition, researchers have built systems that use models to maximize energy conservation in data centers, e.g. [19, 32]. Finally, researchers have been building models that can predict the performance of Internet applications on CPUs with different characteristics [64]; such models can be used in deciding whether to upgrade the server hardware.

However, as discussed in the previous chapter, models need to be re-calibrated and re-validated as the system evolves. Moreover, building these models is expensive, as it requires highly skilled human labor. Given these limitations, in this chapter we demonstrate that experiments are more effective at answering ”what-if” questions and supporting the management tasks we consider. In particular, we demonstrate that it is possible to produce flexible, realistic, and transparent experiments using current virtualization technology.

To support our claims in a challenging environment, we built JustRunIt, an infrastructure for experiment-based management of virtualized data centers hosting multiple Internet services. JustRunIt creates a sandboxed environment in which experiments can be run on a small number of machines (e.g., one machine per tier of a service) without affecting the on-line system. JustRunIt clones a small subset of the on-line VMs (e.g., one VM per tier of the service) and migrates them to the sandbox. In the sandbox, JustRunIt precisely controls the resources allocated to the VMs, while offering the same workload to them that is offered to similar VMs on-line. Workload duplication is implemented by JustRunIt's server proxies. For flexibility, the administrator can specify the resources (and the range of allocations) with which to experiment and how long experiments should be run. If there is not enough time to run all possible experiments (i.e., all combinations of acceptable resource allocations), JustRunIt uses interpolation between actual experimental results to produce the missing results but flags them as potentially inaccurate.

Automated management systems or the system administrator can use the JustRunIt results to perform management tasks on the on-line system. If any interpolated results are actually used by the system or administrator, JustRunIt runs the corresponding experiments in the background and warns the administrator if any experimental result differs from the corresponding interpolated result by more than a threshold amount.

To evaluate our infrastructure, we apply it to systems that automate two common management tasks: server consolidation/expansion and evaluation of hardware upgrades. Modeling has been used in support of both tasks [19, 65], whereas feedback control is only applicable for some cases of the former [19]. JustRunIt combines nicely with both systems. Our evaluation demonstrates that JustRunIt can produce results realistically and transparently, enabling automated management systems to perform their tasks effectively. In fact, JustRunIt can produce system configurations that are as good as those resulting from idealized, perfectly accurate models, at the cost of the time and energy dedicated to experiments.

The remainder of the chapter is organized as follows. The next section describes JustRunIt in detail. Section 2.3 describes the automated management systems that we designed for our two case studies. Section 2.4 presents our evaluation of JustRunIt and the results of our case studies. Finally, Section 2.5 summarizes this chapter, discusses the limitations of JustRunIt, and mentions potential future works.

2.2 JustRunIt Design and Implementation

In this section, we describe the design and implementation of JustRunIt, and present an overview of its use.

2.2.1 Target Environment

Our target environment is virtualized data centers that host multiple independent Internet services. Each service comprises multiple tiers. For instance, a typical three-tier

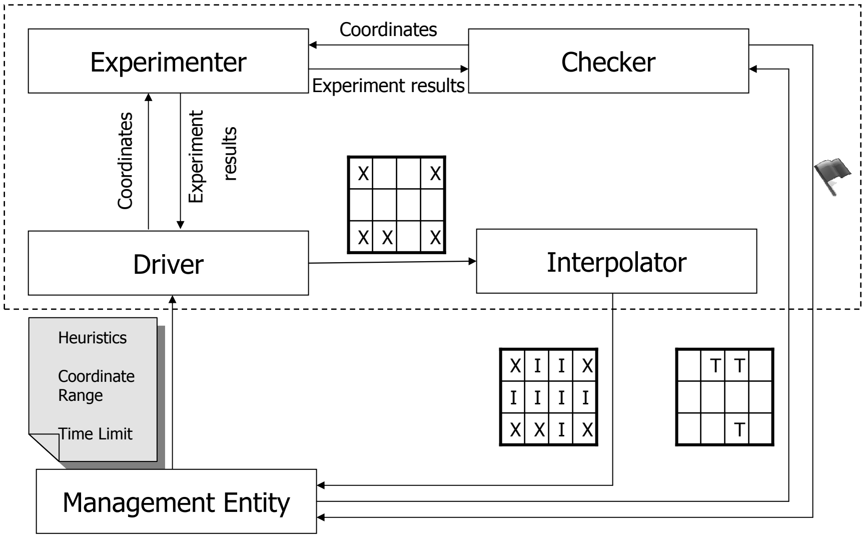

Figure 2.1: Overview of JustRunIt. “X” represents a result obtained through experimentation, whereas “I” represents an interpolated result. “T” represents an interpolated result that has been used by the management entity.

Internet service has a Web tier, an application tier, and a database tier. Each tier may be implemented by multiple instances of a software server, e.g. multiple instances of Apache may implement the first tier of a service. Each service has strict response-time requirements specified in SLAs (Service Level Agreements) negotiated between the service provider and the data center.

In these data centers, all services are hosted in VMs for performance and fault isolation, easy migration, and resource management flexibility. Moreover, each software server of a service is run on a different VM. VMs hosting software servers from different services may co-locate on a physical machine (PM). However, VMs hosting software servers from the same service tier are hosted on different PMs for high availability. All VMs have network-attached storage provided by a storage server.

Our target environment presents many challenges for JustRunIt. First, Internet services exhibit dynamically varying workloads and load intensities, suggesting that it is important to experiment with live workloads for greater realism. Second, Internet services exhibit relatively lightweight units of work (service requests), suggesting that any infrastructure overheads may have a significant impact on observed performance. Finally, Internet services have strict response-time requirements specified in SLAs (Service Level Agreements) negotiated between the service providers and the data center. SLAs again mean that overheads must be minimized.

2.2.2 System Infrastructure

Figure 2.1 shows an overview of the system infrastructure of JustRunIt. There are four components: experimenter, driver, interpolator, and checker. The *experimenter* implements the VM cloning and workload duplication mechanism to run experiments. Each experiment tests a possible configuration change to a cloned software server under the current live workload. A configuration change may be a different resource allocation (e.g., a larger share of the CPU) or a different hardware setting (e.g., a higher CPU voltage/frequency). The results of each experiment are reported as the server throughput, response time, and energy consumption observed under the tested configuration.

The experiment *driver* chooses which experiments to run in order to efficiently explore the configuration parameter space. The driver tries to minimize the number of experiments that must be run while ensuring that all the experiments complete within a user-specified time bound. The driver and experimenter work together to produce a matrix of experimental results in the configuration parameter space. The coordinates of the matrix are the configuration parameter values for each type of resource, and the values recorded at each point are the performance and energy metrics observed for the corresponding resource assignments. When experiments are run with multiple service tiers, a result matrix is generated for each of them.

Blank entries in the matrix are filled in by the *interpolator*, based on linear interpolation from the experimental results in the matrix. The filled matrix is provided to the management entity–i.e., the system administrator or an automated management system–for use in deciding resource allocations for the production system.

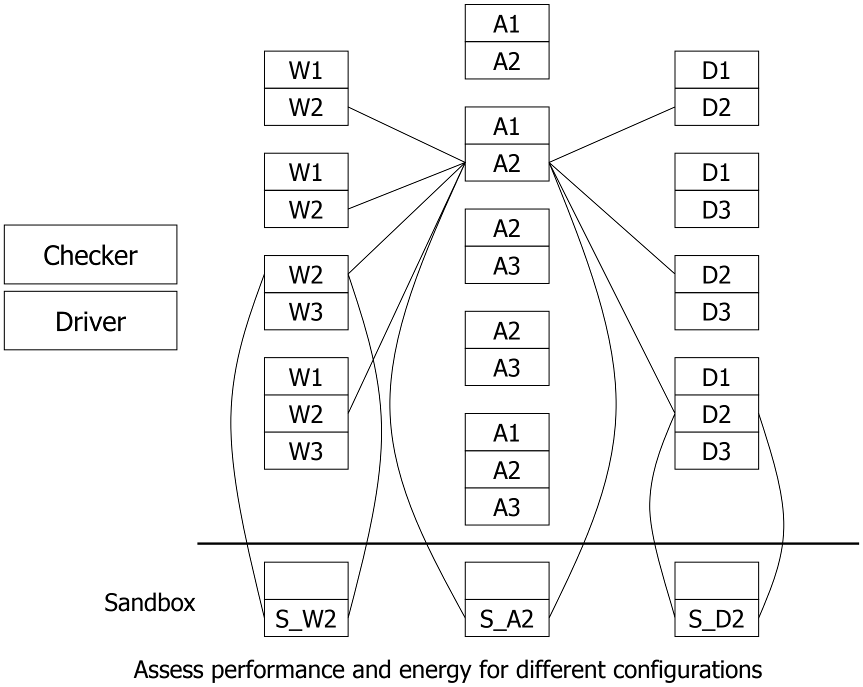

Figure 2.2: Virtualized data center and JustRunIt sandbox. Each box represents a VM, whereas each group of boxes represents a PM. “W2”, “A2”, and “D2” mean Web, application, and database server of service 2, respectively. “S_A2” means sandboxed application server of service 2.

If the management entity uses any of the interpolated performance or energy values, the *checker* invokes the experimenter to run experiments to validate those values. If it turns out that the difference between the experimental results and the interpolated results exceeds a user-specified threshold value, then the checker notifies the management entity.

We describe the design of each component of JustRunIt in detail in the following.

Experimenter.

To run experiments, the experimenter component of JustRunIt transparently clones a subset of the live production system into a sandbox and replays the live workload to the sandbox system. VM cloning instantly brings the sandbox to the same operational state as the production system, complete with fully warmed-up application-level and OS-level caches (e.g., file buffer cache). Thus, tests can proceed with low startup time on a faithful replica of the production system. By cloning only a subset of the system, JustRunIt minimizes the physical resources that must be dedicated to testing. Workload replay to the sandbox is used to emulate the timing and functional behavior of the non-duplicated portions of the system.

The use of JustRunIt in a typical virtualized data center is illustrated in Figure 2.2. The figure shows VMs of multiple three-tier services sharing each PM. Each service tier has multiple identically configured VMs placed on different PMs. (Note that VMs of one tier do not share PMs with VMs of other tiers in the figure. Although JustRunIt is agnostic to VM placement, this restriction on VM placement is often used in practice to reduce software licensing costs [53].) For simpler management, the set of PMs in each tier is often homogeneous.

The figure also shows one VM instance from each tier of service 2 being cloned into the sandbox for testing. This is just an example use of JustRunIt; we can use different numbers of PMs in the sandbox, as we discuss later. Configuration changes are applied to the clone server, and the effects of the changes are tested by replaying live traffic duplicated from the production system. The sandbox system is monitored to determine the resulting throughput, response time, and energy consumption. The experimenter reports these results to the driver to include in the matrix described in Section 2.2.2. If experiments are run with multiple service tiers, a different matrix will be created for each tier.

Although it may not be immediately obvious, the experimenter assumes that the virtual machine monitor (VMM) can provide performance isolation across VMs and includes non-work-conserving resource schedulers. These features are required because the experiments performed in the sandbox must be realistic representations of what would happen to the tested VM in the production system, regardless of any other VMs that may be co-located with it. We can see this by going back to Figure 2.2. For example, the clone VM from the application tier of service 2 must behave the same in the sandbox (where it is run alone on a PM) as it would in the production system (where it is run with A1, A3, or both), given the same configuration. Our current implementation relies on the latest version of the Xen VMM (3.3), which provides isolation for the setups that we consider.

Importantly, both performance isolation and non-work-conserving schedulers are desirable characteristics in virtualized data centers. Isolation simplifies the VM placement decisions involved in managing SLAs, whereas non-work-conserving schedulers allow more precise resource accounting and provide better isolation [53]. Most critically, both characteristics promote performance predictability, which is usually more important than achieving the best possible performance (and exceeding the SLA requirements) in hosting centers.

The experimenter can clone selected VMs of a production service (e.g., a single app server instance of a three-tier service) and use traffic replay to emulate the functional and timing behavior of the rest of the service. The infrastructure thus avoids replicating the entire system and minimizes the resources needed for testing. The infrastructure also enables multiple clones to operate concurrently to carry out multiple tests at a time.

Cloning. Cloning is accomplished by minimally extending standard VM live migration technology [21, 44]. The Xen live migration mechanism copies dirty memory pages of a running VM in the background until the number of dirty pages is reduced below a predefined threshold. Then VM execution is paused for a short time (tens of milliseconds) to copy the remaining dirty pages to the destination. Finally, execution transfers to the new VM, and the original VM is destroyed. Our cloning mechanism changes live migration to resume execution on both the new VM and the original VM.

Since cloning is transparent to the VM, the clone VM inherits the same network identity (e.g., IP/MAC addresses) as the production VM. To avoid network address conflicts, the cloning mechanism sets up network address translation to transparently give the clone VM a unique external identity exposed to the network while concealing the clone VM's internal addresses. We implemented this by extending Xen's back-end network device driver ("netback") to perform appropriate address translations and protocol checksum corrections for all network traffic to and from the clone VM.

The disk storage used by the clone VMs must also be replicated. During the short pause of the production system VM at the end of state transfer, the cloning mechanism creates a copy-on-write snapshot of the block storage volumes used by the production VM, and assigns them to the clone VM. We implemented this using the Linux LVM snapshot capability and by exporting volumes to VMs over the network using iSCSI or ATA Over Ethernet. Snapshotting and exporting the storage volumes incurs only a sub-second delay during cloning. Storage cloning is transparent to the VMs, which see logical block devices and do not know that they are accessing network storage.

JustRunIt may also be configured *not* to perform VM cloning in the sandbox. This configuration allows it to evaluate upgrades of the server software (e.g., Apache), operating system, and/or service application (as long as the application upgrade does not change the application's messaging behavior). In these cases, the management entity has to request experiments that are long enough to amortize any cold-start caching effects in the sandbox execution. However, long experiments are not a problem, since software upgrades typically do not have stringent time requirements.

Proxies. To carry out testing, the experimenter replays live workload to the VMs in the sandbox. Two low-overhead proxies, called in-proxy and out-proxy, are inserted into communication paths in the production system to replicate traffic to the sandbox. The proxies are application protocol-aware and can be (almost entirely) re-used across services that utilize the same protocols, as we detail below. The in-proxy mimics the behavior of all the previous tiers before the sandbox, and the out-proxy mimics the behavior of all the following tiers. The local view of a VM, its cloned sandbox VM, and the proxies is shown in Figure 2.3. Proxies record and reply requests and replies at application protocol-level (e.g., HTTP requests and replies). Thus, proxies are protocol-dependent; they know when requests and replies are received and if a reply

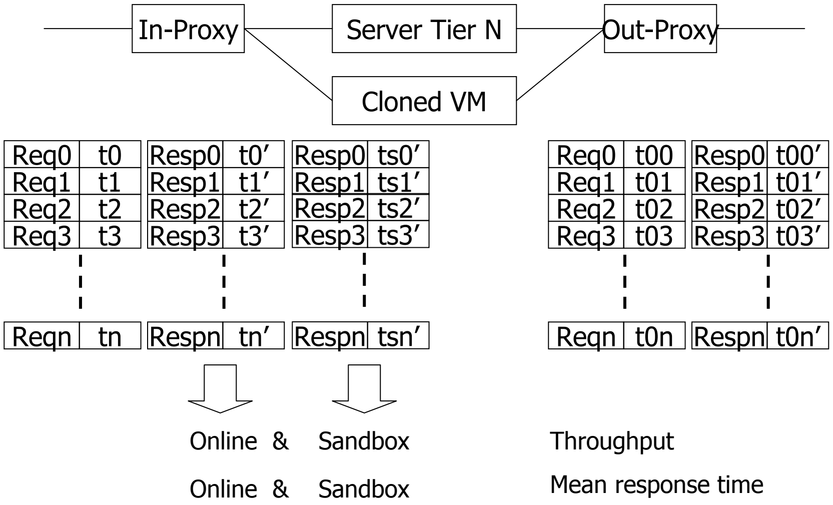

Figure 2.3: Cloned VM and proxy data structures.

is valid.

After cloning, the proxies create as many connections with the cloned VM as they have with the original VM. The connections that were open between the proxies and the original VM at the time it was cloned will timeout at the cloned VM. In fact, no requests that were active in the original VM at the time of cloning get successfully processed at the cloned VM.

The in-proxy intercepts requests from previous tiers to the tested VM. When a request arrives, the in-proxy records the request ($Reqn$ in Figure 2.3) and its arrival time (tn). The in-proxy forwards the request to the on-line production system and also sends a duplicate request to the sandbox for processing. To prevent the sandbox system from running ahead of the production system, the transmission of the duplicate request is delayed by a fixed time interval (it is sufficient for the fixed time shift to be set to any value larger than the maximum response time of the service plus the cloning overhead). Both systems process the duplicated requests and eventually generate replies that are intercepted by the in-proxy. For the reply from the production system, the in-proxy records its arrival time (Tn') and forwards the reply back to the previous tier. Later, when the corresponding reply from the sandbox arrives, the in-proxy records its arrival time (tsn'). The arrival times are used to measure the response times of the production and sandbox systems.

The production and sandbox VMs may need to send requests to the next tier to satisfy a request from the previous tier. These (duplicated) requests are intercepted by the out-proxy. The out-proxy records the arrival times ($t0n$) and content of the requests from the production system, and forwards them to the next tier. The out-proxy also records the arrival times ($t0n'$) and content of the corresponding replies, and forwards them to the production system. When the out-proxy receives a request from the sandbox system, it uses hash table lookup to find the matching request that was previously received from the production system. (Recall that the matching request will certainly have been received because the replay to the sandbox is time-shifted by more than the maximum response time of the service.) The out-proxy transmits the recorded reply to the sandbox after a delay. The delay is introduced to accurately mimic the delays of the subsequent tiers and is equal to the delay that was previously experienced by the production system ($t0n' - t0n$) for the same request-reply pair.

If a sandbox system operates faster than a production system, for example if more resources are assigned to a cloned VM, then a processed request from a sandbox system may arrive earlier than the one from a production system. In that case the out-proxy will not find a matching request. To avoid that, we use delayed replay. An in-proxy will hold all requests for some time and then replay with the same time fidelity. For example, if a user specifies a time shift for the sandbox system to be five seconds, a request that arrives to an in-proxy at time t will be duplicated and sent to the sandbox at time $t + 5$. The rest of the system works the same.

At the end of an experiment, the in-proxy reports the throughput and response time results for the production and sandbox systems. The throughput for each system is determined by the number of requests successfully served from the tiers following the in-proxy. The response time for each system is defined as the delay after a request arrives to the in-proxy until its reply is received. Since out-proxies enforce that the delays of subsequent tiers are equal for the production and sandbox system, the difference of throughput and response time between the production and sandbox systems is the performance difference between the original VM and cloned VM.

The proxies can be installed dynamically anywhere in the system, depending on which VMs the management entity may want to study at the time. However, we have only implemented in-proxies and out-proxies for Web and application servers so far. Cross-tier interactions between proxies, i.e. the communication between the out-proxy of the Web tier and the in-proxy of the application tier, occur in exactly the same way as the communication between regular servers.

We could implement an in-proxy for database servers by borrowing code from the Clustered-JDBC (C-JDBC) database middleware [13]. Briefly, C-JDBC implements a software controller between a JDBC application and a set of DBMSs. In its full-replication mode, C-JDBC keeps the content of the database replicated and consistent across the DBMSs. During experimentation, our in-proxy would do the same for the on-line and sandboxed DBMSs. Fortunately, C-JDBC already implements the key functionality needed for cloning, namely the ability to integrate the sandboxed DBMS and update its content for experimentation. To complete our in-proxy, we would modify C-JDBC to record the on-line requests and later replay them to the sandboxed DBMS. Others have modified C-JDBC in similar ways [46].

Non-determinism. A key challenge for workload replay is to tolerate non-deterministic behavior in the production and sandbox systems. We address non-determinism in three ways. First, to tolerate network layer non-determinism (e.g., packet drops) the proxies replicate application-layer requests and replies instead of replicating network packets directly.

Second, the replay is implemented so that the sandboxed servers can *process* requests and replies in a different order than their corresponding on-line servers; only the timing of the message *arrivals* at the sandboxed servers is guaranteed to reflect that of the on-line servers. This ordering flexibility tolerates non-determinism in the behavior of the software servers, e.g. due to multithreading. However, note that this flexibility is only acceptable for Web and application-tier proxies, since requests from different sessions are independent of each other in those tiers. We would need to enforce ordering more strictly in the in-proxy for database servers, to prevent the original and cloned databases from diverging. Our in-proxy would do so by forcing each write (and commit) to execute by itself *during experimentation only* forcing a complete ordering between all pairs of read-write and write-write operations; concurrent reads will be allowed to execute in any order. Others have successfully created this strict ordering in C-JDBC before [46] and saw no noticeable performance degradation for one of the services we study in this chapter.

Third, we tolerate application-layer non-determinism by designing the proxies to be application protocol-aware (e.g., the Web server in-proxies understand HTTP message formats). The proxies embody knowledge of the fields in requests and replies that can have non-deterministic values (e.g., timestamps, session IDs). When the out-proxy sees a non-deterministic value in a message from the sandbox, the message is matched against recorded messages from the production system using wildcards for the non-deterministic fields.

Our study of two services (an auction and a bookstore) shows that our proxies effectively tolerate their non-determinism. Even though some messages in these services have identical values except for a non-deterministic field, our wildcard mechanism allows JustRunIt to properly match replies in the production and sandbox systems for two reasons. First, all replies from the sandbox are dropped by the proxies, preventing them from disrupting the on-line system. Second, using different replies due to wildcard mismatch does not affect the JustRunIt results because the replies are equivalent and all delays are still accounted for.

Despite our promising experience with the auction and bookstore services, some types of non-determinism may be hard for our proxies to handle. In particular, services that non-deterministically change their messaging behavior (not just particular fields or the destination of the messages) or their load processing behavior (e.g., via non-deterministic load-shedding) would be impossible to handle. For example, a service in which servers may send an unpredictable number of messages in response to each request cannot be handled by our proxies. We have not come across any such services, though.

Experiment Driver.

Running experiments is not free. They cost time and energy. For this reason, JustRunIt allows the management entity to configure the experimentation using a simple configuration file. The entity can specify the tier(s) with which JustRunIt should experiment, which experiment heuristics to apply (discussed below), which resources to vary, the range of resource allocations to consider, how many equally separated allocation points to consider in the range, how long each experiment should take, and how many experiments to run. These parameters can directly limit the time and indirectly limit the energy consumed by the experiments, when there are constraints on these resources (as in Section 2.3.1). When experiment time and energy are not relevant constraints (as in Section 2.3.2), the settings for the parameters can be looser.

The management entity provides the experiment driver with a configuration file that specifies experiment heuristics, configuration coordinate ranges and granularities, and experiment time limits.

Based on the configuration information, the experiment driver directs the experimenter to explore the parameter space within the time limit. The driver starts by running experiments to fill in the entries at the corners of the result matrix. For example, if the experiments should vary the CPU allocation and the CPU frequency, the matrix will have two dimensions and four corners: (min CPU alloc, min CPU freq), (min CPU alloc, max CPU freq), (max CPU alloc, min CPU freq), and (max CPU alloc, max CPU freq). The management entity must configure JustRunIt so at least these corner experiments can be performed. After filling in the corner coordinates, the driver then proceeds to request experiments exactly in the middle of the unexplored ranges defined by each resource dimension. After those are performed, it recursively sub-divides the unexplored ranges in turn. This process is repeated until the number of experiments requested by the management entity have been performed or there are no more experiments to perform.

It then proceeds to run experiments along each resource dimension, starting with low allocation and increasing by increments of one-tenth the coordinate range (so that in nine additional experiments the range would be covered).

We designed two heuristics for the driver to use to avoid running unnecessary experiments along each matrix dimension. The two observations behind the heuristics are that: 1) beyond some point, resource additions do not improve performance; 2) the performance gain for the same resource addition to different tiers will not be the same, and the gains drop consistently and continually (diminishing returns).

Based on observation 1), the first heuristic cancels the remaining experiments with larger resource allocations along the current resource dimension, if the performance gain from a resource addition is less than a threshold amount. Based on observation 2), the second heuristic cancels the experiments with tiers that do not produce the largest gains from a resource addition. As we add more resources to the current tier, the performance gains decrease until some other tier becomes the tier with the largest gain from the same resource addition. For example, increasing the CPU allocation on the bottleneck tier, say the application tier, will significantly improve overall response time. At some point, however, the bottleneck will shift to other tiers, say the Web tier, at which point the driver will experiment with the Web tier and gain more overall response time improvement with the same CPU addition. Both heuristics cut down the number of experiments required while still finding the minimum CPU addition that provides the maximum performance gain.

Interpolator and Checker.

The interpolator predicts performance results for points in the matrix that have not yet been determined through experiments. For simplicity, we use linear interpolation to fill in these blanks, and we mark the values to indicate that they are just interpolated.

If the management entity uses any interpolated results, the checker tries to verify the interpolated results by invoking the experimenter to run the corresponding experiments in the background. If one of these background experimental results differs from the corresponding interpolated result by more than a user-specified threshold value, the checker raises a flag to the management entity to decide how to handle this mismatch.

The management entity can use this information in multiple ways. For example, it may reconfigure the driver to run more experiments with the corresponding resources from now on. Another option would be to reconfigure the range of allocations to consider in the experiments from now on.

2.2.3 Discussion

Uses of JustRunIt. We expect that JustRunIt will be useful for many system management scenarios. For example, in this chapter we consider resource management and hardware upgrade case studies. In these and other scenarios, JustRunIt can be used by the management entity to safely, efficiently, and realistically answer the same “what-if” questions that modeling can answer given the current workload and load intensity.

Moreover, like modeling, JustRunIt can benefit from load intensity prediction techniques to answer questions about future scenarios. JustRunIt can do so because its request replay is shifted in time and can be done at any desired speed. (Request stream acceleration needs to consider whether requests belong to an existing session or start a new session. JustRunIt can properly accelerate requests because it stores enough information about them to differentiate between the two cases.) Section 4.5 discusses how the current version of JustRunIt can be modified to answer “what-if” questions about different workload mixes as well.

Although our current implementation does not implement this functionality, JustRunIt could also be used to select the best values for software tunables, e.g. the number of threads or the size of the memory cache in Web servers. Modeling does not lend itself directly to this type of management task. Another possible extension could be enabling JustRunIt to evaluate the correctness of administrator actions, as in action-validation systems [41, 46]. All the key infrastructure required by these systems (i.e., proxies, cloning, sandboxing) is already part of the current version of JustRunIt, so adding the ability to validate administrator actions should be a simple exercise. Interestingly, this type of functionality cannot be provided by analytic models or feedback control.

Obviously, JustRunIt can answer questions and validate administrator actions at the cost of experiment time and energy. However, note that the physical resources required by JustRunIt (i.e., enough computational resources for the proxies and for the sandbox) can be a very small fraction of the data center's resources. For example, in Figure 2.2, we show that just three PMs are enough to experiment with all tiers of a service at the same time, regardless of how large the production system is. Even fewer resources, e.g. one PM, can be used, as long as we have the time to experiment with VMs sequentially. Furthermore, the JustRunIt physical resources can be borrowed from the production system itself, e.g. during periods of low load.

In essence, JustRunIt poses an interesting tradeoff between the amount of physical resources it uses, the experiment time that needs to elapse before decisions can be made, and the energy consumed by its resources. More physical resources translate into shorter experiment times but higher energy consumption. For this reason, we allow the management entity to configure JustRunIt in whatever way is appropriate for the data center.

Engineering cost of JustRunIt. Building the JustRunIt proxies is the most time-consuming part of its implementation. The proxies must be designed to properly handle the communication protocols used by services. Our current proxies understand the

HTTP, mod_jk, and MySQL protocols. We have built our proxies starting from the publicly available Tinyproxy HTTP proxy daemon [7]. Each proxy required only between 800 and 1500 new lines of C code. (VM cloning required 42 new lines of Python code in the xend control daemon and the xm management tool, whereas address translation required 244 new lines of C code in the netback driver.) The vast majority of the difference between Web and application server proxies comes from their different communication protocols.

The engineering effort required by the proxies can be amortized, as they can be reused for any service based on the same protocols. However, the proxies may need modifications to handle any non-determinism in the services themselves. Fortunately, our experience with the auction and bookstore services suggests that the effort involved in handling service-level non-determinism may be small. Specifically, it took us less than one day to adapt the proxies designed for the auction to the bookstore. This is particularly promising in that he had no prior knowledge of the bookstore whatsoever.

One may argue that implementing JustRunIt may require a comparable amount of effort to developing accurate models for a service. However, *we note that JustRunIt is much more reusable than models, across different services, hardware and software characteristics, and even as service behavior evolves.* Each of these factors requires model re-calibration and re-validation, which are typically labor-intensive. Furthermore, for models to become tractable, many simplifying assumptions about system behavior (e.g., memoryless request arrivals) may have to be made. These assumptions may compromise the accuracy of the models. JustRunIt does not require these assumptions and produces accurate results.

2.3 Experiment-based Management

As mentioned in the previous section, our infrastructure can be used by automated management systems or directly by the system administrator. To demonstrate its use in the former scenario, we have implemented simple automated management systems for two common tasks in virtualized hosting centers: server consolidation/expansion (i.e., partitioning resources across the services to use as few active servers as possible) and evaluation of hardware upgrades. These tasks are currently performed by most administrators in a manual, labor-intensive, and ad-hoc manner.

Both management systems seek to satisfy the services' SLAs. An SLA often specifies a percentage of requests to be serviced within some amount of time. Another possibility is for the SLA to specify an average response time (over a period of several minutes) for the corresponding service. For simplicity, our automated systems assume the latter type of SLA.

The next two subsections describe the management systems. However, before describing them, we note that they are *not* contributions of this work. Rather, they are presented simply to demonstrate the automated use of JustRunIt. More sophisticated systems (or the administrator) would leverage JustRunIt in similar ways.

2.3.1 Case Study 1: Resource Management

Overview. The ultimate goal of our resource-management system is to consolidate the hosted services onto the smallest possible set of nodes, while satisfying all SLAs. To achieve this goal, the system constantly monitors the average response time of each service, comparing this average to the corresponding SLA. Because workload conditions change over time, the resources assigned to a service may become insufficient and the service may start violating its SLA. Whenever such a violation occurs, our system initiates experiments with JustRunIt to determine what is the minimum allocation of resources that would be required for the service's SLA to be satisfied again. Changes in workload behavior often occur at the granularity of tens of minutes or even hours, suggesting that the time spent performing experiments is likely to be relatively small. Nevertheless, to avoid having to perform adjustments too frequently, the system assigns 20% more resources to a service than its minimum needs. This slack allows

```
 1. While 1 do
 2.   Monitor QoS of all services
 3.   If any service needs more resources or
 4.                  can use fewer resources
 5.      Run experiments with bottleneck tier
 6.      Find minimum resource needs
 7.      If used any interpolated results
 8.         Inform JustRunIt about them
 9.      Assign resources using bin-packing heuristic
10.      If new nodes need to be added
11.         Add new nodes and migrate VMs to them
12.      Else if nodes can be removed
13.              Migrate VMs and remove nodes
14.      Complete resource adjustments and migrations
```

Figure 2.4: Overview of resource-management system.

for transient increases in offered load without excessive resource waste. Since the resources required by the service have to be allocated to it, the new resource allocation may require VM migrations or even the use of extra nodes.

Conversely, when the SLA of any service is being satisfied by more than a threshold amount (i.e., the average response time is lower than that specified by the SLA by more than a threshold percentage), our system considers the possibility of reducing the amount of resources dedicated to the service. It does so by initiating experiments with JustRunIt to determine the minimum allocation of resources that would still satisfy the service's SLA. Again, we give the service additional slack in its resource allocation to avoid frequent reallocations. Because resources can be taken away from this service, the new combined resource needs of the services may not require as many PMs. In this case, the system determines the minimum number of PMs that can be used and implements the required VM migrations.

Details. Figure 2.4 presents pseudo-code overviewing the operation of our management system. The experiments with JustRunIt are performed in line 5. The management system only runs experiments with one software server of the bottleneck tier of the service in question. The management system can determine the bottleneck tier by inspecting the resource utilization of the servers in each tier. Experimenting with one software server is typically enough for two reasons: (1) services typically balance the load evenly across the servers of each tier; and (2) the VMs of all software servers of the same tier and service are assigned the same amount of resources at their PMs. (When at least one of these two properties does not hold, the management system needs to request more experiments of JustRunIt.) However, if enough nodes can be used for experiments in the sandbox, the system could run experiments with one software server from each tier of the service at the same time.

The matrix of resource allocations vs. response times produced by JustRunIt is then used to find the minimum resource needs of the service in line 6. Specifically, the management system checks the results in the JustRunIt matrix (from smallest to largest resource allocation) to find the minimum allocation that would still allow the SLA to be satisfied. In lines 7 and 8, the system informs JustRunIt about any interpolated results that it may have used in determining the minimum resource needs. JustRunIt will inform the management system if the interpolated results are different than the actual experimental results by more than a configurable threshold amount.

In line 9, the system executes a resource assignment algorithm that will determine the VM to PM assignment for all VMs of all services. We model resource assignment as a bin-packing problem. In bin-packing, the goal is to place a number of objects into bins, so that we minimize the number of bins. We model the VMs (and their resource requirements) as the objects and the PMs (and their available resources) as the bins. If more than one VM to PM assignment leads to the minimum number of PMs, we break the tie by selecting the optimal assignment that requires the smallest number of migrations. If more than one assignment requires the smallest number of migrations, we pick the one of these assignments randomly. Unfortunately, the bin-packing problem is NP-complete, so it can take an inordinate amount of time to solve it optimally, even for hosting centers of moderate size. Thus, we resort to a heuristic approach, namely simulated annealing [38], to solve it.

Simulated annealing works by trying to iteratively optimize a “current solution”, i.e. a particular assignment of VMs to PMs, starting from an initial guess assignment. Each assignment is generated as follows. We create an ordered list of VMs and an ordered list of PMs. Starting from the front of the VM list, we assign VMs to the first PM until no more VMs can be added. (Note that fully packing a PM is not a problem because each VM already includes 20% slack in the resources it is assigned.) At that point, we move on to the next PM and fill it (again taking VMs from the ordered list), and so on. The only additional constraint to this placement approach is the placement restrictions we mentioned in Section 2.2.1.

The initial guess assignment is trivially created based on the current VM to PM assignment. Each other “candidate solution” is created by randomly swapping two VMs in the ordered VM list and placing the VMs on the PMs as described above. A candidate solution becomes the new current solution in two cases: when it produces a smaller number of PMs than the current solution, or when it produces the same number of PMs but with a smaller number of migrations. If a candidate solution does not fit these two cases, it might still become the new current solution, but with a decreasing probability. Accepting a solution that is worse than the current solution allows the algorithm to skip out of local minima. After evaluating each candidate solution, a new one is generated and the process is repeated. The number of iterations and the probability of accepting a relatively poor candidate solution is determined by a “temperature” parameter to the annealing algorithm. More details about simulated annealing can be found in [38].

Finally, in lines 10–14, the resource-allocation system adjusts the number of PMs and the VM to PM assignment as determined by the best solution ever seen by simulated annealing.

Comparison. A model-based implementation for this management system would be similar; it would simply replace lines 5–8 with a call to a performance model solver.

The solver would use information about the current state of the system (e.g., the offered load, resource allocation, and the tier response times) and the analytical model to estimate the minimal resource requirements for the service in question. Obviously, the model would have to have been created, calibrated, and validated *a priori*.

A feedback-based implementation would replace lines 5–8 by a call to the controller to execute the experiments that will adjust the offending service. However, note that feedback control is only applicable when repeatedly varying the allocation of a resource or changing a hardware setting does not affect the on-line behavior of the co-located services. For example, we can use feedback control to vary the CPU allocation of a service without affecting other services. In contrast, increasing the amount of memory allocated to a service may require decreasing the allocation of another service. Similarly, varying the voltage setting for a service affects all services running on the same CPU chip, because the cores in current chips share the same voltage rail. Cross-service interactions are clearly undesirable, especially when they may occur repeatedly as in feedback control. The key problem is that feedback control experiments with the on-line system. With JustRunIt, bin-packing and node addition/removal occur before any resource changes are made on-line, so interference can be completely avoided in most cases. When interference is unavoidable, e.g. the offending service cannot be migrated to a node with enough available memory and no extra nodes can be added, changes to the service are made only once.

2.3.2 Case Study 2: Hardware Upgrades

Overview. For our second case study, we built a management system to evaluate hardware upgrades. The system assumes that at least one instance of the hardware being considered is available for experimentation in the sandbox. For example, consider a scenario in which the hosting center is considering purchasing machines of a model that is faster or has more available resources than that of its current machines. After

```
1. For each service do
2.   For one software server of each tier
3.      Run experiments with JustRunIt
4.      Find minimum resource needs
5. If used any interpolated results
6.    Inform JustRunIt about them
7. Assign resources using bin-packing heuristic
8. Estimate power consumption
```

Figure 2.5: Overview of update-evaluation system.

performing experiments with a single machine of the candidate model, our system determines whether the upgrade would allow servers to be consolidated onto a smaller number of machines and whether the overall power consumption of the hosting center would be smaller than it currently is. This information is provided to the administrator, who can make a final decision on whether or not to purchase the new machines and ultimately perform the upgrade.

Details. Figure 2.5 presents pseudo-code overviewing our update-evaluation system. The experiments with JustRunIt are started in line 3. For this system, the matrix that JustRunIt produces must include information about the average response time and the average power consumption of each resource allocation on the upgrade-candidate machine. In line 4, the system determines the resource allocation that achieves the same average response time as on the current machine (thus guaranteeing that the SLA would be satisfied by the candidate machine as well). Again, the administrator configures the system to properly drive JustRunIt and gets informed about any interpolated results that are used in line 4.

By adding the extra 20% slack to these minimum requirements and running the bin-packing algorithm described above, the system determines how many new machines would be required to achieve the current performance and how much power the entire center would consume. Specifically, the center power can be estimated by adding up the power consumption of each PM in the resource assignment produced by the simulated annealing. The consumption of each PM can be estimated by first determining the “base” power of the candidate machine, i.e. the power consumption when the machine is on but no VM is running on it. This base power should be subtracted from the results in the JustRunIt matrix of each software server VM. This subtraction produces the average dynamic power required by the VM. Estimating the power of each PM then involves adding up the dynamic power consumption of the VMs that would run on the PM plus the base power.

Comparison. Modeling has been used for this management task [19]. A modeling-based implementation for our management system would replace lines 2–6 in Figure 2.5 with a call to a performance model solver to estimate the minimum resource requirements for each service. Based on these results and on the resource assignment computed in line 7, an energy model would estimate the energy consumption in line 8. Again, both models would have to have been created, calibrated, and validated *a priori*. In contrast, feedback control is not applicable to this management task.

2.4 Evaluation

In this section, we describe our evaluation methodology, evaluate the overhead of JustRunIt, and illustrate its behavior in our two automated-management case studies.

2.4.1 Methodology

Our hardware comprises 15 HP Proliant C-class blades interconnected by a Gigabit Ethernet switch. Each server has 8 GBytes of DRAM, 2 hard disks, and 2 Intel dual-core Xeon CPUs. Each CPU has two frequency points, 2 GHz and 3 GHz. Two blades with direct-attached disks are used as network-attached storage servers. They export Linux LVM logical volumes to the other blades using ATA over Ethernet. One Gbit Ethernet port of every blade is used exclusively for network storage traffic. We measure the energy consumed by a blade by querying its management processor, which monitors the peak and average power usage of the entire blade.

Virtualization is provided by XenLinux kernel 2.6.18 with the Xen VMM [8], version 3.3. For improving Xen's ability to provide performance isolation, we pin Dom0 to one of the cores and isolate the service(s) from it. Note, however, that JustRunIt does not itself impose this organization. As JustRunIt only depends on the VMM for VM cloning, it can easily be ported to use VMMs that do not perform I/O in a separate VM.

We populate the blade cluster with one or more independent instances of an on-line auction service. To demonstrate the generality of our system, we also experiment with an on-line bookstore. Both services are organized into three tiers of servers: Web, application, and database tiers. The first tier is implemented by Apache Web servers (version 2.0.54), the second tier uses Tomcat servlet servers (version 4.1.18), and the third tier uses the MySQL relational database (version 5.0.27). (For performance reasons, the database servers are not virtualized and run directly on Linux and the underlying hardware.) We use LVS load balancers [85] in front of the Web and application tiers. 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 each instance of the services using a client emulator. The auction 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. The bookstore workload comprises a "shopping mix", where 20% of the requests are read-write. 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 services, their workloads, and the client emulator are from the DynaServer project [57] and have been used extensively by other research groups.

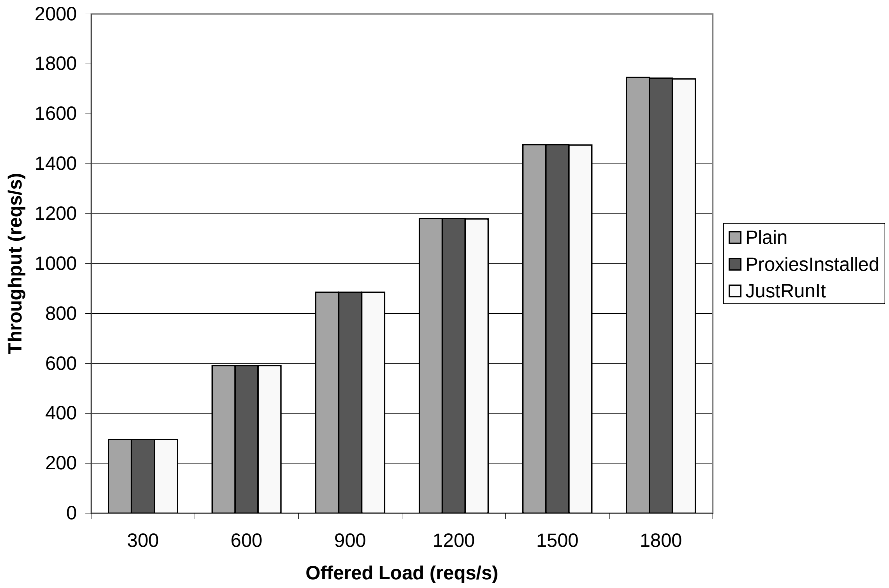

Figure 2.6: Throughput as a function of offered load.

2.4.2 JustRunIt Overhead

Our overhead evaluation seeks to answer two questions: (1) Does the overhead of JustRunIt (proxies, VM cloning, workload duplication, and reply matching) degrade the performance of the on-line services? and (2) How faithfully do servers in the sandbox represent on-line servers given the same resources?

To answer these questions, we use our auction service as implemented by one Apache VM, one Tomcat VM, and MySQL. Using a larger instance of the service would hide some of the overhead of JustRunIt, since the proxies only instrument one path through the service. Each of the VMs runs on a different blade. We use one blade in the sandbox. The two proxies for the Web tier run on one of the blades, whereas those for the application tier run on another. The proxies run on their own blades to promote performance isolation for the auction service. In all our experiments, the time shift used by JustRunIt is 10 seconds behind the on-line service.

Overhead on the on-line system? To isolate the overhead of JustRunIt on the on-line service, we experiment with three scenarios: (1) Plain – no proxies are installed; (2)

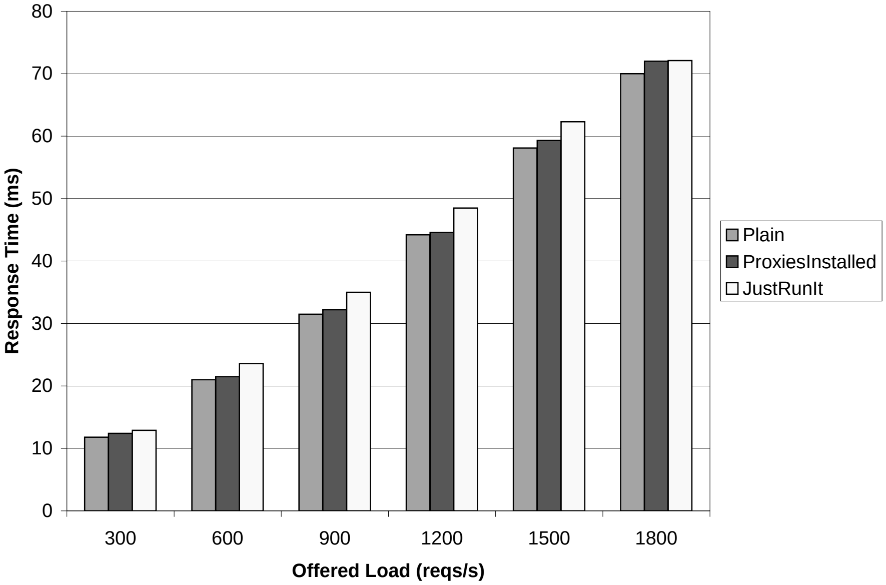

Figure 2.7: Response time as a function of offered load.

ProxiesInstalled – proxies are installed around the Web and application servers, but they only relay the network traffic; and (3) JustRunIt – proxies are installed around the Web and application servers and perform all the JustRunIt functionality.

Figures 2.6 and 2.7 depict the average throughput and response time of the on-line service, respectively, as a function of the offered load. We set the CPU allocation of all servers to 100% of one core. In this configuration, the service saturates at 1940 requests/second. Each bar corresponds to a 200-second execution.

Figure 2.6 shows that JustRunIt has no effect on the throughput of the on-line service, even as it approaches saturation, despite having the proxies for each tier co-located on the same blade. In fact, the resource utilizations suggest that the service itself (its application server) would saturate well before JustRunIt. Specifically, the application server is more than twice as utilized as any JustRunIt proxy.

Figure 2.7 shows that the overhead of JustRunIt is consistently small ($< 5ms$) across load intensities. We could further optimizing the implementation to reduce the JustRunIt overheads further. However, remember that the overheads in Figure 2.7 are exaggerated by the fact that, in these experiments, *all* application server requests are

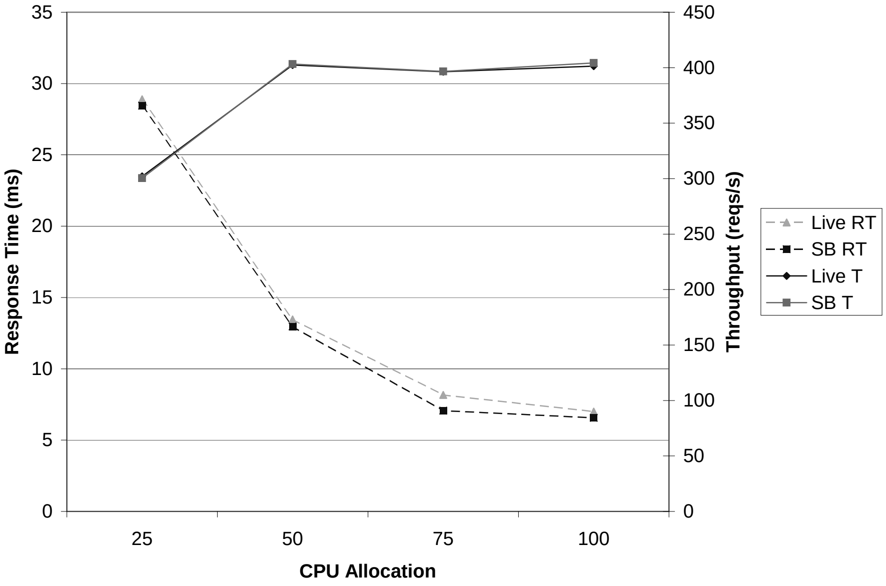

Figure 2.8: On-line and sandboxed performance as a function of CPU allocation at offered load 500 requests/second.

exposed to the JustRunIt instrumentation. If we had used a service with 4 application servers, for example, only roughly 25% of those requests would be exposed to the instrumentation (since we only need proxies for 1 of the application servers), thus lowering the average overhead by 75%.

Performance in the sandbox? The results above isolate the overhead of JustRunIt on the on-line system. However, another important consideration is how faithful the sandbox execution is to the on-line execution given the same resources. Obviously, it would be inaccurate to make management decisions based on sandboxed experiments that are not very similar to the behavior of the on-line system.

Figures 2.8 and 2.9 compare the performance of the on-line application server (labeled "Live") to that of the sandboxed (labeled "SB") application server at 500 requests/second and 1000 requests/second, respectively. In both figures, response times (labeled "RT") and throughputs (labeled "T") are measured at the application server's in-proxy. Again, each result represents the average performance over 200 seconds.

As one would expect, the figures show that increasing the CPU allocation tends to

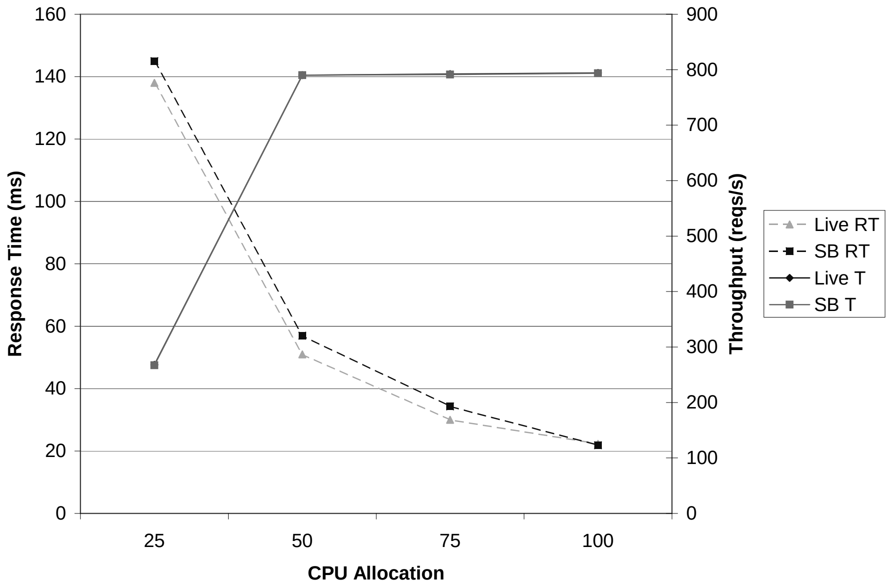

Figure 2.9: On-line and sandboxed performance as a function of CPU allocation at offered load 1000 requests/second.

increase throughputs and reduce response times. The difference between the offered load and the achieved throughput is the 20% of requests that are served directly by the Web server and, thus, do not reach the application server's in-proxy. More interestingly, the figures clearly show that the sandboxed execution is a faithful representation of the on-line system, regardless of the offered load.

The results for the Web tier also show the sandboxed execution to be accurate. Like the application-tier results, we ran experiments with four different CPU allocations, under two offered loads. When the offered load is 500 reqs/s, the average difference between the on-line and sandboxed results is 4 requests/second for throughput and 1 ms for response time, across all CPU allocations. Even under a load of 1000 requests/second, the average throughput and response time differences are only 6 requests/second and 2 ms, respectively.

Our experiments with the bookstore service exhibit the same behaviors as in Figures 2.6 to 2.9. The throughput is not affected by JustRunIt and the overhead on the response time is small. For example, under an offered load of 300 requests/second,

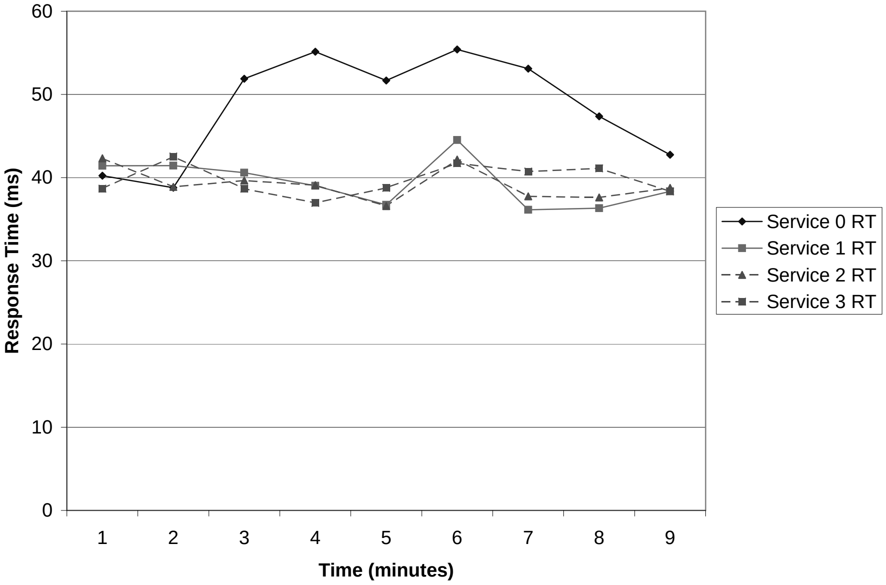

Figure 2.10: Server expansion using JustRunIt.

JustRunIt increases the mean response time for the bookstore from 18 ms to 22 ms. For 900 requests/second, the increase is from 54 ms to 58 ms. Finally, our worst result shows that JustRunIt increases the mean response time from 90 ms to 100 ms at 1500 requests/second.

2.4.3 Case Study 1: Resource Management

As mentioned before, we built an automated resource manager for a virtualized hosting center that leverages JustRunIt. To demonstrate the behavior of our manager, we created four instances of our auction service on 9 blades: 2 blades for first-tier servers, 2 blades for second-tier servers, 2 blades for database servers, and 3 blades for storage servers and LVS. Each first-tier (second-tier) blade runs one Web (application) server from each service. Each server VM is allocated 50% of one core as its CPU allocation. We assume that the services' SLAs require an average response time lower than 50 ms in every period of one minute. The manager requested JustRunIt to run 3 CPU-allocation experiments with any service that violated its SLA, for no longer than

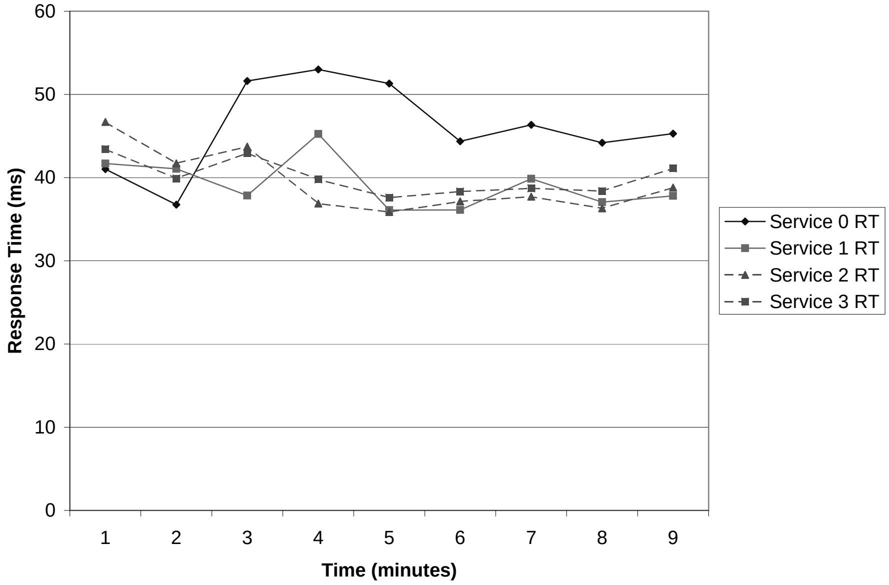

Figure 2.11: Server expansion using accurate modeling.

3 minutes overall. A 10th blade is used for the JustRunIt sandbox, whereas 2 extra blades are used for its Web and application-server proxies. Finally, 2 more blades are used to generate load.

Figure 2.10 shows the response time of each service during our experiment; each point represents the average response time during the corresponding minute. We initially offered 1000 requests/second to each service. This offered load results in an average response time hovering around 40 ms. Two minutes after the start of the experiment, we increase the load offered to service 0 to 1500 requests/second. This caused its response time to increase beyond 50 ms during the third minute of the experiment. At that point, the manager started JustRunIt experiments to determine the CPU allocation that would be required for the service's application servers (the second tier is the bottleneck tier) to bring response time back below 50 ms under the new offered load. The set of JustRunIt experiments lasted 3 minutes, allowing CPU allocations of 60%, 80%, and 100% of a core to be tested. The values for 70% and 90% shares were interpolated based on the experimental results.

Based on the response-time results of the experiments, the manager determined that the application server VMs of the offending service should be given 72% of a core (i.e., 60% of a core plus the 20% of 60% = 12% slack). Because of the extra CPU allocation requirements, the manager decided that the system should be expanded to include an additional PM (a 15th blade in our setup). To populate this machine, the manager migrated 2 VMs to it (one from each PM hosting application server VMs). Besides the 3 minutes spent with experiments, VM cloning, simulated annealing, and VM migration took about 1 minute altogether. As a result, the manager was able to complete the resource reallocation 7 minutes into the experiment. The experiment ended with all services satisfying their SLAs.

Comparison against highly accurate modeling. Figure 2.11 shows what the system behavior would be if the resource manager made its decisions based on a highly accurate response-time model of our 3-tier auction service. To mimic such a model, we performed the JustRunIt experiments with service 0 under the same offered load of Figure 2.10 for all CPU allocations off-line. These off-line results were fed to the manager during the experiment free of any overheads. We assumed that the model-based manager would require 1 minute of resource-usage monitoring after the SLA violation is detected, before the model could be solved. Based on the JustRunIt results, the manager made the same decisions as in Figure 2.10.

The figure shows that modeling would allow the system to adjust 2 minutes faster. However, developing, calibrating, and validating such an accurate model is a challenging and labor-intensive proposition. Furthermore, adaptations would happen relatively infrequently in practice, given that (1) it typically takes at least tens of minutes for load intensities to increase significantly in real systems, and (2) the manager builds slack into the resource allocation during each adaptation. In summary, the small delay in decision making and the limited resources that JustRunIt requires are a small price to pay for the benefits that it affords.

2.4.4 Case Study 2: Hardware Upgrade

We also experimented with our automated system for evaluating hardware upgrades in a virtualized hosting center. To demonstrate the behavior of our system, we ran two instances of our auction service on the same number of blades as in our resource manager study above. However, we now configure the blades that run the services to run at 2 GHz. The blade in the JustRunIt sandbox is set to run at 3 GHz to mimic a more powerful machine that we are considering for an upgrade of the data center. We offer 1000 requests/second to each service. We also cap each application server VM of both services at 90% of one core; for simplicity, we do not experiment with the Web tier, but the same approach could be trivially taken for it as well.

During the experiment, the management system requested JustRunIt to run 4 CPU-allocation experiments for no longer than 800 seconds overall. (Note, though, that this type of management task does not have real-time requirements, so we can afford to run JustRunIt experiments for a much longer time.) Since each server is initially allocated 90% of one core, JustRunIt is told to experiment with CPU allocations of 50%, 60%, 70%, and 80% of one core; there is no need for interpolation. The management system's main goal is to determine (using simulated annealing) how many of the new machines would be needed to achieve the same response time that the services currently exhibit. With this information, the energy implications of the upgrade can be assessed.

Based on the results generated by JustRunIt, the management system decided that the VMs of both services could each run at 72% CPU allocations (60% of one core plus 12% slack) at 3 GHz. For a large data center with diverse services, a similar reduction in resource requirements may allow for servers to be consolidated, which would most likely conserve energy. Unfortunately, our experimental system is too small to demonstrate these effects here.

Again, an analytic model could have made the same decisions here but at the cost of greater complexity, worse accuracy, and intense human labor.

2.4.5 Discussion

The results above demonstrate that the JustRunIt overhead is small, even when all requests are exposed to our instrumentation. In real deployments, the observed overhead will be even smaller, since there will certainly be more than one path through each service (at the very least to guarantee availability and fault-tolerance). Furthermore, the results show that the sandboxed execution is faithful to the on-line execution. Finally, the results demonstrate that JustRunIt can be effectively leveraged to implement sophisticated automated management systems. Modeling could have been applied to the two systems, whereas feedback control is applicable to resource management (in the case of the CPU allocation), but not upgrade evaluation. The hardware resources consumed by JustRunIt amount to one machine for the two proxies of each tier, plus as few as one sandbox machine. Most importantly, *this overhead is fixed and independent of the size of the production system*.

2.5 Summary

This chapter introduced a novel infrastructure for experiment-based management of virtualized data centers, called JustRunIt. The infrastructure enables an automated management system or the system administrator to answer “what-if” questions experimentally during management tasks and, based on the answers, select the best course of action. The current version of JustRunIt can be applied to many management tasks, including resource management, hardware upgrades, and software upgrades.

JustRunIt leverages virtualization technology, a small amount of hardware placed in a sandbox, and real on-line workload duplication to perform experiments. To demonstrate JustRunIt, we evaluated it in the context of two systems that automate common management tasks. Our results showed that JustRunIt can be nicely combined with automated management systems and produces results realistically and transparently. In fact, JustRunIt produces system configurations that are as good as those resulting from idealized, perfectly accurate models, at the cost of the time and energy consumed by the experiments. Furthermore, JustRunIt is more general than feedback control and machine learning in its ability to support many management tasks.

Limitations. There are three types of "what-if" questions that sophisticated models can answer (by making simplifying assumptions and costing extensive human labor), whereas JustRunIt currently cannot. First, service-wide models can answer questions about the effect of a service tier on other tiers. In the current version of JustRunIt, these cross-tier interactions are not visible, since the sandboxed virtual machines do not communicate with each other.

Second, models that represent request mixes at a low enough level can answer questions about hypothetical mixes that have not been experienced in practice. Currently, JustRunIt relies solely on real workload duplication for its experiments, so it can only answer questions about request mixes that are offered to the system. Nevertheless, JustRunIt *can* currently answer questions about more or less intense versions of real workloads, which seems to be a more useful property.

Finally, models can sometimes be used to spot performance anomalies, although differences between model results and on-line behavior are often due to inaccuracies of the model. Because JustRunIt uses complete-state replicas of on-line virtual machines for greater realism in its experiments, anomalies due to software server or operating system bugs cannot be detected.

Nevertheless, JustRunIt can be used to detect a range of improper or incorrect administrator actions (performed first in the sandbox for testing), including those that cause software servers to change message contents incorrectly. A similar approach has been taken in [41, 46]. Modeling can only deal with these scenarios if they at the same time affect performance by more than the expected modeling inaccuracy.

Future work. JustRunIt could be extended to allow cross-tier communication between the sandboxed servers. This would allow the administrator to configure sandboxing with or without cross-tier interactions. We could also extend JustRunIt to create infrastructure to allow experimentation with request mixes other than those observed on-line. The idea here is to collect a trace of the on-line workload offered to one server of each tier, as well as the state of these servers. Later, JustRunIt could install the states and replay the trace to the sandboxed servers. During replay, the request mix could be changed by eliminating or replicating some of the traced sessions. Finally, JustRunIt could be extended to include an in-proxy for a database server, starting with code from the C-JDBC middleware.

Chapter 3

Automatic Configuration Infrastructure

3.1 Introduction

Recent research has found that operator mistakes are a significant source of availability, performance, and security problems in cluster-based Internet services [41, 48]. Moreover, these studies found that misconfigurations are the most common type of operator mistake. For example, Oppenheimer and Patterson [48] 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.* [41] 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 [49], as in [14, 39, 47].

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 chapter, 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 parameters. 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 [41]. Using their operator action logs and a previously proposed approach to quantifying configuration complexity [11], we find that our infrastructure can simplify the service operation significantly while eliminating 58% of the misconfigurations found in [41]. 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. [2, 3, 12]. 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

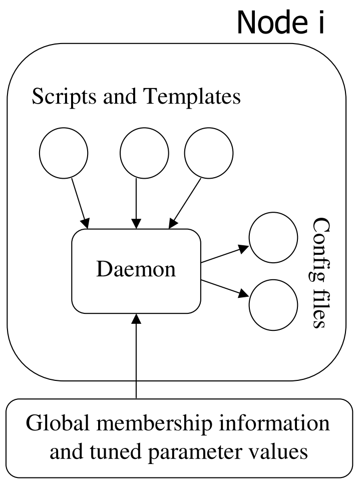

Figure 3.1: Overview of each node.

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.

The remainder of this chapter is organized as follows. Sections 3.2 and 3.3 describe our infrastructure in detail. Section 3.4 describes our experimental setup and discusses our main results. Section 3.5 summarizes our findings.

3.2 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 3.1 and 3.2, 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 run-time 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).

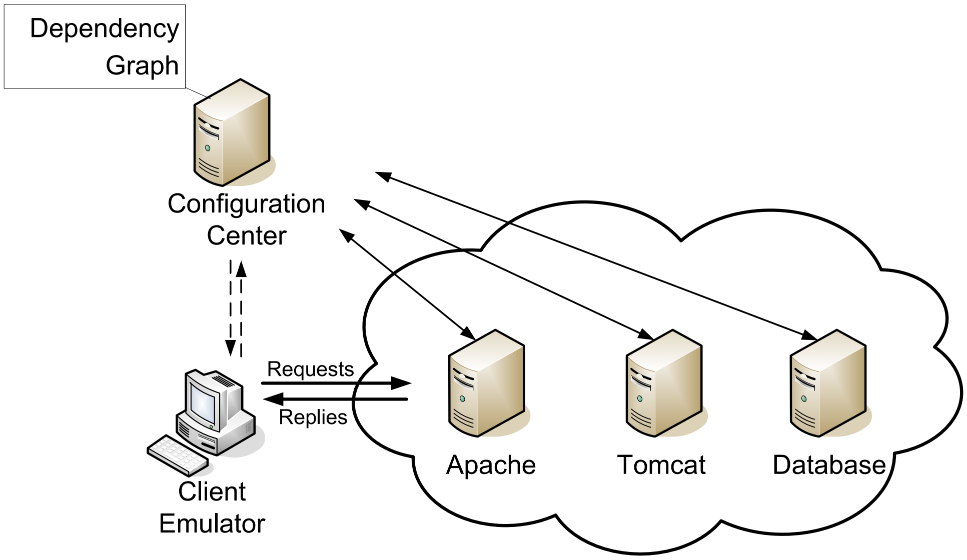

Figure 3.2: Overview of the entire system. For simplicity, we only show one server node per service tier.

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 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

```
# File worker.properties.tmp

# List workers by name
worker.list = [WORKER_LIST]loadbalancer

# Describe properties of each worker
[WORKER_PROPERTIES]

# Describe properties of load balancer
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=[WORKERS]
```

Figure 3.3: Template for worker.properties file.

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 3.3 and 3.4. Figure 3.3 shows a very simple example template for the worker.properties file used by the `mod_jk`

```
# Global section

$TEMPLATE = "worker.properties.tmp";
$COMMENT_CHAR = "#";

# Program section

# Generate list of tier-2 application servers
ITER($TIER2.COUNT, $index)
{
  $name1 = $name1 + "tomcat" + $index + ",";
};
[WORKER_LIST] = "worker.list=" + $name1;

# Generate properties of each application server
ITER($TIER2.COUNT, $index)
{
  $port = "worker.tomcat" + $index + ".port=8009\n";
  $host = "worker.tomcat" + $index + ".host=" +
          $TIER2.NAMELIST.EACH + "\n";
  $lb = "worker.tomcat" + $index + ".lbfactor=1\n";
  $worker_props = $worker_props + $port + $host + $lb;
};
[WORKER_PROPERTIES] = $worker_props;

# Generate properties of the load balancer
CHOP($name1);
[WORKERS] = $name1;
```

Figure 3.4: Script for generating worker.properties.

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 3.4 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 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 heartbeat 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.

3.3 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 [40], 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 configuration 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

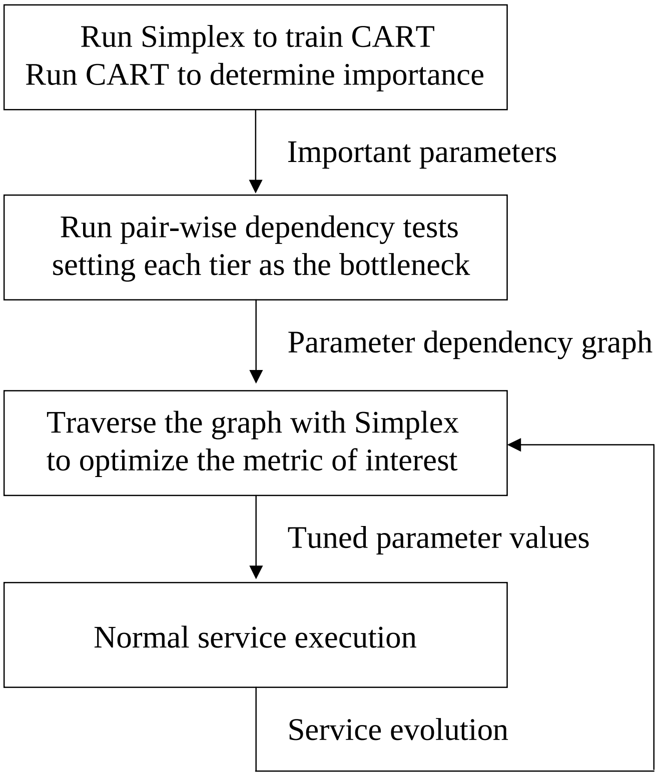
Run Simplex to train CART
Run CART to determine importance
Important parameters
Run pair-wise dependency tests
setting each tier as the bottleneck
Parameter dependency graph
Traverse the graph with Simplex
to optimize the metric of interest
Tuned parameter values
Normal service execution
Service evolution

Figure 3.5: Steps involved in tuning parameters.

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.

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 3.5 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.

3.3.1 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 interest. To accomplish this reduction, we apply the Classification And Regression Tree (CART) algorithm [10], 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 3.4), 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 [43], 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 3.6 and 3.7. Figure 3.6 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 3.7.

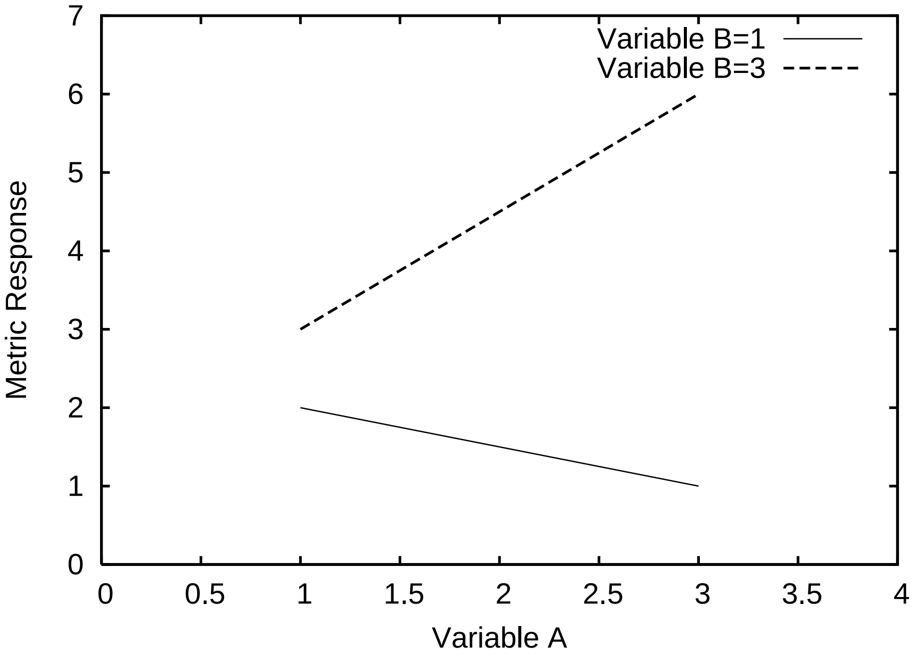

Figure 3.6: A depends on B.

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 3.6 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

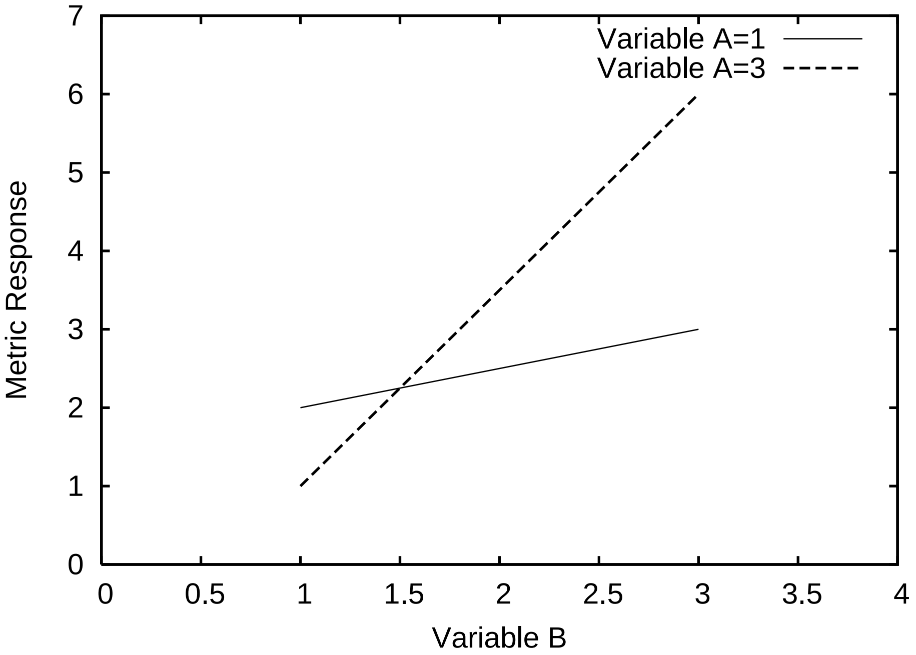

Figure 3.7: B does not depend on A.

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 thesis 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.

3.3.2 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.

3.3.3 Simplex Algorithm

The Simplex algorithm, as extended by Nelder and Mead [43], 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 3.8.

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

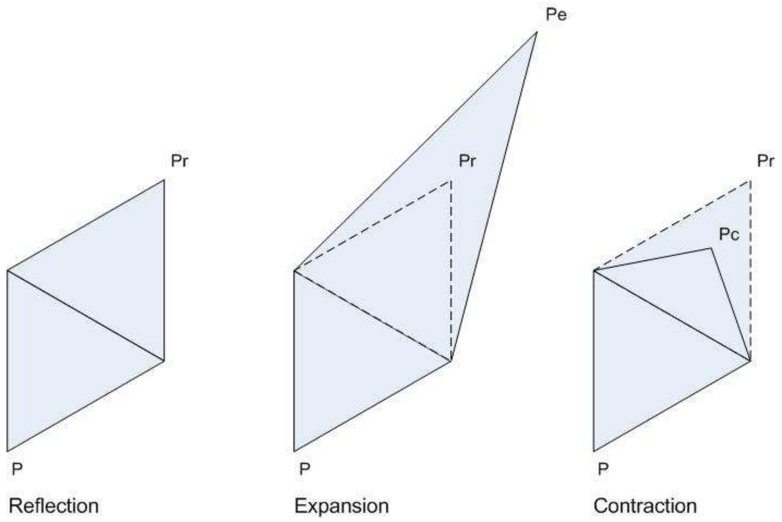

Figure 3.8: Main operations in the Simplex algorithm.

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.

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.

3.4 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.

3.4.1 Experimental Setup

For comparison purposes, we experiment with the same online auction service used in [41]. 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 [57].

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) 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.

3.4.2 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.* [11]. 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 [41]. Table 3.1 lists the complexity results with and without our infrastructure for the four tasks that involve configuration: node addition

	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	≥ 1	0	4	2
	MemSizeAvg	2.4	0	2	1.2	NA	0	1.33	0.67
	MemDepthMax	7	0	3	2	≥ 1	0	2	2
	MemDepthAvg	2.3	0	1.4	1.2	NA	0	1	0.67
	MemLatMax	3	0	3	0	≥ 1	0	1	1
	MemLatAvg	0.5	0	0.6	0	NA	0	0.44	0.33

Table 3.1: Management complexity with and without our infrastructure.

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

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

Table 3.2: Misconfigurations with our infrastructure.

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 [41]. In particular, we use the operator action traces that they collected for the four tasks above. Table 3.2 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

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 3.3: Performance-relevant parameters.

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).

3.4.3 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 3.3. 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.

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 3.4: Parameters with relative importance greater than 20 as computed by 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 3.4 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.

The second step in generating the graph involves finding dependencies between

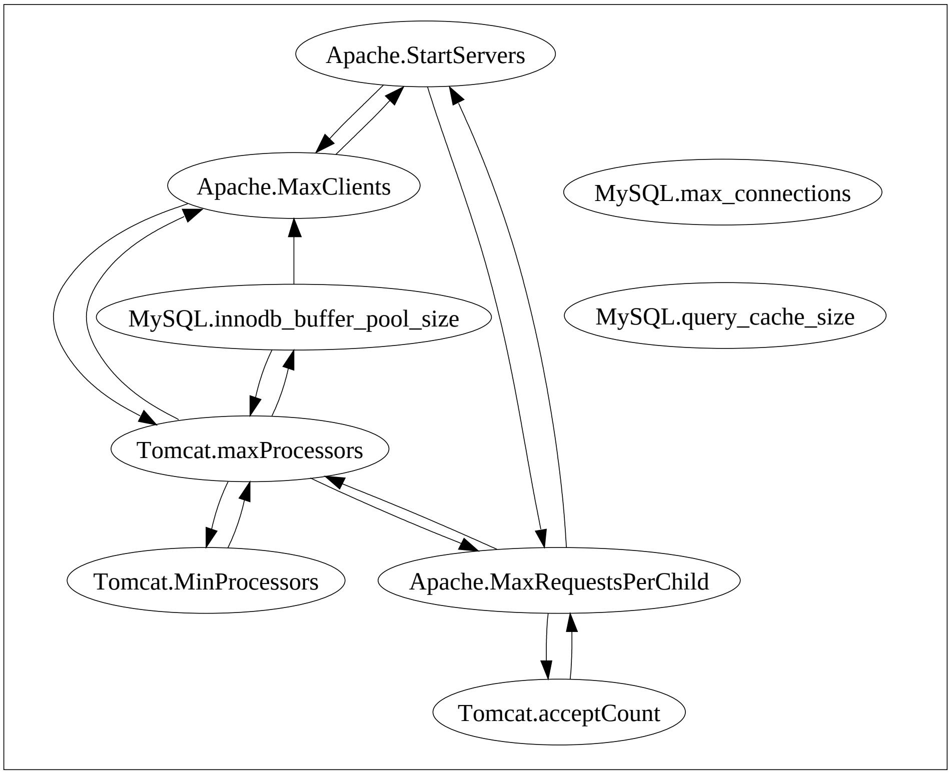

Figure 3.9: Parameter dependencies.

parameters. Following the procedure described in Section 3.3.1, we performed 567 experiments. To determine the dependencies, we set the dependency threshold to 10%, leading to the relationships portrayed in Figure 3.9.

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:

- Exhaustive search: Explores all combinations of parameter values. Overall,

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 3.5: Comparing performance tuning approaches for two service evolutions.

$O(R^N)$ 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 3.3.3 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 3.9), 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 3.5 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.

3.5 Summary

In this chapter, 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 could extend the work presented here in a few different ways. In particular, we could study 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 could extend our infrastructure to deal with application-level evolutions. Furthermore, we would extend our infrastructure to automatically detect significant changes in workload characteristics, which should also prompt parameter tuning. As a longer-term goal, we could 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.

Chapter 4

MassConf

4.1 Introduction

The previous chapter describes how ACI automatically tunes configuration parameters as a system evolves. When the system is first configured (i.e., before it starts evolving), ACI relies on Simplex to initialize configuration parameter values. However, Simplex often needs to run a large number of experiments. Thus, those users who deploy server software for the first time cannot take advantage of ACI, and still face great challenges to tune configuration parameters for a specific performance or energy target. While vendors often provide default configuration files that work well for many users, configurations need to be tuned in many other cases.

Unfortunately, as described in the previous chapter, configuring modern software can be extremely difficult in those cases. The reason is that a good configuration depends (at least) on the hardware environment, the workload, the load intensity, and the target behavior (e.g., some level of performance or availability) the user wants to achieve. Moreover, it is very hard (if not impossible) for users to completely understand the configuration-hardware-workload-intensity-target relationship.

Due to the size and complexity of this configuration space, previous research has focused on approaches and tools to detect misconfigurations and/or troubleshoot them [1, 37, 41, 65, 78, 79, 81], to study the resilience of systems in the face of configuration errors [35], to automatically configure a large number of machines in a single installation [2, 3, 12], and to automatically tune configurations for performance [73].

Although these efforts have been useful, a user's ability to configure her software to achieve a certain target behavior is still far from ideal in practice. For example, a user who wants her server software to produce an average response time of 50 milliseconds is left clueless, when the default configuration reaches only 100 milliseconds. As long as the parameters that affect performance are identified, this user can run existing algorithms (e.g., [43]) to optimize the server's performance by experimentation with different configuration settings. However, tuning performance may involve a very large number of time-consuming experiments [73]. For example, each tuning experiment with a database server may involve restoring a large database and its indexes back to a specific initial state.

We argue that vendors need to do more to help users configure their software. One approach vendors could take is to create automatic configurers that run locally at the users' sites and select the best values for the parameters, either through experimentation or modeling. A simpler and cheaper approach, the one we advocate here, is for the vendor to collect configuration information from the existing user community of its software and use it in configuring the software for new users.

Our approach is based on two key observations. First, *a configuration may actually work well for many users*, i.e. it may work well for many workloads, load intensities, and target behaviors, especially when the users use similar hardware platforms. For example, the default configuration often produces acceptable behavior for many users. This observation means that popular (i.e., frequently used) existing configurations may work well for many new users of the software. Figure 4.1 illustrates this observation with a simple example. Configuration C_9 is more popular than C_2 and C_4, as it is used by more users (2 instead of 1).

The second observation is that *multiple configurations may actually work well for each user*, i.e. they may all meet the user's target behavior. This observation means that there is flexibility in which existing configuration to select for each new user; even configurations that are unpopular may work well for her. For example, a user seeking

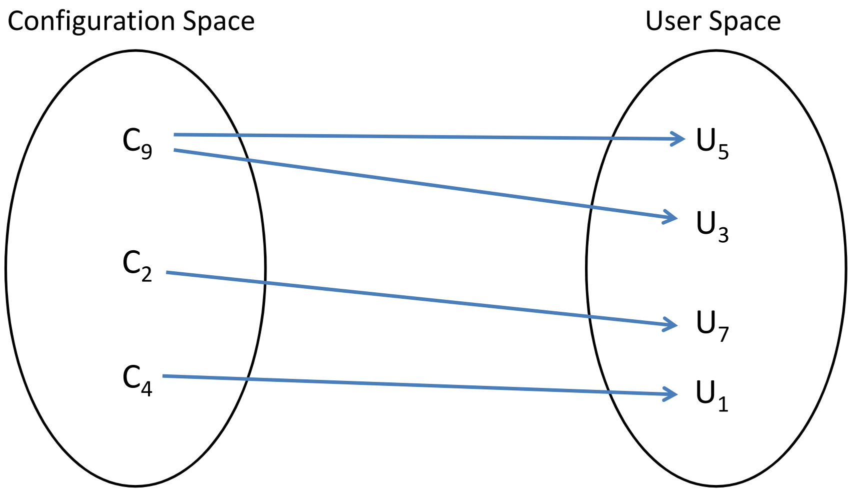

Figure 4.1: Many users may use the same configuration. An arrow from A to B means that configuration A is used by user B. Solid arrows represent information that MassConf has.

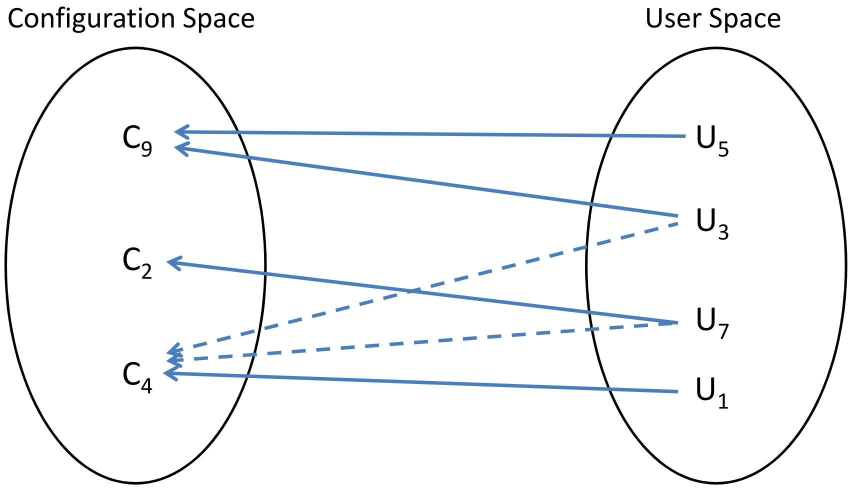

Figure 4.2: Many configurations may work for each user. An arrow from A to B means that user A can use configuration B. Dashed arrows represent data that MassConf cannot obtain.

an average response time of 50 milliseconds for a light workload may be able to use many different configurations, including the default one and a configuration that has enabled a single user with a particularly heavy workload to achieve high performance. Similarly, many other users may be able to use the latter configuration. Figure 4.2 illustrates this observation, showing that users U_3 and U_7 could also use configuration C_4 (besides C_9 and C_2, respectively).

These observations mean that any work that users may do to tune their configurations can benefit new users of the software as well. Thus, in this chapter we propose to leverage the existing users' configurations to find a good configuration for each new user. To demonstrate this idea, we designed MassConf, a system that automatically collects configuration and environment information from existing users, clusters users according to environment, produces an ordered (ranked) list of possible configurations, and tests each configuration in turn at the new user's site until the target behavior is met. After the configuration of each new user is complete, MassConf may change the ranking of configurations. MassConf seeks to (1) reach the target behavior for as many new users as possible and (2) minimize the average number of experiments required at the new users' sites. Because users are sometimes reluctant to divulge information about their systems, MassConf stores as little data as possible about them: only their environment descriptions and the non-sensitive parts of their configurations.

The most interesting technical aspect of MassConf is its ranking of configurations. Faced with our first observation above, one would be tempted to rank configurations based on popularity; more popular configurations would be tried first at the new users' sites. The popularity information is readily available from the existing users' deployed configurations. In our example, the popularity information corresponds to the relationships depicted in Figure 4.1. However, as our experiments shall demonstrate, the popularity-based ranking is not the best choice. The reason is that particularly effective but difficult-to-find configurations would tend to appear towards the end of the list. Ranking them higher would allow more new users to be configured with fewer experiments.

To account for this effect, MassConf would require information about how every deployed configuration would do for every existing user. In our example, it would need to know at least about the dashed arrows in Figure 4.2. Knowing this information, MassConf would rank C_4 higher than any other configuration, since C_4 can satisfy more users than any other configuration. Unfortunately, *such information is obviously not obtainable.* Thus, MassConf adapts its behavior over time, by moving the configuration that is selected for each new user toward the front of the ranked list, regardless of its actual popularity. This adaptation increases the chance that another new user will also experiment with (and hopefully benefit from) the selected configuration.

Our optimized version of MassConf, which we call MassConf+, improves ranking further by cutting off the ranked list of configurations after an initial "learning" period. Shortening the list rids MassConf+ of configurations that are unlikely to satisfy a large number of new users, thereby reducing the average number of experiments. (Hereafter, we only refer to MassConf+ explicitly when discussing material that does not apply to MassConf. When the context does not require such a distinction, we refer to both systems simply as MassConf.)

To evaluate MassConf's ranking of configurations in an interesting (yet understandable) case study, we investigate its use for automatically configuring the Apache Web server to achieve a response-time target. Despite the popularity of Apache, evaluating MassConf poses a challenge for academics, namely the unavailability of massive user configuration datasets in the public domain for any piece of software. Instead of being discouraged by this challenge, we decided to evaluate MassConf using a synthetic user population. In this context, we study different speeds for moving a selected configuration to the front of the ranking, as well as the popularity-based ranking. As a baseline for comparison, we use Simplex [43].

The results of our case study show that adaptive ranking requires many fewer experiments than the popularity-based ranking to configure a population of new users.

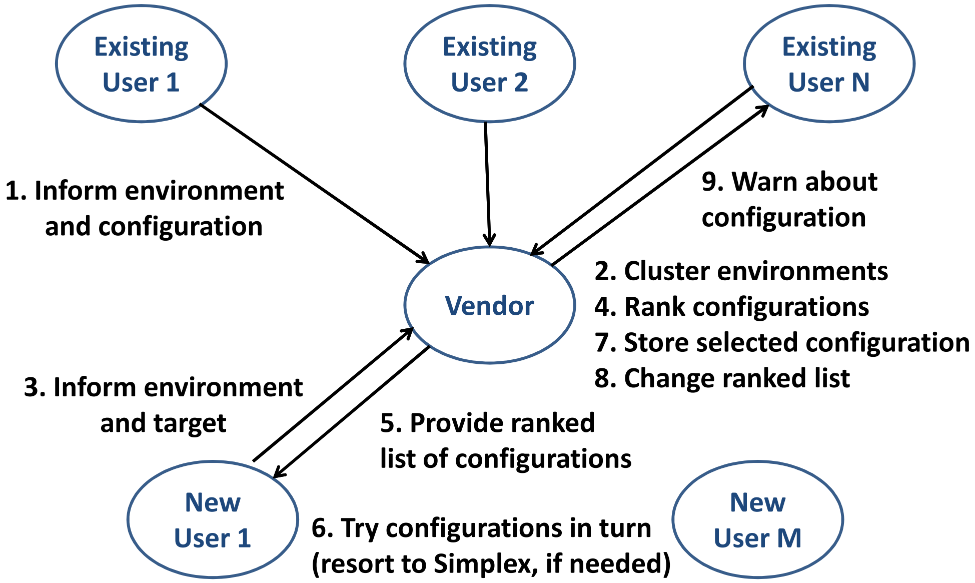
Existing User 1
Existing User 2
Existing User N
1. Inform environment and configuration
9. Warn about configuration
Vendor
2. Cluster environments
4. Rank configurations
7. Store selected configuration
8. Change ranked list
3. Inform environment and target
5. Provide ranked list of configurations
New User 1
6. Try configurations in turn (resort to Simplex, if needed)
New User M

Figure 4.3: MassConf overview.

Regarding adaptation speeds, we find that the faster we move a selected configuration to the front of the ranking, the better on average. In fact, the best approach is to always move such a configuration straight to the front of the ranked list. Popularity-based ranking is only faster on average than the slowest moving adaptation. We also find that MassConf can configure many more new users than Simplex. Moreover, MassConf requires many fewer experiments than Simplex, even when we consider only those new users that both systems can configure. MassConf+ reduces our average number of experiments per new user even further. Based on our experience with the case study, we qualitatively extrapolate from it to identify the general conditions under which MassConf is most effective.

Our experience and results illustrate that software configuration can be significantly simplified by having users contribute parts of their configurations and use them to configure the software for other users. Because of its simplicity and effectiveness, we conclude that MassConf and its adaptive ranking of configurations have great potential to work well in practice.

The remainder of the chapter is organized as follows. The next section describes MassConf in detail, including its configuration ranking algorithms and Simplex. Section 4.3 introduces our Apache case study and experimentally evaluates MassConf. Section 4.4 extrapolates from the Apache study. Finally, Section 4.5 summarizes the chapter and discusses some opportunities for future work.

4.2 MassConf

In this section, we first detail the design of MassConf. We then describe its ranking approaches and discuss its bootstrapping. After that, we describe Simplex. The last subsection discusses some potential refinements to MassConf.

4.2.1 Design

Figure 4.3 illustrates the MassConf design. The next few paragraphs detail each part of the design in turn.

Data collection from existing users. MassConf is run by the software vendor. First (step 1 in the figure), it collects configuration and environment information from each existing user that is willing to participate. (Although some users may refuse to provide this information, many would likely be willing to contribute to the community since they can benefit from it as shall be clear below.) This information is extracted by instrumentation in the server software itself and sent to the vendor.

The configuration information describes the settings of each configuration parameter of the software. The settings can be of any type, e.g. boolean, numeric, or character strings. When the configuration information may include sensitive data, only a few relevant parameter values may be collected. (The vendor should know which parameters may include sensitive data.)

As part of the configuration information, MassConf must be informed about the users' *high-level goals* when they selected their configurations. For example, the goal may have been to improve performance, improve performability (performance + availability), or lower energy consumption.

MassConf stores the parameter settings it receives without modification, except in the case of numeric parameters. For each numeric parameter, MassConf breaks the range of possible values into 10 evenly sized chunks. Two configurations are grouped together if their values for each parameter fall in the same chunk. For example, suppose that each configuration has two parameters, p_1 and p_2, with possible values ranging from 1 to 200 (chunks of size 20) and from 1 to 100 (chunks of size 10), respectively. Further, suppose that the values of these parameters for configurations C_4 and C_5 are: $C_4(p_1)$ = 10 (first chunk), $C_4(p_2)$ = 18 (second chunk), $C_5(p_1)$ = 16 (first chunk), and $C_5(p_2)$ = 12 (second chunk). Because the chunks match for each parameter, C_4 and C_5 would be grouped together.

The configurations in each group are represented by a single “average” configuration. In the average configuration, each parameter is given the average of the values seen for that parameter in the corresponding group. For example, the average configuration C_{avg} for the cluster formed by configurations C_4 and C_5 above would have $C_{avg}(p_1)$ = 13 and $C_{avg}(p_2)$ = 15.

The environment information is a description of the hardware (e.g., number of cores, amount of memory) and possibly the low-level software (e.g., operating system, settings for relevant environment variables) at the user’s site. This information is necessary since the behavior of the software to be configured may depend heavily on the environment.

Clustering existing users according to environment. Using the environment information, MassConf then clusters the existing users (step 2) as was done in Mirage [22] for software upgrade deployment. The idea is to cluster users that have similar environments together, so that their configuration information can be used for new users with similar environments. For example, the vendor of a multithreaded server may want to separate out user sites with vastly different numbers of cores or threading libraries, as these aspects of the environment may have a significant effect on the ideal number of threads with which to configure the server. Conversely, user sites with similar numbers of cores and thread libraries should be clustered together. A number of algorithms can be used for clustering, but we prefer the Quality Threshold (QT) algorithm [33]. QT starts with one site per cluster. It then iteratively adds sites to clusters (effectively merging clusters) while trying to achieve the smallest average inter-site distance and not to exceed a pre-defined maximum cluster diameter. The algorithm stops when no more clusters can be merged together. Our distance metric involves the aspects of the environment that differ between clusters. Each aspect is weighted by the vendor, according to its importance to the software configuration.

Collecting information from a new user. After clustering some existing users, MassConf is ready to configure a new user. It first deploys the software to the new user's site and collects its environment information (step 3). Then, it requests from the user a description of the software's target behavior (step 3). The target behavior reveals the high-level goal for the configuration tuning. With the new user's environment information, MassConf can now identify the best cluster for it.

Ranking configurations. Using the configuration information from this cluster, MassConf produces a ranked list (or ranking) of configurations to be tried at the new user's site (step 4). The list is formed by the configurations of the existing users that had the same goal as the new user. (For example, we do not want to use information about configurations that were selected to lower energy consumption when configuring servers for maximum throughput.) The exact ordering of the list is influenced by the order of the users' (both new and existing) arrivals, as described below. The list is transferred to the new user's site (step 5). At this point, MassConf can run experiments with each configuration, until the desired behavior is met or it runs out of configurations to try (step 6).

Testing configurations at the new user's site. These experiments are run under the user's actual workload and load intensity, so the user herself may have to provide a realistic test harness to exercise the software. If all experiments are run and the desired behavior is never achieved, the user is warned. MassConf's inability to reach a target may mean that the target is unrealistic for the workload and load intensity, or that it still does not have enough information (i.e., enough existing users) to produce a large enough coverage of the possible configurations. (We discuss this bootstrapping problem in a later subsection.) If the user confirms that the target is achievable and the parameter values are numeric, MassConf resorts to Simplex, *starting from the best configuration it has found so far*. However, we expect that MassConf would rarely have to resort to other approaches in practice; in most cases, the new user would relax the target. In these cases, MassConf would most likely have already found an appropriate configuration.

Storing the selected configuration. When a configuration is selected, MassConf includes information about it in its central database of existing users (step 7). At that point, the new user becomes one of the existing users within the corresponding cluster. (As MassConf found the configuration for the new user, it already has all the information required from an existing user.) Thus, after the bootstrapping period, the population of existing users should exhibit similar characteristics (as a group) to the new users (also taken as a group).

Adapting the ranking. As it is impossible to predict the set of new users that will want to join the system, MassConf adjusts its ranking (step 8) by moving the configurations that have been selected for each new user towards the top of the ranking. This adjustment enables very good configurations to be chosen more often. When MassConf needs to resort to Simplex, the new configuration is added to the end of the ranking. We discuss these decisions in detail in the next subsection.

Providing feedback to existing users. Finally, MassConf warns existing users when their configurations seem suboptimal (i.e., new users with the same goal have selected other configurations) with respect to the rest of the users in the same cluster (step 9). This feedback to existing users is an incentive for them to provide their configuration and environment information, even when they had to configure their software entirely by hand or when the community was still small. Another important incentive may be to help these users configure an upgraded version of the software by leveraging information from the users who benefited most from MassConf to configure the existing version.

Discussion. Previous systems have also relied on information from their users [1, 66, 78, 79]. However, those systems seek to troubleshoot configurations, not tune them as MassConf does. In our context, the specific and diverse characteristics of the users' workloads and their behavior targets mean that configuration information is also diverse (i.e., coercion as in PeerPressure [78] does not apply) and prior actions from a user do not produce the same results for another user (i.e., local experiments are necessary). For these reasons, our main focus has been studying adaptive ranking algorithms and the number of tuning experiments to which they lead on average. Neither of these issues was considered by these prior works.

Although we have focused on the use of MassConf to configure software at the users' sites, our system can also be used for software that users deploy to Cloud Computing services such as Amazon's EC2. In this case, MassConf would require information about the *virtual* environment (and, possibly, the service) on which the software will be run. Every other aspect of MassConf would remain as described above.

4.2.2 Configuration Ranking

Dynamically adapting the ranking. As we mentioned before, a popularity-based ranking can be misleading. It is possible that unpopular configurations can actually satisfy many more new users than popular ones. The reason these highly useful configurations are not more popular may be that they are harder to find, e.g. they are only

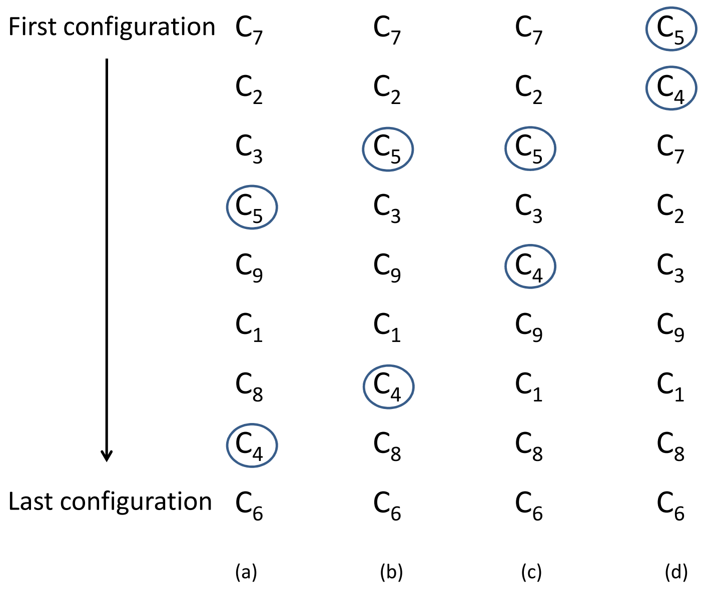
First configuration
Last configuration
C7 C2 C3 C5 C9 C1 C8 C4 C6
C7 C2 C5 C3 C9 C1 C4 C8 C6
C7 C2 C5 C3 C4 C9 C1 C8 C6
C5 C4 C7 C2 C3 C9 C1 C8 C6
(a)
(b)
(c)
(d)

Figure 4.4: Original ranking (a) and slow (b), fast (c), and fastest (d) adaptation approaches, after configurations C_4 and C_5 are selected by two consecutive new users.

needed for heavy workloads or hard-to-achieve target behaviors.

Instead of relying on popularity, MassConf dynamically adapts its rankings to eventually concentrate configurations that can satisfy many new users at the top. We study three approaches for promoting the selected configurations within a ranking: *slow*, *fast*, and *fastest*. The slow approach moves a selected configuration one slot up in the ranking. The fast approach moves the configuration to the halfway point between its current slot and the top of the ranking. The fastest approach moves the configuration directly to the first slot of the ranking. Figure 4.4 shows an example of how ranking (a) is adjusted after configuration C_4 and C_5 are selected by two consecutive new users, using the slow (b), fast (c), and fastest (d) adaptation approaches. For example, in the fast approach, C_4 is first moved from the 8th to the 4th slot in the ranking. This moves $C_5, C_9, C_1,$ and C_8 one slot down the ranking. Then, when C_5 is selected by the next new user, it moves from the 5th to the 3rd slot. This moves C_3 and C_4 one slot down.

Regardless of the speed of promotion, any new configurations that are added to the system are appended to the end of the corresponding ranking. The reason is that we want to see more than one user benefit from a new configuration before we promote it up the ranking.

Note that configurations coming from existing users are treated the same as those selected for the new users, despite the fact that the former users select their configurations by means other than MassConf (i.e., the ranked configurations are not tested in turn for these users). We also considered the possibility of not altering the ranking when an existing user joins with a configuration that had already been seen. We ultimately decided against this approach because it would disregard the fact that the configuration satisfied an additional user.

Cutting off the tail of the ranking (MassConf+). After a period of adaptive ranking in MassConf, the configurations that satisfy the most users' targets will tend to rank high and reduce the average number of experiments per new user. Conversely, configurations that are not as widely useful will tend to be left at the tail of the ranking. This means that the likelihood that a configuration will satisfy a new user decreases rapidly as we move past the first set of configurations. Beyond this set, it may actually be more advantageous for MassConf to cut off the ranking and resort to Simplex right away, instead of trying a large number (potentially all) of the less useful configurations.

Based on this observation, we designed an optimized version of MassConf (called MassConf+) defining two thresholds: (1) the number of new users to see before cutting the ranking off; and (2) where the ranking should be cut off. MassConf+ uses heuristics to select these thresholds. For (1), it waits until the average number of experiments for configuring each new user has gone down many times in a row (10 times by default). Another option would have been to wait for a period with a stable average number of experiments per new user. We selected our current approach, because it allows MassConf+ to cut the list faster (before the average has stabilized). For (2), it cuts off the ranking at the number of configurations that has satisfied a large percentage (80% by default) of the new users seen so far.

Picking these thresholds properly is important, since any new configurations that are added to the system are *not* added to the corresponding shortened ranking. The reason is that adding these configurations to the shortened ranking could discard more useful configurations. A more robust approach could be to repeatedly prune the ranking, allow it to grow (which would happen ever more slowly) for a period, and then prune it again. Our simpler approach has worked well in our experiments, so we leave the more sophisticated one for future work.

4.2.3 Bootstrapping the System

Any system that relies on other systems' information to make decisions faces a bootstrapping problem. MassConf is no different. It starts performing well when the existing users within each cluster become a good representation of the new users to come into the same cluster. Until that point, MassConf may be unable to meet the target behavior requested by new users without resorting to (experiment-intensive) Simplex. Instead of resorting to Simplex, the new user may also decide to optimize the configuration manually until the target behavior is achieved. Fortunately, these Simplex-derived or manually generated configurations contribute to MassConf just the same as the configurations of the existing users that join MassConf.

4.2.4 Simplex

The details of the Simplex algorithm appear in section 3.3.3. In the context of this chapter, each vertex of the simplex is a configuration. The operations done to each vertex involve operating with the corresponding parameters of the configuration. For example, a reflection involves reflecting each configuration parameter of the worst configuration independently with respect to its value in the centroid configuration.

In our experiments, we set Simplex to terminate when a target average response time is reached or the standard deviation of the vertices' response times is smaller than 5 milliseconds [43]. Under these stopping criteria, Simplex required between 7 and

174 experiments.

4.2.5 Potential Refinements

We have considered many refinements to MassConf. We describe some of them next.

Storing workload and load intensity information. We currently only collect configuration and environment information from users. However, we could also potentially collect workload and load intensity data and use it to improve our ranking of configurations. The obvious difficulty is how to characterize these new data in a manner that enables a meaningful comparison of different user sites. For example, we could collect resource utilization data summarizing behaviors at each user's site. However, the resource utilizations of two sites may be similar while the actual workload and load intensity are quite different (e.g., high CPU utilization may be the result of a light load that is computationally intensive or a high load that is network stack intensive). Configuration and environment information (including the users' tuning goals) are sufficiently well-defined that characterization is not a problem.

Storing the results of experiments. Right now, MassConf only stores the configuration that is selected for each user; the exact behavior (e.g., response time or energy consumption) to which this configuration leads is not used or stored. Another potential refinement would be to store all the configurations and actual experiment results on the way to meeting targets. The experiment results from the existing users could be compared to the new users' targets to potentially improve ranking further. Specifically, the configurations leading to results that are close to the targets could be ranked higher than others. However, this would require more complex ranking algorithms that consider the experiment results. Furthermore, it is unclear that experiment results obtained for the specific workload, load intensity, and target of one user would be useful in configuring the software for another user.

More aggressive re-ranking. Right now, MassConf does not change its ranking as a result of each experiment at a new user's site. Changes are only made after a good configuration has actually been selected for the new user. Another approach could be to change the ranking as we observe the behaviors of different configurations at the new site. For example, this more aggressive adaptation could be started after finding out that the configurations tested early on lead to very poor behavior compared to the target.

As Section 4.3.4 demonstrates, MassConf is very effective, so we decided that these refinements were not worth their additional complexity.

4.3 Case Study: Apache Configuration

To understand and validate MassConf and its adaptive ranking, we consider the Apache Web server as a case study. In particular, we focus on configuring Apache to achieve a target average response time. Because we lack real configuration data, we create a synthetic population of users. In this section, we first describe our approach for generating the populations of existing and new users. Then, we analyze the bootstrapping behavior and the characteristics of the configurations deployed by our users. Finally, we evaluate MassConf by comparing its adaptive ranking against the popularity-based ranking, as well as comparing its results to those of Simplex.

4.3.1 Methodology

Apache configuration and performance. As listed in Table 3.3, Apache has five main configuration parameters that affect performance: StartServers, MinSpareServers, MaxSpareServers, MaxClients, and MaxRequestsPerChild [70]. StartServers specifies the number of server processes that should be started, MinSpareServers specifies the minimum number of server processes that should be kept in a spare pool, MaxSpareServers specifies the maximum number of server processes that should be kept in the spare pool, MaxClients defines the maximum number of server processes allowed to start, and MaxRequestsPerChild defines the maximum number of requests a server process may serve (0 means infinite). The default configuration assigns values of 5, 5, 10, 150, 0, respectively, to each of these parameters.

Workloads, intensities, and targets. Each user in our synthetic population represents a different combination of workload, load intensity, and response-time target.

We define each workload by its fraction of requests for three types of content: small static files (average size 13KB with 20% cache miss rate), a large static file (130KB in size with 0% cache miss rate), and dynamic CGI scripts (each consuming 14ms of CPU execution). Each of these types of requests stresses a different part of the system: the file system, networking, and the CPU, respectively. We refer to the fraction of CGI requests F_{CGI} as a fraction of the total number of requests. In contrast, we refer to the large-file component of the workload F_{LF} as a fraction of the static requests. The remaining percentage represents the requests to small files. We vary F_{CGI} and F_{LF} from 0% to 100%.

To define load intensities for each workload that do not overload the system, we experimentally find the intensity that leads to saturation assuming the default configuration. We call this the “maximum throughput” for the workload. Then, we assign load intensities for each workload from 50 requests/second to the maximum throughput with a step of 50 requests/second. Since the maximum throughput T_{max} is not always a multiple of 50, the maximum load intensity L_{max} is

$$L_{max} = \begin{cases} \lfloor \frac{T_{max}}{50} \rfloor * 50 & T_{max}\%50 < 25, \\ T_{max} & T_{max}\%50 \geq 25. \end{cases}$$

For each workload, we select different targets along the reachable range. The targets are evenly distributed between the performance of the default configuration $P_{default}$ and the best performance we can achieve with Simplex $P_{simplex}$. Specifically, the targets are $[P_{default}, P_{default} * 0.95, \cdots, P_{simplex}]$. If $P_{simplex}$ is not exactly a multiple of 5% away from $P_{default}$, the last target we use is the lowest such value that is still higher than $P_{simplex}$. We choose 5% because it creates a good number of targets and poses a non-trivial challenge for configuration tuning.

User populations. We synthetically generated an initial set of "existing users" that is evenly spread in the 3D space of workloads, load intensities, and response-time targets. We assigned workloads of the form $F_{CGI} \in (0\%, 20\%, \cdots, 100\%)$ and $F_{LF} \in (0\%, 20\%, \cdots, 100\%)$ to these users. We selected intensities and targets as described above.

We define the existing users' configurations by running Simplex. In particular, we set Simplex to start from the default Apache configuration and stop trying new configurations (new values for the different configuration parameters) when the response-time target is met. Only this last configuration is stored for each existing user.

The set of "new users" in our synthetic population is also evenly spread across the parameter space, but is completely distinct from the set of existing users. In particular, the new users' workloads are defined as $F_{CGI} \in (10\%, 30\%, \cdots, 90\%)$ and $F_{LF} \in (10\%, 30\%, \cdots, 90\%)$. The load intensities for these users are selected as described above. When setting targets for the new users, we select targets that are achievable by either MassConf or Simplex alone. Our goals are to select configurations for as many of these new users as possible, while using the smallest possible number of experiments on average.

Overall, we create 219 existing users. We start with 31 different workloads. After selecting acceptable load intensities for each of these workloads, we produce 91 combinations. By defining reachable targets for each workload, we get to 219 combinations. We also create 195 new users, starting with 25 different workloads. When load intensities are considered, we reach 66 combinations. Finally, the addition of the reachable targets brings us to 195 combinations.

As one would expect, our population of users is quite diverse. For example, we have a user with F_{SF} = 100% and a load intensity of 400 requests/second that observes a response time of 125 ms, assuming the default Apache configuration. A second user requests F_{LF} = 100% at 87 requests/second for a response time of 278 ms, while a third user requests F_{CGI} = 100% at 80 requests/second for a response time of 174 ms.

Note that our evenly spread and non-overlapping populations of users represent a pessimistic scenario for MassConf. The reason is that any concentration of users in specific parts of the workload-intensity-target space would increase the likelihood that (1) many users would deploy the same configuration; and (2) many users could be satisfied by each configuration. These are the two basic premises behind MassConf, as mentioned in Section 4.1 and illustrated in Figures 4.1 and 4.2.

Finally, note that, for simplicity, we assume that all users have selected their configurations with the goal of lowering response time and belong to the same environment cluster. (In fact, in our experiments, all users use the same hardware and low-level software environment. Although it would have been interesting to investigate the effect of slight environment variations, this assumption does *not* skew our results. The reason is that, in the real world, there are many more users per environment than environments, just like in our experiments.) Because MassConf operates on clusters independently, our results extrapolate trivially to scenarios with multiple clusters.

Running experiments. Our experiments are run on two Dell 2650 machines. Each machine has one Intel Xeon CPU (2.80 GHz), 2 GB of memory, and a 7200 rpm disk. One machine hosts an HTTP client emulator and the other the Apache Web server (version v2.0.4). The machines run Linux 2.6.18 and are linked by a 100-Mbit Ethernet switch.

Using the client emulator, multiple clients concurrently send requests to the server. During each experiment, a pre-defined workload is sent to the server at a fixed rate, i.e. the pre-defined load intensity for that experiment. The inter-request time follows a Poisson distribution. At the end of each experiment, the emulator reports the average response time and throughput.

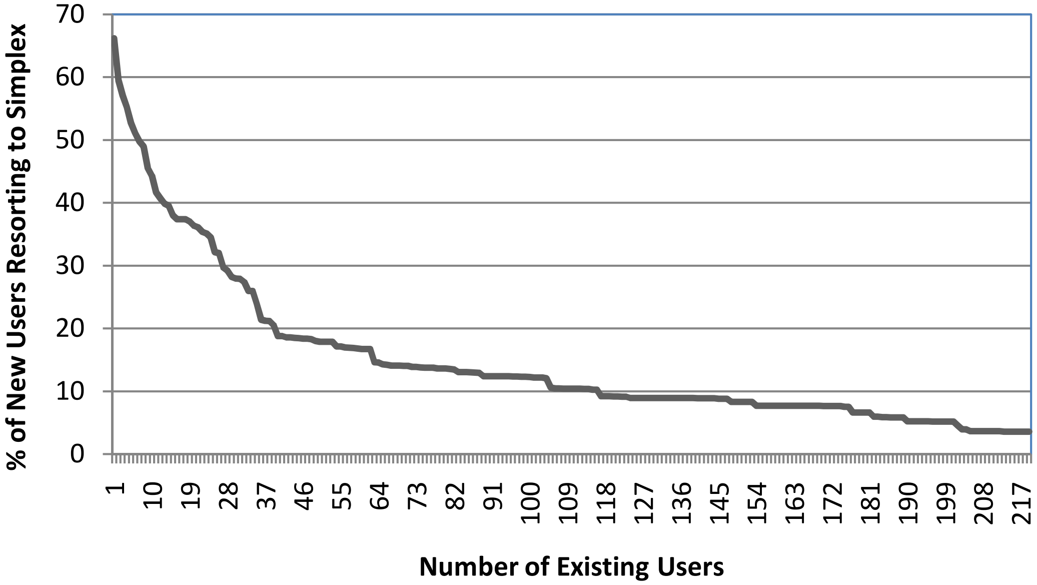

Figure 4.5: Bootstrapping in MassConf.

4.3.2 Understanding Bootstrapping

MassConf becomes most useful when the population of existing users has "stabilized" as a good representation of the new users to come, i.e. the probability that MassConf will have to resort to Simplex has become relatively small. This point occurs only when MassConf has gathered enough configurations.

Figure 4.5 illustrates the bootstrapping process. The figure assumes that MassConf is about to configure a new user, after a certain number of existing users have joined the system. The figure plots the probability that MassConf will have to resort to Simplex for the new user, as a function of the number of existing users who have already joined. We compute each probability by assessing the fraction of our new users that would require Simplex to run given the set of existing users. Since the state of MassConf at each point depends on exactly which existing users have joined, we plot the average fraction from 10 different (random) arrival orders.

As one would expect, the probability of needing Simplex is high when the number of existing users who have joined the system is small. As this number increases, the

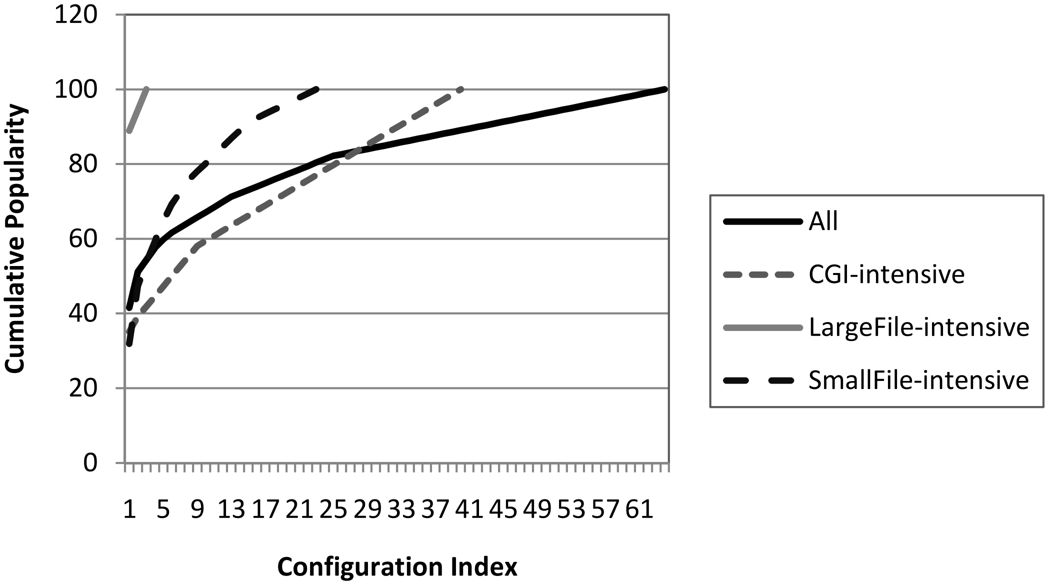

Figure 4.6: Popularity of CGI-intensive, large-file-intensive, small-file-intensive workloads, and all workloads.

probability falls sharply. Beyond roughly 37 existing users, the probability of needing Simplex falls below 20%. At that point, we can say that MassConf had been bootstrapped.

4.3.3 Understanding Ranking

Popularity. Popularity refers to how often an exact set of parameter values appears in a collection of configurations. For example, if the set of values $\{200, 17, 3, 1000\}$ for four relevant parameters appears in 12 out of 20 configurations, we say that this configuration has 60% popularity. Any configuration that appears a smaller percentage of times is considered less popular than this one. Popularity-based ranking ranks configurations based solely on their popularity.

Figure 4.6 illustrates the popularity of the configurations that met the performance targets for our existing users. The figure plots the popularity for workloads dominated by small files, large files, and CGI requests, as well as the popularity when all workloads are considered together. We define a workload to be CGI-intensive when

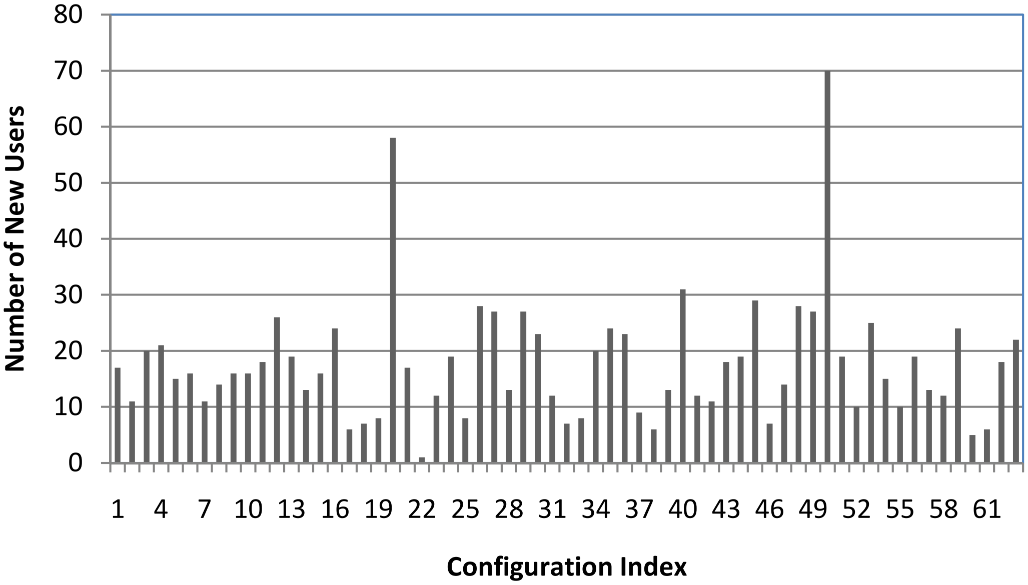

Figure 4.7: Popularity ranking and number of new users that can be satisfied by each configuration.

$F_{CGI} \geq 50\%$. A workload is large-file-intensive when $F_{LF}(1-F_{CGI}) \geq 50\%$, whereas it is small-file-intensive when $(1-F_{LF})(1-F_{CGI}) \geq 50\%$. On the X-axis, the figure shows the index of the unique configurations in decreasing order of popularity (from left to right). On the Y-axis, the figure shows the cumulative popularity of the configurations on the X-axis. The leftmost point of each curve is the default configuration.

The figure confirms one of the basic premises of MassConf, namely that certain configurations work well for many existing users, despite our pessimistic assumptions about the population of existing users. Specifically, we can see that the default configuration indeed works well for a large fraction of users. In addition, the fact that the curves are not straight lines shows that other configurations, besides the default one, are used by multiple users. Moreover, the figure shows that the configuration popularity of the three types of loads is quite different. Large-file-intensive workloads show the least amount of popularity, whereas small-file-intensive workloads show the most. These observations suggest that it is harder to configure the large-file-intensive workloads than others. Nevertheless, MassConf has potential to greatly benefit users with these types of workloads, as shown by the curve that accounts for all workloads together.

The other side of the coin. To fully understand ranking, we have to consider its impact given a population of new users. As we have suggested, ranking configurations based on popularity only may hide the fact that very good configurations just happen to be unpopular. Figure 4.7 illustrates this effect clearly. On the X-axis, the figure lists the index of each existing configuration in popularity-based order (the most popular on the left, the least popular on the right). Only the default configuration (index #0) is not listed. On the Y-axis, the figure lists the number of new users in our population that could be satisfied by each configuration.

The default configuration can satisfy the performance targets of 66 new users. As the figure shows, there is another configuration (#50) that can satisfy even more new users (70). Unfortunately, this configuration appears very late, i.e. it is very unpopular. This means that 49 other configurations would be tried before reaching this very good one. A similar observation can be made of configuration #20, which can satisfy 58 new users. Moving either (or both) of these configurations up the ranking would allow many new users to be configured with a smaller average number of experiments. In contrast, the two most popular configurations can only satisfy 17 and 11 new users. These observations clearly suggest that the popularity-based ranking can cause a higher than necessary number of experiments.

4.3.4 Experimental Evaluation

We now turn to evaluating the use of MassConf for configuring new user installations. We study two scenarios. In the first, we initialize MassConf with configurations from our 219 existing users and use it to configure our 195 new users to meet their response-time targets. In the second scenario, we initialize MassConf with configurations from 1/3 of our existing user population chosen at random. After this initialization phase, we use MassConf to configure our new users and integrate another 1/3 of our existing users (again chosen randomly) at the same time. The order of these user arrivals is also random. To mimic the fact that some users may decide not to join, the final 1/3 of existing users are not used in this scenario.

To evaluate MassConf in these scenarios, we compare its ranking adaptation algorithms to popularity-based ranking. In popularity-based ranking, the ranking changes whenever the selection of an existing configuration causes the popularity ordering to change; more popular configurations appear first. New configurations (found by Simplex) are added to the end of the ranking, as they are the least popular.

To put the results in context, we also compare them against Simplex (running on its own and starting from the default configuration). In addition, we present results for the "optimal" static ranking, i.e. the static ranking that generates the smallest possible number of experiments in configuring our population of new users. This ranking sorts the configurations in decreasing order of number of (unique) new users that they satisfy; selecting a configuration for a new user does not alter this order. Obviously, the optimal ranking can only be determined because we know the entire set of new users in advance, which is impossible in practice. We present results for the optimal ranking simply as a lower bound on the number of experiments.

Next, we discuss each of the arrival scenarios in turn.

First Scenario.

We make several observations from our experiments with the first scenario:

1. MassConf successfully reached the performance targets of all new users. Out of our 195 new users, 66 were able to meet their response-time targets using the default configuration. MassConf was able to configure *all* 129 new users that could not use the default configuration. When MassConf is not allowed to resort to Simplex (MassConf-without-Simplex), it is able to configure 122 of these new users. Two out of the 7 new users that MassConf-without-Simplex cannot configure have light workloads and

Number of Experiments	Popularity Ranking	MassConf Adapt-slow	MassConf Adapt-fast	MassConf Adapt-fastest	Optimal Static
Total	1519	2383	1380	1272	873
Avg.	11.8	18.5	10.7	9.9	6.8
Max.	84	84	84	84	84

Table 4.1: MassConf vs. popularity for 129 new users.

Number of Experiments	Popularity Ranking wo Simplex	MassConf wo Simplex Adapt-slow	MassConf wo Simplex Adapt-fast	MassConf wo Simplex Adapt-fastest	Optimal Static wo Simplex
Total	1023	1887	884	776	377
Avg.	8.4	15.5	7.2	6.4	3.1
Max.	64	60	59	59	12

Table 4.2: MassConf-without-Simplex vs. popularity for 122 new users for which both approaches reached the targets.

seek to achieve 5% better performance than the default configuration can produce. The other 5 new users had higher targets and missed them by several percentage points.

2 and 3. Adaptive ranking beats popularity-based ranking. The faster the adaptive algorithm promotes configurations, the better. Table 4.1 summarizes the statistics for our new users. Table 4.2 summarizes the statistics for the case in which MassConf (with popularity-based or adaptive ranking) is not allowed to resort to Simplex. For this latter table, we only show results for the 122 new users that MassConf-without-Simplex can configure. Since the behavior of the adaptation algorithms depends on the exact sequence in which new users join the system, for both tables we generated 10 random sequences and averaged the results.

Both tables show that two adaptive ranking approaches (Adapt-fast and Adapt-fastest) require fewer experiments on average than the popularity-based ranking. The analysis of adaptive ranking from the previous section suggested this result. The faster selected configurations are promoted up the ranking, the smaller the average number of experiments per user. *The best adaptive ranking (Adapt-fastest) runs up to 24%*

fewer experiments per user than popularity-based ranking on average. In contrast, Adapt-slow actually requires up to 85% more experiments per user than popularity-based ranking on average. There are two effects at play here: (1) on the positive side, moving a good configuration up enables it to satisfy more users; and (2) on the negative side, it may increase the number of experiments required when a configuration that was moved down is selected. When moving up one slot at a time, only a few extra users can be satisfied by the promoted configuration, so the negative effect becomes more prominent. When moving configurations up faster, the good configurations can satisfy many extra users, making the positive effect more prominent.

The fact that Adapt-fastest is the best approach confirms the two observations that motivated our MassConf design: one configuration works well for multiple users and multiple configurations work well for each user. If only one configuration met each user's target, Adapt-fastest would make the worst decision. At the other extreme, if all configurations met all new users' targets, all approaches would produce the same number of experiments, i.e. 1.

When we compare MassConf Adapt-fastest to the optimal (but unrealistic) static ranking, we find that our system is 31% slower (Table 4.1). Nevertheless, as shall be seen, MassConf+ actually performs better than this optimal ranking, because it shortens the list of configurations to be tried. (We shall compare MassConf+ to a different optimal static ranking below.)

4. MassConf successfully reached the performance targets for many more users than Simplex. As mentioned above, MassConf was able to configure all 129 new users that could not use the default configuration. In contrast, *Simplex failed to configure 74 of these new users.* Even when MassConf is not allowed to resort to Simplex, it still can configure 67 more new users than Simplex (122 vs. 55). The reason Simplex cannot configure these new users is that it gets stuck at local minima, trying configurations that lead to very similar performance.

The ability of MassConf and MassConf-without-Simplex to configure many more

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-slow	MassConf Adapt-fast	MassConf Adapt-fastest	Optimal Static
Total	1023	818	978	762	730	639
Avg.	18.6	14.9	17.8	13.9	13.3	11.6
Max.	112	84	84	84	84	84

Table 4.3: MassConf vs. Simplex for the 55 new users for which Simplex reached the targets.

new users than Simplex is particularly interesting since our existing user configurations were originally derived using Simplex. This result reinforces the point that Simplex has to search a large space of configurations each time it is used and so, for any particular search, it may miss some "good" configurations. MassConf is completely different in that it is guided by the tuning efforts of existing users and its adaptive ranking algorithms.

5. MassConf is faster than Simplex. To properly compare the number of experiments required by MassConf and Simplex, we consider only the subset of 55 new users for which *both* MassConf and Simplex were able to achieve the performance targets. Table 4.3 summarizes the statistics for these new users. Again, these results are the average over 10 random sequences. The table shows that the three adaptive ranking approaches require between 13.3 and 17.8 experiments per user on average. The best approach, Adapt-fastest, requires 13.3 experiments on average, which is *28% faster than Simplex.*

6. MassConf+ improves significantly on MassConf. MassConf runs a significant number of experiments when the chosen configurations are either at the tail of the ranking or not in the ranking at all. As an example of the latter scenario, MassConf unsuccessfully tries all 64 configurations before resorting to Simplex for the 7 new users' targets that were not met by MassConf-without-Simplex. MassConf+ was designed exactly to reduce the number of unsuccessful experiments in MassConf.

Number of Experiments	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1529	1262	793	528
Avg.	11.9	9.8	6.1	4.1
Max.	84	84	37	34

Table 4.4: MassConf+ vs. MassConf for 129 new users. The popularity and MassConf results here are different than in Table 4.1 because this table only considers one sequence of arrivals.

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1023	818	734	336	294
Avg.	18.6	14.9	13.3	6.1	5.3
Max.	112	84	84	34	34

Table 4.5: MassConf+ vs. Simplex for the 55 new users for which Simplex reached the targets. The popularity and MassConf results here are different than in Table 4.3 because this table only considers one sequence of arrivals.

To evaluate MassConf+, we investigated a number of sequences of new user arrivals. The results we discuss next represent a randomly selected such sequence. For that sequence, MassConf+ decided to cut off the tail of the ranking after having seen 28 new users. (Recall that MassConf+ selects this point after having seen 10 consecutive decreases in average number of experiments per new user.) After the 28th new user was configured, MassConf+ also decided to cut the ranking off at the 14th configuration. (Recall that the cut off point is the minimum size that was required to satisfy 80% of the 28 new users.) Starting with the 29th new user, MassConf+ only tries a maximum of 14 configurations for each new user before resorting to Simplex.

Using these thresholds, MassConf+ is able to find configurations that meet the targets of *all* the 129 new users that cannot use the default configuration. Table 4.4 summarizes the results of MassConf+, while comparing them against popularity-based ranking and MassConf. The Optimal+ system ranks configurations in the best possible order and cuts the ranking off at the optimal point (12 configurations). Again, Optimal+

is unrealistic and is presented simply as a lower bound on the number of experiments.

The table shows that MassConf+ reduced the overall number of experiments by 469, compared to MassConf. As a result of this reduction, *MassConf+ finds configurations 37% and 48% faster than MassConf and popularity-based ranking, respectively, on average.* In addition, it cuts the maximum number of experiments for any new user to less than half of those performed by MassConf and popularity-based ranking. Comparing these MassConf+ results with the optimal ranking results of Table 4.1, we can see that our system actually performs better. The reason is that MassConf+ prevents a large number of experiments by cutting off the ranking. Compared to Optimal+ ranking, MassConf+ incurs 33% more experiments.

Table 4.5 compares MassConf+, MassConf, and Simplex for the 55 new users for which Simplex reached the targets. The table shows that *MassConf+ finds configurations 67% faster than Simplex*, again while significantly reducing the maximum number of experiments for any new user.

To understand the impact of its two thresholds, we performed a number of sensitivity experiments with MassConf+ Adapt-fastest, assuming the entire set of 129 new users and the same sequence of new user arrivals. For the first threshold, we considered 5 and 15 continuous decreases of the average number of experiments per new user, besides the default setting of 10 continuous decreases. For the second threshold, we considered cutting off the list at the size that would satisfy 70% and 90% of the new users, besides the default setting of 80%.

When the first threshold is set to 5, the numbers of new users resorting to Simplex are 88, 88, and 71, when the second threshold is set to 70%, 80%, and 90%, respectively. However, regardless of the setting of the second threshold, MassConf+ does *not* reach the targets of all new users. The reason is that the ranking had not been trained enough before it was cut off. In contrast, when the first threshold is 15, MassConf+ always reaches the new users' targets. For this first threshold, the total numbers of experiments for the different settings of the second threshold are 604, 607, and 846,

Number of Experiments	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1447	1250	717	528
Avg.	11.2	9.7	5.6	4.1
Max.	70	66	37	34

Table 4.6: MassConf+ vs. MassConf for 129 new users in the second scenario.

respectively. In this case, MassConf+ achieves low experiment totals for 70% and 80% settings of the second threshold. Overall, these results suggest that it is most efficient to select a relatively low value for the second threshold (e.g., 70%), as long as the ranking is trained for long enough by picking a relatively high value for the first threshold (e.g., 10 or higher). In fact, note that picking a value of 15 for the first threshold would lead to significantly better MassConf+ results than those in Tables 4.4 and 4.5.

Second Scenario.

One could argue that the results of our first scenario above were optimistic in the sense that all existing users joined MassConf before any new user had to be configured. To evaluate MassConf in more pessimistic circumstances, we now turn to our second scenario. Recall that, in this scenario, we initialize MassConf with configurations from only a random 1/3 of our existing user population. After this initialization, the new users start arriving concurrently with another random 1/3 of the existing users. The final 1/3 of the existing users never joins.

From this scenario, we can make two observations:

7 and 8. The benefits of MassConf and MassConf+ remain significant. The comparisons between systems exhibit the same trends as before. Table 4.6 compares MassConf and MassConf+ with popularity-based ranking and Optimal+ ranking for our population of 129 new users. These results confirm the trends we observed from the first scenario. Specifically, (1) both MassConf and MassConf+ can configure all new users, even in the absence of a large fraction of existing users; (2) MassConf+ reduces the number of experiments by 42%, compared to MassConf; (3) MassConf and

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1023	747	702	365	294
Avg.	18.6	13.6	12.8	6.6	5.3
Max.	112	70	66	37	34

Table 4.7: MassConf+ vs. Simplex for the 55 new users for which Simplex reached the targets in the second scenario.

MassConf+ involve 14% and 50% fewer experiments than popularity-based ranking, respectively; and (4) MassConf+ runs only 27% more experiments than the unrealistic Optimal+ ranking algorithm.

Moreover, we can see that the MassConf, MassConf+, and popularity-based ranking *results are actually better in absolute terms than those for the first scenario* (Table 4.4). The reason is that some of the configurations that cannot help many new users are never tried, as the corresponding existing users either join the system too late or not at all.

Table 4.7 compares MassConf and MassConf+ to Simplex for the 55 new users that the latter was able to configure. First, note that MassConf and MassConf+ can configure many more new users than Simplex, even under adverse conditions that have no effect on Simplex. In addition, the table shows that our systems perform 31% and 65% fewer experiments than Simplex, respectively. Again, these results exhibit the same trends as in the first scenario.

4.4 Extrapolating Beyond The Case Study

The results from the previous section are very positive, but they are specific to our Apache case study. In this section, we qualitatively extrapolate from them by *abstracting away* the server software, the high-level tuning goal, the workloads, the load intensities, and the target behaviors.

The extrapolation is based on the observation that three aspects of the user-configuration space matter most in determining the number of experiments for a sequence of new user arrivals: (1) the number of new users that can be satisfied by each configuration; (2) the number of configurations that can satisfy each new user; and (3) the number of new users for which MassConf would have to resort to Simplex.

Adapt-fastest and Adapt-fast behave better than Adapt-slow and popularity-based ranking when there is significant potential for re-use of the configurations that are promoted forward in the ranking. This potential increases when aspects #1 and #2 above are skewed towards a subset of configurations and new users, respectively, and the tails of the distributions are short. In other words, the re-use potential increases when (a) a significant fraction of configurations can satisfy many new users and (b) a significant fraction of new users can be satisfied by many configurations. When the amount of skew is limited or tails are long, Adapt-slow should perform best.

Compared to Simplex, MassConf can benefit from configuration re-use to achieve a lower average number of experiments per new user. Moreover, only a small fraction of new users should require MassConf to resort to Simplex (aspect #3), since the existing and new user populations should have the same characteristics (after bootstrapping).

In our Apache study, we saw significant skew, short tails, and limited use of Simplex by MassConf. Specifically, aspect #1 can be approximated as a power-law function, in which the configuration that can satisfy the most new users satisfies 70 out of 129 such users. Aspect #2 of our Apache study can be approximated as an exponential function, in which the new user that can be satisfied by the most configurations can be satisfied by 52 out of 64 configurations. MassConf had to resort to Simplex for only 7 new users.

We expect real user populations to benefit from MassConf even more than in our Apache study. The reason is that our synthetic population is evenly spread across the workload-intensity-target space; greater concentration in part of the space would increase the potential for configuration re-use. With high potential for re-use, either Adapt-fastest or Adapt-fast would be a good choice; the vendor can select the best approach after configuring a number of new users.

4.5 Summary

In this chapter, we addressed the problem of configuring enterprise software efficiently. Specifically, we proposed MassConf, a system that uses existing configurations to automatically configure the software for new users. The configuration process relies on dynamic adaptation of the order of configurations (ranking) to be tried. To evaluate MassConf, we used it to configure Apache for performance for a population of users. Our results compared three ranking adaptation algorithms to popularity-based ranking. The results showed that our fastest adaptation leads to the smallest number of experiments. The results also showed that MassConf is able to configure more users in fewer experiments than Simplex, an efficient optimization algorithm.

Note that the MassConf results we presented above were obtained for systems that use the same hardware and low-level software, i.e. a single low-level environment. Furthermore, we made the assumption that, in practice, there would be many more users per cluster than clusters. Unfortunately, it is difficult to verify this assumption without access to proprietary data from real software vendors. In addition, it is difficult to predict the impact of our single-environment experiments on the relative performance of our ranking algorithms. Although we believe that our assumption and results should hold in practice, it is conceivable that they would have to be adjusted. We hope that our study will encourage software vendors to repeat our study for their real user data.

Future work. Our future work could address the benefits of MassConf for multi-tier services, rather than stand-alone servers. In particular, we could investigate whether configurations exhibit strong popularity across these systems and what is the best ranking approach for new service installations.

Chapter 5

Related Work

Our work relates to previous efforts in using modeling, feedback control, machine learning, and sandboxing; simulation and emulation of data centers; and scaling down data centers; all in the context of resource and energy management of data centers.

We also make contributions related to configuration parameter relationships; strategies for configuration management; and performance tuning; all in the context of Internet services. Next, we discuss the related works in turn.

5.1 Modeling and Other Approaches for Managing Data Centers

Modeling, simulation, emulation, feedback control, and machine learning. State-of-the-art management systems rely on analytical modeling, simulation, emulation, feedback control, and/or machine learning to at least partially automate certain management tasks, e.g. [9, 19, 25, 27, 31, 32, 55, 64, 71, 75].

More specifically, many works have considered resource management in non-virtualized hosting centers with such goals as service differentiation, increasing revenue, or increasing resource utilization [4, 15, 36, 60, 76]. These works did not benefit from the flexibility, performance and fault isolation, and migration capability that VMs enable. JustRunIt relies on virtualization to enable resource management through experimentation.

In a virtualized hosting center, Padala *et al.* [52, 53] combined analytical modeling and feedback control to adaptively assign resources to VMs, while meeting all SLAs.

As aforementioned, resource management using modeling involves extensive human labor, whereas feedback control has limitations regarding global management of resources and energy management in the face of multi-core CPUs. Our work relies on neither modeling nor feedback control.

Wood *et al.* [83] employ resource usage monitoring and/or application-level SLA monitoring to find resource "hot spots" in VMs. When a hot spot is detected, they estimate the peak resource needs of the corresponding VM, and compute a new mapping of VMs to physical machines using a greedy bin-packing algorithm and the resource information about all VMs. The estimation of peak resource needs involves analytical modeling and/or simple *online* trial-and-error. In contrast to JustRunIt, they did not consider energy management or local resource reassignment. Furthermore, we evaluate resource allocations offline (through experimentation rather than modeling) to avoid interfering with the applications unnecessarily.

CloudScale [61] uses online resource demand prediction, online resource cap changes, and VM migration to minimize SLA violations. When predictions are inaccurate, tentative remedial actions are taken again online. In contrast, JustRunIt's approach is to evaluate resource allocations offline, without affecting the production system, until a more definitive allocation decision is made.

Many works have considered energy management in homogeneous [28, 34, 54, 56] and heterogeneous [32, 42, 58] non-virtualized Internet services. More related to our work, Chase *et al.* [15] and Chen *et al.* [18, 19] studied energy management for hosting centers. In contrast with these works, our approach is to assess the potential energy savings of power management mechanisms and policies through sandboxed experimentation, rather than using modeling or online feedback control.

In summary, modeling has complexity and accuracy limitations. Simulation has some of the same limitations as modeling, such as the need for re-validation as the system evolves, and high slow-down factors as compared to experiments. Besides the need for re-validation, emulation requires enough hardware to realistically represent the real data center. Our experiment-based approach to management has none of these limitations.

Feedback control is not applicable to many types of tasks. Although machine learning is useful for certain management tasks, such as fault diagnosis, it also has applicability limitations. The problem is that machine learning can only learn about system scenarios and configurations that have been seen in the past and about which enough data has been collected. For example, it applies to neither of the tasks we study in JustRunIt. Nevertheless, machine learning can be used to improve the interpolation done by JustRunIt, when enough data exists for it to derive accurate models. In summary, this dissertation takes a fundamentally different approach to management; one in which accurate experiments replace modeling, feedback control, and machine learning.

Scaling down data centers. Gupta *et al.* [29] proposed the DieCast approach for scaling down a service. DieCast enables some management tasks, such as predicting service performance as a function of workload, to be performed on the scaled version. Scaling is accomplished by creating one VM for each PM of the service and running the VMs on an offline cluster that is an order of magnitude smaller than the online cluster. Because of the significant scaling in size, DieCast also uses time dilation [30] to make guest OSes think that they are running on much faster machines. For a 10-fold scale down, time dilation extends execution time by 10-fold.

Although DieCast executes the VMs natively, it only simulates disks and emulates switches. It also relies on application-specific workload generators. DieCast and JustRunIt have fundamentally different goals and resource requirements. First, JustRunIt targets a subset of the management tasks that DieCast does; the subset that can be accomplished with limited additional hardware resources, software infrastructure, and costs. In particular, JustRunIt seeks to improve upon modeling by leveraging native execution. Because of time dilation, DieCast takes excessively long to perform each experiment.

Second, JustRunIt includes infrastructure for automatically experimenting with services, as well as interpolating and checking the experimental results. Third, JustRunIt minimizes the set of hardware resources that are required by each experiment without affecting its running time. In contrast, to affect execution time by a small factor, DieCast requires an additional hardware infrastructure that is only this same small factor smaller than the entire online service.

Finally, JustRunIt is simpler and more practical than DieCast, as it does not require additional emulators, simulators, or workload generators. The large and complex hardware and software infrastructures of DieCast pose many management challenges and substantial extra (labor and energy) costs.

Sandboxing and duplication for managing data centers. A few efforts have proposed related sandboxing and request-duplication infrastructures for managing data centers. Specifically, [41, 45, 46] considered validating operator actions in an Internet service by using request duplication to a sandboxed extension of the service. For each request, if the replies generated by the online environment and by the sandbox ever differ during a validation period, a potential operator mistake is flagged. Further, Tjang [74] validated operator actions against predefined models. Tan *et al.* [69] considered a similar infrastructure for verifying file servers. Ding [24] uses Splitter to detect potential problems by comparing results generated from the production and migrated applications in the virtualized IT environments.

Our work differs from these previous efforts both in terms of goals and software infrastructure. Instead of operator-action validation, our goal is to experimentally evaluate the effect of different resource allocations, parameter settings, and other potential system changes (such as hardware upgrades) in data centers. Thus, our work is much more broadly applicable than previous works.

As a result, our infrastructures are quite different than previous systems. Without workload duplication, ACI proposed to configure an Internet service backed by multiple data centers automatically, by isolating one center at a time (during periods of light load) and experimentally testing different values for the configuration parameters of the (user-level) servers that implement the service. Moreover, JustRunIt is the first system that may explore a large number of scenarios that differ from the online system, while extrapolating results from the experiments that are actually run, and verifying its extrapolations if necessary. These differences also make JustRunIt more practical than previous infrastructures.

5.2 Configuration Management and Performance Tuning

Configuration management. Most previous works in configuration management seek to automatically detect and correct misconfigurations. Validation [41] detects many classes of operator mistakes, including misconfigurations, in Internet services. The Chronus system [81] seeks to identify misconfigurations in stand-alone computers that go undetected for a period of time. Also targeting stand-alone computers, the Glean system [37] infers correctness constraints for configuration entries contained in the Windows Registry. Along similar lines, the Strider system [79] 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, [79] 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 described in Section 3.1.) To eliminate the need for manually selecting correct snapshots, PeerPressure [78] 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. ConfAid [5] uses dynamic information flow analysis to identify the likely root cause of misconfigurations.

ACI 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. In contrast, ACI (almost) completely eliminates operator involvement in many common configuration tasks.

Furthermore, these previous works have focused on stand-alone systems, except for validation, which like our work targets Internet services. Finally, these systems can detect misconfigurations in application and operating system state, whereas ACI focused solely on application configurations. Nevertheless, our work can benefit from these previous systems in their ability to detect and perhaps correct misconfigurations of the operating system state.

Like our infrastructure ACI, several other systems have proposed to generate and manage configurations automatically, e.g. [2, 3, 12]. Cfengine [12] 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 [3] 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 ACI 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 ACI assumes that machines are already running the operating system and focuses solely on service management and evolution. In these respects, ACI is more closely related to SmartFrog [2].

Like Cfengine, LCFG, and ACI, 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 parameters, 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 ACI 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, ACI generates configurations that maximize a specific metric, relying on explicit configuration parameter relationships. SmartFrog does not represent or leverage such relationships.

Performance tuning. Several papers, e.g. [16, 20, 23, 26, 59, 65, 73, 82], have considered performance tuning of Internet services. Diao *et al.* [23] studied the performance tuning of a Web server, using an agent-based feedback control system. Chung and Hollingsworth [20] 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 [43], they find the ideal machine-tier assignment for each workload. Also for a three-tier system, [59] proposed to use an evolutionary algorithm for tuning server configurations. Thonangi *et al.* [73] consider the problem of exploring a large parameter space within a limited budget of experiments. Again considering multi-tier services, Stewart and Shen [63, 65] 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 [38]. Similarly, several works studies performance tuning for database systems [6, 68, 80]. Babu [6] also considered experiments for database configuration tuning independent of our work.

Comparing to previous approaches for performance tuning of Internet services, ACI 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.

Some previous works have proposed sophisticated approaches for selecting the experiments to run when benchmarking servers [62] or optimizing their configuration parameters [73]. Such approaches are largely complementary to our work. Specifically, they can be used to improve experiment-based management in two ways: (1) automated management systems can use them to define/constrain the parameter space that JustRunIt should explore; or (2) they can be used as new heuristics in JustRunIt's driver to eliminate unnecessary experiments.

Leveraging existing data on configurations. Several previous works have investigated how to leverage others' configurations to diagnose and troubleshoot misconfigurations [1, 66, 78, 79]. Strider [79] compares the Windows registry of a misconfigured computer with a known healthy registry state to diagnose misconfigurations. To eliminate the need for manually selecting correct states, PeerPressure [78] compares the registry of a misconfigured computer to those of a population of other computers in the same installation, and uses statistical techniques to rank the configuration parameters that may be the root-cause of the misconfiguration. PeerPressure troubleshoots the misconfiguration by coercing the culprit parameters to the values used by the majority of properly configured users. Along similar lines, NetPrints [1] diagnoses misconfigurations in network applications by applying decision-tree-based learning on the configuration states of a population of users. Solutions to misconfigurations are stored as signatures and can be used by other users to troubleshoot their systems. Autobash [66] provides a set of interactive tools that help users and system administrators manage configurations. Specifically, Autobash leverages previous users' experience by recording their actions and replaying them on different systems speculatively. Su [67] further automatically generates predicates for configuration troubleshooting from user traces.

Even though MassConf also relies on configuration information from a population of users, it focuses on a completely different problem: configuration tuning; there are no misconfigurations to troubleshoot. As detailed in Section 4.2, the impact of this key difference is that our main focus has been on issues that have not been addressed before, namely the study of adaptive ranking algorithms and the average number of experiments to which they lead. In fact, the specific and diverse characteristics of the users' workloads and their behavior targets mean that configuration information is also diverse (i.e., coercion as in [78] does not apply) and prior actions from a user do not produce the same results for another user (i.e., local experiments are necessary). For these reasons, our main focus has been studying adaptive ranking algorithms and the number of tuning experiments to which they lead on average. Neither of these issues was considered by these prior works.

Similarly, Chen *et al.* [17, 16] proposed to obtain a Bayesian network as the byproduct of the tuning process of a system and use this ”experience” to help configuration tuning of the evolved system. Osogami *et al.* [50, 51] focused on shortening each experiment, rather than reducing the number of experiments.

MassConf differs from these works in four main ways: (1) it seeks to produce configurations that meet the users' target behaviors, rather than to find the best possible configuration; (2) it relies on configuration information from a population of systems, rather than a single system; (3) it relies on adaptive ranking algorithms to tune performance efficiently; and (4) unless it needs to resort to Simplex, it tests existing configurations for new users, rather than trying to use experience or dependencies to create new configurations.

Chapter 6

Conclusion

The dissertation addresses an important problem, namely the management of data centers.Management includes a wide range of topics, including performance tuning, system configuration, resource and energy management, and software and hardware upgrades.

The dissertation proposed the notion of experiment-based management, which leverages real systems and workloads to help understand and predict the system behavior. The experimental results can be used for creating new configurations (as in MassConf), for tuning configurations (as in ACI), or for resource management (as in JustRunIt). The advantages of experiments are significant, compared to popular approaches such as modeling and feedback control.

We addressed the key challenges in experiment-based management, namely how to produce accurate experiments and limit the amount of resources and/or time used in the experimentation. In JustRunIt, we leveraged virtualization technology to clone virtual machines and answer what-if questions about them with minimal resource waste. In ACI, we leveraged our proposed parameter dependency graph and optimization to speed up reconfigurations, after systems evolve. In MassConf, we leveraged configuration information from existing users of server software and search heuristics to meet behavior targets. Our evaluations show that our systems can always automate management with a limited number of experiments. In some cases, we can also restrict the number of machines used for the experiments.

As data centers become more complex and cloud computing becomes more popular, management will become an even greater challenge. In fact, approaches such as modeling will likely become impractical due to the sheer complexity of the systems' behaviors, the interactions between different parts of the systems, and the inability of cloud providers to ”see” inside their customers' virtual machines. We hope that our systems and positive results will encourage researchers to continue exploring experiment-based management in these scenarios. For example, sandboxing and reconfiguration will become more challenging as services incorporate an increasing number of interacting tiers. The time spent running more complex experiments may also become a concern for cloud providers who want to minimize the duration of SLA violations. Further research along these avenues (e.g., [77]) will certainly pay off.

Vita

Wei Zheng

1997 - 2001	**Wuhan University, Wuhan, China.**
1991 - 2004	**Wuhan University, Wuhan, China.**
2004 - 2011	**Ph.D. program in Computer Science, Rutgers University, New Brunswick, New Jersey.**
2004 - 2006	Teaching Assistant.
2006 - 2011	Graduate Assistant.

Selected Publications

2007	"Automatic Configuration of Internet Services.". Wei Zheng, Ricardo Bianchini, and Thu Nguyen. Proceedings of EuroSys'2007, March 2007.
2009	"JustRunIt: Experiment-Based Management of Virtualized Data Centers". Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. Proceedings of the USENIX Annual Technical Conference, June 2009.
2011	"MassConf: Automatic Configuration Tuning By Leveraging User Community Information". Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Proceeding of the International Conference on Performance Engineering, March 2011.

References

[1] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2009.

[2] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control. In *Proceedings of the Systems Administration Conference*, 2003.

[3] P. Anderson and A. Scobie. LCFG: The Next Generation. In *UKUUG Winter Conference*, 2002.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2000.

[5] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2010.

[6] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated experiment-driven management of (database) systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2009.

[7] Banu. Tinyproxy. http://www.banu.com/tinyproxy/, 2008.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[9] S. Bouchenak, N. D. Palma, D. Hagimont, S. Krakowiak, and C. Taton. Autonomic management of internet services: Experience with self-optimization. In *Proceedings of the International Conference on Autonomic Computing*, 2006.

[10] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classfication and Regression Trees*. 1984.

[11] A. B. Brown, A. Keller, and J. L. Hellerstein. A Model of Configuration Complexity and its Application to a Change Management System. In *Proceedings of the International Symposium on Integrated Network Management*, 2005.

[12] M. Burgess. Cfengine: A site configuration engine. 1995.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[14] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2005.

[15] J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[16] H. Chen, W. Zhang, and G. Jiang. Experience Transfer for the Configuration Tuning in Large Scale Computing Systems. In *Poster Session of the International Conference on Measurement and Modeling of Computer Systems*, 2009.

[17] H. Chen, W. Zhang, and G. Jiang. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering*, 2011.

[18] S. Chen, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and W. H. Sanders. Cpu gradients: Performance-aware energy conservation in multitier systems. In *Proceedings of the International Conference on Green Computing*, 2010.

[19] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2005.

[20] I. Chung and J. K. Hollingsworth. Automated Cluster-Based Web Service Performance Tuning. In *Proceedings of IEEE International Symposium on High-Performance Distributed Computing*, 2004.

[21] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, 2005.

[22] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2007.

[23] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal*, 42(1), 2003.

[24] X. Ding, H. Huang, Y. Ruan, A. Shaikh, B. Peterson, and X. Zhang. Splitter: a proxy-based approach for post-migration testing of web applications. In *Proceedings of the European Conference on Computer Systems*, 2010.

[25] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[26] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *Proceedings of International Conference on Very Large Data Bases*, 2009.

[27] T. Dumitras, P. Narasimhan, and E. Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[28] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Proceedings of the Workshop on Power-Aware Computing Systems*, 2002.

[29] D. Gupta, K. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, 2008.

[30] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, G. M. Voelker, and A. Vahdat. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, 2006.

[31] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[32] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[33] L. J. Heyer, S. Kruglyak, and S. Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 9(11), 1999.

[34] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic Voltage Scaling in Multi-tier Web Servers with End-to-End Delay Control. *IEEE Transactions on Computers*, 56(4), 2007.

[35] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2008.

[36] B. Khargharia, H. Luo, Y. Al-Nashif, and S. Hariri. Appflow: Autonomic performance-per-watt management of large-scale data centers. In *Proceedings of the International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing*, 2010.

[37] E. Kiciman and Y.-M. Wang. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the International Conference on Autonomic Computing*, 2004.

[38] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 1983.

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

[40] K. Nagaraja, G. M. C. Gama, R. Bianchini, R. P. Martin, W. M. Jr., and T. D. Nguyen. Quantifying the Performability of Cluster-Based Services. *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[41] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[42] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting Platform Heterogeneity for Power Efficient Data Centers. In *Proceedings of the International Conference on Autonomic Computing*, 2007.

[43] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 1965.

[44] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[45] F. Oliveira. *Towards mistake-aware systems*. PhD thesis, Rutgers University, 2010.

[46] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[47] F. Oliveira, J. Patel, E. V. Hensbergen, A. Gheith, and R. Rajamony. Blutopia: Cluster Life-Cycle Management. Technical Report RC23784, IBM Austin, 2005.

[48] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[49] D. Oppenheimer and D. Patterson. Architecture and Dependability of Large-Scale Internet Services. *IEEE Internet Computing*, 2002.

[50] T. Osogami and T. Itoko. Finding Probably Better System Configurations Quickly. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2006.

[51] T. Osogami and S. Kato. Optimizing System Configurations Quickly by Guessing at the Performance. *SIGMETRICS Perform. Eval. Rev.*, 2007.

[52] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the ACM European Conference on Computer Systems*, 2009.

[53] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the ACM European Conference on Computer Systems*, 2007.

[54] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Dynamic Cluster Reconfiguration for Power and Performance. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, 2003.

[55] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No Power Struggles: Coordinated Multi-level Power Management for the Data Center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[56] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.

[57] Rice University. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[58] C. Rusu, A. Ferreira, C. Scordino, A. Watson, R. Melhem, and D. Mosse. Energy-Efficient Real-Time Heterogeneous Server Clusters. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[59] A. Saboori, G. Jiang, and H. Chen. Autotuning Configurations in Distributed Systems for Performance Improvements Using Evolutionary Strategies. In *Proceedings of the International Conference on Distributed Computing Systems*, 2008.

[60] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002.

[61] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, 2011.

[62] P. Shivam, V. Marupadi, J. Chase, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[63] C. Stewart. *Performance modeling and system management for Internet services*. PhD thesis, Rochester University, 2009.

[64] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A Dollar from 15 Cents: Cross-Platform Management for Internet Services. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[65] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, 2005.

[66] Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2007.

[67] Y. Su and J. Flinn. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[68] D. G. Sullivan. *Using Probabilistic Reasoning to Automate Software Tuning*. PhD thesis, Harvard University, 2003.

[69] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based File Server Verification. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[70] The Apache Software Foundation. Apache HTTP Server Version 2.0. http://httpd.apache.org/docs/2.0/mod/prefork.html.

[71] E. Thereska. *Enabling What-If Explorations in Systems*. PhD thesis, Carnegie Mellon University, 2007.

[72] E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2008.

[73] R. Thonangi, V. Thummala, and S. Babu. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2008.

[74] A. Tjang, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Model-based validation for internet services. In *Proceedings of the Symposium on Reliable and Distributed Systems*, 2009.

[75] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions on Adaptive and Autonomous Systems*, 2008.

[76] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002.

[77] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. Dejavu: Accelerating resource allocation in virtualized environments. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[78] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[79] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of Large Installation Systems Administration Conference*, 2003.

[80] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[81] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[82] J. Wildstrom, P. Stone, E. Witchel, R. J. Mooney, and M. Dahlin. Towards Self-Configuring Hardware for Distributed Computer Systems. In *Proceedings of the International Conference on Autonomic Computing*, 2005.

[83] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2007.

[84] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2011.

[85] W. Zhang. Linux Virtual Server for Scalable Network Services. In *Proceedings of the Linux Symposium*, 2000.