Thesis proposal: An automated approach for mitigating service performance problems with efficient resource allocations

Elie Krevat
Computer Science Department
Carnegie Mellon University

Abstract

Distributed and cloud computing services are increasingly built atop a preexisting infrastructure of shared services. These services have separate performance characteristics and require enough resources to support each application's service level objectives (SLOs), while preferably not wasting too many resources from overprovisioning. This document describes an automated approach to mitigating performance problems through reactive resource provisioning. When a problem occurs, whether it be from a service performance change or an increase in load, we attempt to mitigate the problem in the short term through automatic resource assignments. Our proposed approach makes use of end-to-end request traces, resource usage measurements, and system feedback to anticipate the types of resources that are needed and to determine services that can usefully apply them.

1 Background and motivation

1.1 System management challenges

The task of managing resources over many connected shared services with complex performance characteristics has become increasingly difficult for even the most skilled system administrators. Changes in system performance can occur a few times a day for any number of reasons, such as from modified system configurations, service upgrades, hardware failures, or increased loads. When performance changes occur, shared resources need to be reallocated fast and effectively across services to relieve resource bottlenecks.

When there are only a few services and computing resources are dedicated, responding to performance problems is fairly tractable. If the problem is a resource bottleneck, then typical shared services are designed to scale well over many machines so many performance problems can be improved by assigning more computing resources or better machine types. However, as the number and inter-relationships of these services grows, determining where and what resources to apply efficiently, without costly overprovisioning, is a more difficult problem that we wish to tackle.

Overprovisioning is often tolerated because of the difficulty of avoiding it and the importance of maintaining steady performance. Large services such as those at Google and Amazon are expected to respond quickly to customers, even with thousands of concurrent requests of varying types. If a service cannot respond in time to meet its service level objectives (SLOs), there can be significant consequences (e.g., financial penalties, lost business, and disappointed customers).

Due to its complexity, resource allocation is all too often a manual and ad-hoc activity performed by system administrators in response to observed changes in the system. Overprovisioned services can avoid certain performance problems, but at large costs. Statically allocated systems may operate at certain administrator-tuned thresholds (e.g., five memcached machines for every ten application servers), but then the composition

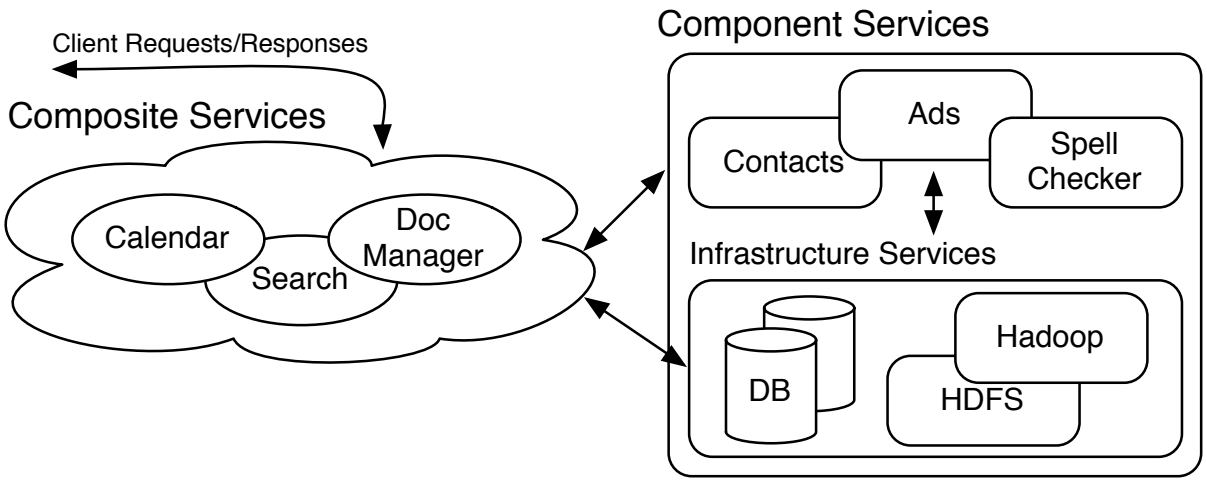

Figure 1: Shared services architecture. *Composite services* are front-end customer facing services that are selectively composed of *component services* (and other composite services) that provide answers to subqueries and may provide other infrastructure services to access data. Dependencies occur as services are built atop others.

of requests can change in production. Even dynamically-allocated systems can rely on logical but arbitrary heuristics that are wasteful because they do not account for the actual source of latency problems (e.g., add two machines to every service when request timeouts are observed). None of these approaches are satisfactory.

1.2 Target system architecture: component and composite services

As shown in Figure 1, the complexity of services has increased with the evolution of many *component services* that may be selectively used for specialized purposes (e.g., location service, ads generator). Google, for example, has such a large and diverse set of essential platforms and services that, even back in 2007, a single web search touched roughly 50 services and 1000s of machines [8].

Component services are building blocks for *composite services* that perform compound tasks typical for large Internet services (e.g., search engine, cloud document manager). Some component services are more specifically classified as *infrastructure services*, which have replaced the traditional data tier with multi-layered shared storage services (e.g., Hadoop [3], HBase [4]). Each component service may be selectively used by a composite service, and there are dependencies both within and across these types of services.

More formally, the system architecture includes:

c types of component services (s_1,s_2,...,s_c)

n types of machines (m_1,m_2,...,m_n)

q types of composite or component requests (r_1,r_2,...,r_q), where composite requests require sequential or parallel responses from many different component services

Target systems should also have the characteristic that resources have a cost and that there is not an overabundance of them. In typical environments, there can also be many requests from different sources running concurrently.

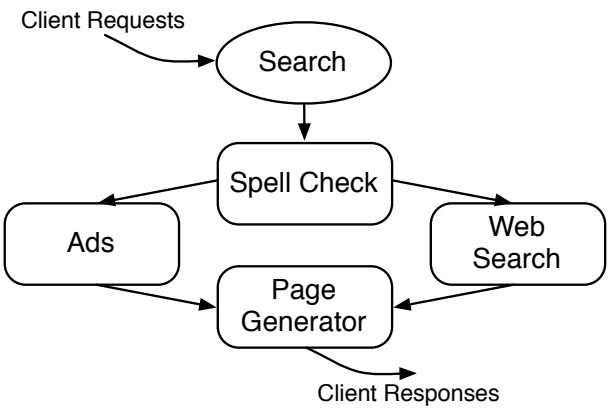

Figure 2: Simple search example. A simplified composite search service feeds client requests first into a spell checker to filter the request, and then in parallel to an ads generator and a web search service. After a page generator service receives both responses, it ranks the results and generates a response to the client.

1.3 System monitoring and end-to-end tracing

To deal with system complexity, there has been an emergence of better monitoring tools such as end-to-end request tracing that informs system administrators of the health of the system. End-to-end request tracing provides timing and path dependency information (e.g., latency and throughput). After some processing, these traces are traditionally used to find sources of latency [31] or to discover structural or response-time mutations [33]. End-to-end tracing frameworks are not ubiquitous, but are becoming more popular (e.g., Carnegie Mellon's Stardust [39], Google's Dapper [35] and Twitter's Zipkin [9]). These infrastructures have demonstrated that tracing data can be sampled and collected with minimal overhead (e.g., Stardust's overhead is less than 1% for the SpecSFS [34] benchmark running on the Ursa Minor distributed storage system).

Resource usage and performance statistics can also provide a stockpile of information to discover resource bottlenecks. Administrators can be alerted when a particular resource threshold is exceeded (e.g., CPU utilization above 90%), which can inform both the reasons for service delays and a good plan of response (e.g., supply more or faster CPUs).

However, much of the analysis of this monitored data is still done ad-hoc and manually, if at all. Often this information, if collected, is stored individually by each service or machine in separate silos of log files. More sophisticated setups consolidate everything into a common database for offline analysis. Absent an automated solution, system administrators are the experts that understand the system and have learned ways to make some sense of monitoring information to fix problems as they arise. These experts will always be useful in some capacity, but they increase the cost of ownership of a system and can be difficult to identify and retain.

1.4 Issues with uninformed allocations

Consider a complex composite search service from the viewpoint of clients that are affected only by the content of the response and the total end-to-end latency. See Figure 2 for a simplified illustration. If the search service is composed of separately managed component services that are necessary for it to function (e.g., an advertising service, map service, web search service, etc.), traditionally each service independently allocates its resources based on local information. But, how would local service times affect the big picture from the client's perspective? That would not be clear, since the advertising service may see its query latencies

double from 10ms to 20ms under higher load but not understand how critical this change is to the total search time (e.g., the ads might be served in parallel with the web search generation, which takes 200ms). So, is it necessary after all to maintain stringent performance criteria on the ads service (e.g., must 99% of ads requests really finish within 15ms)? This is the setting for uninformed and ad hoc resource allocation.

1.5 Our proposal

We argue for an automated approach to mitigating performance problems in shared service environments through reactive resource provisioning. We are not predicting when a performance problem will occur, or even diagnosing the root cause when a problem occurs. Instead, once a problem occurs, we look for mechanisms that alleviate the problem in the short term by assigning the right types and quantities of computing resources across many component services that can use them—a process we call *right-sizing*. In the longer term, if there is a larger issue to address, experts can leverage the vast literature on problem localization to further pinpoint and fix the underlying root cause.

The mechanisms that we describe in this proposal look to improve upon basic uninformed and manual resource allocation by exploiting end-to-end request flows and resource usage measurements. With end-to-end request flow tracing we can determine the actual service flow and synchronicity requirements. With resource usage statistics we can understand more clearly the resource demands of each service enough to deduce, for example, that the ads service would not benefit from running on a server with more CPU cores because CPU usage is only at 50% of current capacity. And, with both sets of information, we can anticipate the types of resources that are needed as well as the best locations to apply them.

Besides our goal of minimizing human administration, we also believe that good resource assignments can be made without requiring prior models or detailed descriptions of service behaviors, which is important because such information rarely exists. Our general approach does not specialize for a particular service or modify the underlying algorithms. We can treat service internals as a black box and focus on communication across service boundaries. Specifically, we can use end-to-end tracing data of just Remote Procedure Calls (RPCs) to infer service dependencies and performance statistics.

The rest of this proposal is organized as follows. Section 2 explains the thesis focus and validation plan, as well as some sample scenarios with which we plan to experiment. Section 3 describes our monitoring and allocation strategies in more detail. Section 4 surveys related work. Section **??** concludes with a timeline of important milestones.

2 Thesis focus and validation

2.1 Thesis statement

This thesis will demonstrate an approach to mitigating performance problems in shared services by automatically applying resources where they can be used most effectively. It is intended for a live and elastic system of services, like backend services now common at companies like Google, Facebook, and Amazon.

Specifically, the thesis statement is:

An integrated request tracing and resource monitoring framework, combined with a right-sizing feedback mechanism, is an effective tool for automated resource assignment across shared services.

We will explore the relative fitness of a spectrum of static and dynamic strategies, where our most informed strategy combines end-to-end tracing and resource usage monitoring into an iterative *right-sizing* approach.

Monitoring data and past performance statistics instruct our approach, so that when performance problems surface, we are better prepared to mitigate the problem. Since no reallocation plan can be guaranteed to work with stateful and dynamic services, we also fall back on immediate system feedback to more effectively shape and assess our response.

2.2 Assumptions, goals, and non-goals

An open system is assumed instead of a closed system, because the setting assumes many non-interactive requests arriving concurrently into top-level services from many different users. Request rates may change unexpectedly at any time, but we are focusing on situations where the change persists long enough that a reaction makes sense (i.e., not very brief transient load spikes). Services are dynamic and stateful with varying performance. There are different types of machines with various properties that can be assigned, and allocations are made in whole machine units. We assume that a few additional spare machines are free for immediate allocation (e.g., because they are running useful but non-critical batch jobs).

We also make a number of simplifying assumptions to assist with initial validation efforts, which we will later relax. Initial experiments will start with constant request rates, only one type of machine, and with a focus on services that bottleneck on the CPU.

In order to come up with a general and tractable strategy, we treat the internal workings of each service as a black box and focus on measurable end-to-end and performance statistics. Each service can depend on many different subservices to respond to a particular request type, and the services are stateful, so our approach should not specialize for particular service types. We do require scalable services, or a mix of scalable and non-scalable services where the scalability properties of each service can be assessed during the course of experimentation and observation.

The actual mechanism for gathering end-to-end trace information is not a focus of this thesis, as we can leverage existing systems (i.e., X-Trace [19] for end-to-end traces and Flume [2] for distributed aggregation) to meet our needs. We will need to extend these systems to integrate resource usage monitoring information, to scale up with our elastic workloads, and to store data into a decentralized datastore (i.e., HBase [4]). However, we expect the novelty of this work to be in the use of this information and not the engineering efforts necessary to gather it quickly and concisely.

We avoid complex modeling efforts that cannot be generalized, but recognize the usefulness of simple queuing-based operational laws like the Utilization Law and Little's Law to predict response times [22]. For example, a simple service can be specified as $M/M/m/PS$: request inter-arrival times and service times are exponentially distributed, over m machines, with a processor-sharing (PS) queuing discipline. From Little's Law, given the service rate μ and the arrival rate λ, the average response time RT would be:

$$RT = \frac{1}{m\mu - \lambda} \tag{1}$$

Simple models like the one above can be used to predict the additional benefit of another machine applied to a service. However, they are not always accurate, and are not the focus of this thesis. We plan to use any input from queuing-based models as secondary guidance, falling back on an iterative feedback mechanism as our primary tool.

2.3 Validation plan

Validation of the thesis will proceed as follows:

- I will build a simplified system of mock services and workloads (Section 2.3.3) that will mimic the kinds of inter-service connectivity patterns that are now common in enterprise-scale environments.
- The simple service environment will be instrumented with a scalable end-to-end request tracing framework that is also extended to provide resource usage statistics (Section 2.3.1).
- The instrumentation will be incorporated into a few different resource allocation strategies (Section 2.3.2 and Section 3).
- The effectiveness of these resource allocation strategies will be evaluated by introducing synthetic performance problems into the system (Section 2.3.3).

2.3.1 Monitoring information

The end-to-end monitoring framework needs to discover request dependencies across the relevant services. To do that, it will summarize the dependency path of the services involved for each request type (i.e., a *service summary*), as well as a breakdown of the request types that flow through each service (i.e., a *request summary*). Metrics of interest, with average and 99% quantiles, include:

- Arrival rate λ (requests per s)
- Service rate μ (requests per s)
- Service time $s = \frac{1}{\mu}$ (ms)
- Connect time c (ms)
- Response time $r = c + s$ (ms)
- Error count (e.g., from request timeouts)

Queueing delays because of oversubscribed resources can show up in connect time measurements from ingress to a service. Additionally, in contrast to the theoretical notion of service time that doesn't include queuing delays, in practice the measured service time may also include delays from resource contention or internal locks within a service.

Resource usage monitoring includes CPU utilization, memory, and disk and network I/O. We plan to measure these values on a per-request basis by extending X-Trace records to include more monitoring information.

2.3.2 Allocation strategies

Our feedback loop approach for assigning resources will explore a number of techniques that vary in their knowledge of the system. None of our planned strategies rely on the internal workings of a service. The strategies that we compare vary from static black box techniques with no end-to-end information and only static allocations, to dynamic gray box techniques with request flow information or request usage statistics. Roughly, from least to most informed, some possible strategies that we will explore are:

- *Equal layout (static):* Each service receives a fixed and equal number of computing resources.

- *Resource Proportional (static):* Using just the observed average resource usage of one instance, each service is given a fixed number of servers in proportion to its observed average resource usage for a number of concurrent requests. We leverage related work on resource scheduling and consider the dominant machine resource [21] when determining these proportions.
- *Resource Aware:* With only resource usage statistics at our disposal, services that appear bottlenecked or have unusually high usage of a resource are assigned additional resources.
- *Request Aware:* With just the incoming request rates at every service, but no information about the flow of requests across services and the particular demands of each request, we can apply machines in a less informed manner.
- *Performance Feedback:* Depending on request flow statistics, a feedback loop will attempt to improve each service based on historical performance measurements. Candidates for additional resources are determined by comparing recent connect and service times (which include queueing delays) with historical averages and minima on a per-request type basis. The adaptability of each service to respond to additional resources must be learned over time.
- *Performance Feedback + Resource Aware:* Combining request flow measurements with resource usage statistics should allow for the best ranking of candidates to attempt resource changes and to pick a good mix of machines within the feedback loop.

For the static techniques, there are limited-resource and overprovisioned variants. The limited-resource variant should have just enough resources, with a good resource assignment, to maintain SLO response times. The overprovisioned variant will supply enough resources so that applying the same resource assignment from the limited-resource variant will achieve 60% capacity. This number comes from studies of datacenters at large companies [37] as well as discussions with practicing engineers [1]. Other related work [40] has considered a similar policy for overprovisioning that targets 70% utilization when serving requests at the maximum recent load.

2.3.3 Evaluating strategies for different services

Assessing the performance of these strategies is as much contingent on the types of services and workloads in the system as it is on the efficacy of the automated assignments. To that end, we will need to sufficiently categorize different types of services and workloads and how they are affected by additional resources. Services that are not able to apply additional machines should be recognized by the automated feedback mechanism, either through hints from a system administrator or through trial and error.

Automatic assignment and delegation of resources occurs most readily for perfectly elastic CPU-bound services. Not only do these services scale up well with additional resources, but assigning these resources within the service can be as easy as booting up and adding another service instance behind a load balancer. More complicated services have stateful conditions that affect elasticity properties. For example, distributed storage systems need to manage data movement which can be done faster and more costly if necessary, or slower and less obtrusively as a rate-limited background task. Therefore, we will evaluate our strategies against perfectly elastic services, perfectly inelastic services that must be recognized as such, and some middle ground of complex stateful services that need to be satisfied with the right mix of resources.

To demonstrate our ideas, we propose an experimental setup with a few different types of core services over 10s of machines. We will build up a system of these simple but representative services:

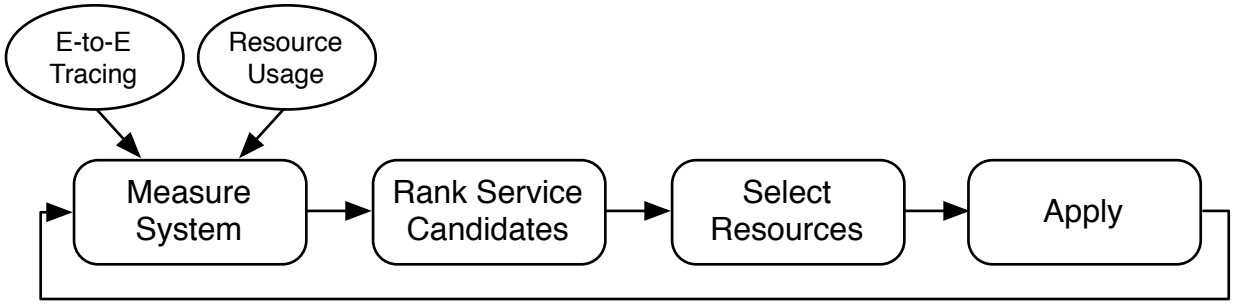

Figure 3: Proposed workflow for iterative right-sizing. The workflow begins with measuring the end-to-end performance characteristics and resource usage of all services. We then identify and rank component services that are in need of resources, and select the right mix of resources for them. Resources are then applied in a feedback loop that measures the effect of assignments on system performance.

- CPU-intensive java servlets
- In-memory database lookup (e.g., dictionary service)
- Distributed elastic storage (e.g., JackRabbit [15])
- Forwarder and aggregator services that coordinate other component service requests

A number of possible request types will travel between these services in a pre-determined manner, representing inter-service dependencies and allowing for requests to arrive from many different sources at once, as is typical in a shared service environment. Services will be run on Tomcat [6] servers and through standalone java RPC services built with Thrift [5] libraries. We would also like to externally validate our approach on other systems, but are still looking for good candidates.

There are a few types of synthetic scenarios that we plan to study. To simulate a slowdown to a service (e.g., caching effects), we will double the response times of a key component service. To simulate a typical large and high load increase that can happen when an application or web page suddenly becomes popular (e.g., the slashdot effect), we will perform a one-time increase in the request rates of a high-level composite service by a large amount (e.g., 10x). To simulate a new service dependency that is added by a code change, we will change the request flow behavior so that requests start to cycle synchronously through a new service that initially has only a few requests.

Our approach will be deemed successful under the following criteria. First, our monitoring framework should provide performance and resource statistics that measurably localize areas of performance changes, but need not analyze the problem source. Second, some combination of strategies should effectively react to and mitigate performance changes in an automated way: settling within 10s of seconds on a good resource allocation that brings the system under an externally-specified SLO threshold for latency response times. A good resource allocation is one that not only returns a service to its SLO performance fast enough (i.e., to improve user happiness), but also one that doesn't waste too many shared resources in the process (i.e., to improve system happiness). These tradeoffs between effectiveness, speed to recovery, fairness, and resource efficiency will need to be explored.

3 Monitoring, analysis, and allocation mechanisms

Figure 3 illustrates the high level resource allocation workflow that is invoked when a performance problem is recognized in the system. The system is measured with end-to-end tracing and resource usage data. From these measurements, performance analysis is done to rank the service candidates that are in need of resources. Once a good service candidate is identified, more machines are assigned and service latencies are measured again as part of an iterative feedback loop.

Section 2.3.2 listed several strategies that we plan to explore. The rest of this section will expand on the steps involved in the *Performance Feedback + Resource Aware* strategy, which addresses the following problems:

Primary problem: After a particular composite request type undergoes a change in behavior (observed externally to our tool), determine how many machines of each type to apply to every component service in order to satisfy an SLO.

Secondary problem: After applying resources, check the forecast of improved performance with the actual results on the system. Either 1) confirm success, 2) predict failure due to resource shortage or service inelasticity, or 3) confirm partial improvement, refine forecast, and request more machines to apply.

3.1 Initiating the right-sizing loop

The first steps of the workflow are to detect a performance problem and to initiate the right-sizing loop.

3.1.1 Detecting performance problems

A simple detector can recognize when an SLO violation has occurred or is close to occurring. A typical invocation might occur when a composite service violates its SLO because requests are delayed on the critical path by a downstream component service running at full capacity. There are also more advanced early warning detection schemes that can predict violations based on other symptoms. Or perhaps the detection is made by a system administrator who specifies that "there is a problem with service type S starting at time T," and we would want to apply our right-sizing feedback loop to restore S to its performance before T.

Our work is orthogonal to whatever problem detection approach is utilized. We aren't innovating or advocating any particular approach, but we just need a reason to initiate our right-sizing loop with the aim of improving a particular service type in a problem state, and a way of determining how much improvement is necessary until performance is restored (e.g., within a 200ms SLO threshold).

3.1.2 Inputs to the right-sizing tool

We require the following pieces of information as inputs:

M_i: The number of machine instances assigned to service s_i, as a vector of each machine instance type m_j, for all i and j

M_{avail}: The number of machine instances that are available for reallocation, as a vector of each instance type

$RT^{avg}_{SLO}(t)$: The largest acceptable average response time of a type t request in this system that meets the service level objective

$RT_{SLO}^{99th}(t)$: The largest acceptable 99th-percentile response time of a type t request in the system that meets the service level objective

3.2 Measuring and analyzing the system

Measuring the system involves three primary tasks: discovering the performance dependencies and paths of requests between services, measuring important performance statistics (e.g., response times), and measuring resource usage statistics (e.g., CPU utilization). Because services can handle many different types of requests with varying behaviors, it is necessary to differentiate between request types to produce more accurate statistics. We rely on end-to-end traces, extended to include additional resource usage measurements, as our primary tool for achieving all of these tasks.

Using the measurements, service performance statistics are analyzed by comparing average and 99% response times to historical (weighted) averages. We also associate performance statistics with historical allocations of resources.

3.2.1 Reconstructing request flows

In typical large clusters with shared services, each service generates a set of requests with dependencies on other services. Any one service may generate different classes of requests, but the general path structure of each class is fixed with some probability. For example, a valid read request will always either hit in the cache with some probability or else initiate a disk read at a storage service node.

End-to-end tracing is an existing technique to track the dependencies and paths between different components of a distributed system. This is in contrast to traditional log files that only measure local information pertaining to a specific service, or general tracing mechanisms that store lots of unstructured information from different components without causal relationships. End-to-end traces designate a number of trace points throughout the system and store *activity records* with request names, the current time, and other contextual information like request types. During runtime, the tracing mechanism assigns a unique ID to each request and tracks the movement of requests across service boundaries. By stitching together activity records, one can reconstruct the full path of each request across service boundaries, the time that each request spent within a service, and the path of generated subrequests.

Many systems have been instrumented with different end-to-end tracing infrastructures and have demonstrated their effectiveness with minimal impact on the running workloads [14, 19, 38, 39]. We specifically plan to use X-Trace [19], a popular tracing library that requires very little tracing infrastructure to handle different services. In order to get full cluster-wide information, at the gateway of incoming and outgoing requests at each service, X-Trace library calls can link the generation of new RPC calls to the parent request. We have extended X-Trace to use Flume [2] as an aggregator that feeds into an HBase table store.

3.2.2 Appending resource usage

We plan to append resource usage statistics directly to the request flow traces, instead of storing it through a separate monitoring infrastructure. This has the benefit of tying resource usage to particular requests instead of just general machine-wide measurements. The downside of this approach is that it adds extra steps to retrieve resource usage from the tracepoints. During periods of low activity, it can be more efficient in terms of the amount of records kept, but during periods of high activity many different tracepoints providing the same resource measurement (e.g., 100% CPU usage) may be redundant.

3.2.3 Comparing performance statistics

Full end-to-end performance statistics provide a good deal of help in localizing performance changes to particular services. Without this information, if we had to consider each service separately, like a strawman example for a multi-tiered service in related work [42], then we might find ourselves applying changes in cascading fashion in many different areas before the actual bottleneck is improved (if at all).

With end-to-end tracing, we can pinpoint where response time changes happen by comparing per-request type per-service performance over time. Even when the resource allocation workflow is not explicitly running, the performance of requests are still being traced. We can group these request traces into separate time intervals (e.g., 5s) to help control for transient performance spikes and irregularities.

3.2.4 Gauging elasticity

It is important to recognize the limitations of services that cannot respond as well to specific resources. Historical information will show how well a service adapted to new resources in the past (e.g., with faster or more stable response times) and how long it took for changes to take effect. Care must be taken to control for changes in load that could have had more of an effect than any resource changes. Services that have shown less elasticity in the past will be weighted accordingly when ranking candidate services for more resources. As a further optimization, hints from system administrators, such as marking specific services as unparallelizable, can be useful to more quickly improve the quality of resource assignments. However, even without these hints, our system should discover this information automatically through trial resource assignments as part of the feedback loop.

One important measurement that we store is a *reaction time* statistic based on historical changes of allocations made to each service, and the time it took to recognize an actual difference in request latencies. As long as the request flows are not naturally very bursty or irregular, then changes caused by performance problems or different resource allocations should become apparent, if they helped at all.

3.3 Ranking service candidates

The process of ranking services is meant to identify those services that have the largest slowdown effect on composite service requests and also to have the greatest opportunity to make use of additional resources to improve their performance.

3.3.1 Analyzing slowdown in the critical path

In order to find those services that are most responsible for the slowdown of this particular composite request type, we analyze the average end-to-end request traces and sum up the time spent connecting to and being processed by each component service. Component services that consume a relatively greater amount of time on a composite service's critical path offer particularly good opportunities for performance improvements.

Considering some of the subtle communication patterns of synchronous and parallel operations is an important part of this task. For example, in the simplified search example of Figure 2, if the client response is stalled on a response from parallel queries to the ads generation service and the web search service, with the web search service taking the most time, then improving web search to be as fast as the Ads service will be beneficial, but improving it to be faster than the ads service will have no additional effect on the client.

3.3.2 Predicting improvement opportunities

Next, we apply weighting factors that modify the initial ranking based on our prediction of the opportunity for each service to see the greatest potential improvement with additional resources. The following factors affect these weights:

- The service's elasticity properties, as measured by the *reaction time* statistic. Large reaction times correspond to inelastic services.
- How closely current performance matches historical min, average, and max values, and especially factoring in recent changes in response times. A large increase in short-term response time that hovers near a max historical response time value suggests a big problem.
- A rough assessment of the service's current queuing delay. For example, there could be an opportunity for a 2x performance improvement if the current response time, which includes queueing delays and connect time, is twice as long as the service time. We can also make use here of simple queuing theory models like Little's Law [22].
- Preference for services with fewer machines already allocated, not for fairness, but because the expected effect of adding another machine is greater.

Historical ranges need to be used carefully. On the one hand, they give a picture of the best and worst case performance of a service. On the other hand, a service's response time characteristics can change when the service is modified (e.g., a new binary version is pushed through that slows down all requests with more processing). Therefore, historical ranges need to favor recent measurements.

3.4 Selecting resources

Right-sizing involves adjusting both the quality and the quantity of computing resources. The quality of resources may be adjusted by assigning better computing instances (e.g., faster CPU, more memory). The quantity of resources may be adjusted by revoking or assigning different quantities of each instance type. Some services are not parallelizable; therefore, request latencies can only be improved with better instance types. Other services may be bottlenecked on a particular resource, so providing it more of the wrong resource will not be helpful.

When resource usage is being monitored, we have an opportunity to select more precisely from heterogeneous resources to give a service more of the resource that it is bottlenecked on. Otherwise, only through trial and error can we determine if a service responds well to a particularly fast or large resource.

3.5 Assigning resources

Machines are provisioned between many services, and a central cluster scheduler is assumed to be responsible for managing services and accounting for resource usage. The cluster scheduler offers a mechanism for changing resource allocations but does not force its own policies (such as an equal sharing layout). For our purposes, this scheduler exposes methods for assigning different bundles of resources at course resource granularity (e.g., 4 CPUs and 12GB RAM), but assigns those resources without topology awareness (i.e., not explicitly from machine X or rack Y). While more powerful cluster schedulers exist, such as planned for Google's next generation scheduler [45], this simpler allocation model is commonly used by cloud services

such as Amazon's Elastic Compute Cloud (EC2) [11]. Amazon simplifies the possible types of virtual machine allocations even more into small or large CPU- or memory-rich bundles, which is a simplification that we also adopt for our experiments.

We make specific use of the Open Cirrus research testbed for our experiments, which has 78 nodes running KVM [7] coupled with the *Tashi* and *Zoni* services for managing virtual machine instances and physical machine allocations [26]. Each physical machine has 2x quad-core 2.66 GHz CPUs, 16 GB RAM, and 2x 1 TB SATA drives, networked with 1 Gbps Ethernet.

In order to assign cluster resources, we distribute a variable number of pre-configured bundles of resources. Better granularity within a single machine to account for different compositions of resources would be useful, but is ignored for now as a simplification. We focus on four instance types, with specific values assigned to match the hardware capabilities that are available to us on the Open Cirrus testbed.

1. *Fast-CPU Large Instance*: 4x 2.66 GHz CPUs, 12 GB RAM.
2. *Fast-CPU Small Instance*: 2x 2.66 GHz CPUs, 4 GB RAM.
3. *Slow-CPU Large Instance*: 4x 2 GHz CPUs, 12 GB RAM.
4. *Slow-CPU Small Instance*: 2x 2 GHz CPUs, 4 GB RAM.

Each physical node on our Open Cirrus testbed can either host one large instance or two small instances. As a practical matter, four CPUs remain unallocated on each physical server because spare CPU cycles are required to process network headers for each virtual server. Since all the CPUs on this cluster are the same speed, we need to explicitly frequency scale CPUs for the slower CPU instances to create a more heterogeneous environment.

3.6 Feedback loop and termination

An iterative feedback loop is used to determine the actual effectiveness of the applied changes to resource allocations, repeating the previous steps after a small period of time that allows for the changes to take effect. This length of time varies depending on the elasticity properties of the service that was modified, and this is where we make use of the *reaction time* statistics that were previously measured.

Once the composite service has returned to an acceptable operating region (meeting its SLO), then the feedback loop is terminated. If the composite service does not improve, then it must be pinpointed to a particular component service that is not improving – or perhaps it does improve but not sufficiently, and the amount of available machines have been exhausted. In this case, the particular component service will be flagged as the culprit for a system administrator to address. It is possible in this last case that our feedback loop could explore targetted revocation of resources from one service and reallocation to another.

4 Related Work

Related work that provides building blocks essential to our approach (e.g., end-to-end tracing) has been discussed in the flow of the document. There has also been work that addresses similar problems, or that uses similar techniques, which is described in this section.

Finding sources of latency delays: Our approach does not attempt to explain latency changes, only to pinpoint the component services that could most benefit from more resources in addressing a composite

service SLO issue. A few related projects have looked at using historical data in similar large systems of services to discover the sources of errors or latency delays in an offline manner, as opposed to our online feedback approach. Latency errors are hard to find because it can be difficult to determine which RPC requests are fully parallel or otherwise dependent on each other and require synchronization. Google's approach uses historical Dapper [35] end-to-end tracing and a nearest-neighbor estimator to find clusters of request flows that are similar, and perform a very large search on each type of flow to modify the frequency and timings of requests to try and explain observed changes in behavior [31, 30]. LinkedIn's approach does not have timing information for each request, but does pass along a common request identifier to find dependencies [25]. They then use sensor measurements at each component service to find patterns of changes during an (externally-provided) anomalous time period, solving an optimization problem to find the most similar sensor readings. The advantage of this latter approach is that correlations can be found where two services were collocated and interfered with each other but otherwise did not query each other as part of a request flow. However, the former approach, which is closer to our setup, has access to per-request timing and dependency/happens-before information that can reveal latency problems.

End-to-end tracing data has previously been applied to *what-if* performance questions [39], but only as specific user-based queries to predict workload behavior in particular settings, not as an automated approach to managing service performance. End-to-end tracing combined with monitored attributes has also been used to predict and explain the latencies of services [27], but their approach is prone to imprecision (i.e., false positives and false negatives) and has been formulated as an optimization problem with a worst case exponential running time as the number of features grows (shown in practice to be about 2 minutes with 100 binary features but for some methods almost 3 hours with 400 features).

Feedback control for resource allocation: Feedback control loops [20] have been used to improve performance in self-managing systems, for actions such as throttling backup services or modifying available thread pools across Apache Servers [16]. For automatic online sizing of resource partitions on a shared server, control loops can help adjust the amount of allocated CPU utilization and the mean response of different services when they are built to recognize various operating regions and static resource saturation scenarios [44, 32]. SCADS [40] focuses on online control of high quantile latency response times using model-predictive control (MPC), reevaluating the current system state after each change in order to recompute the next target state. Related control theory projects use throttling and admission control [23, 24, 10]. Much of this work requires computing transfer functions and linear regression models that fit historical data in services that are either predictable (e.g., serving static web content) or specific (e.g., storage systems). Malkowski et. al [29] looked at empirical multi-model techniques to measure a live stream of CPU utilization and SLO performance data within a control loop. However, they depend on the reproducibility of historical configurations to be able to sustain certain throughput levels. It should be interesting to explore how effective feedback control loops are for our environment of more complex stateful services with dynamic content and a changing set of performance dependencies.

Queuing theory models for guiding allocation: Other tailored approaches to understand performance effects in specific systems make use of analytical queueing-theory based models of web services, which also tend to specialize for particular systems. Slothouber [36] provided one of the earliest open-system analytical performance models for web servers transferring files over the network. This study verified how response time increases suddenly for even small load increases when the server is already near its capacity, and that multiple simultaneous connections make this problem worse. Doyle et. al [17] explicitly model a single-tier web server for CPU, memory, and disk bandwidth. More projects add different queueing models [28].

One project in particular by Urgaonkar et. al [41, 42] studied reactive provisioning for multi-tier architectures using analytical and control theory models. They predict response times for particular resource assignments using queueing-based models and online measurements, and they predict request arrival rates with control

theory by considering past peak demand as well as reacting to inaccurate predictions that differ by more than a threshold level from actual measurements. The queueing-based models require a number of assumptions in the workload distribution (e.g., Poisson), the scheduling mechanism (e.g., processor sharing), and in the constant directional flow of all requests. Moreover, they require model parameters to be either specified or estimated for particular combinations of workloads and services, without providing a general mechanism for obtaining these parameters.

Avoiding SLO violations: The problem of meeting SLOs has also been studied specifically for data-intensive map-reduce style systems, where a delay in the results of a single phase in a series of batch jobs will also delay later phases in the job pipeline that depend on that data. Batch jobs can effectively use more computing resources to grow and effectively spread work across many machines. Jockey [18] monitors and predicts the remaining runtime of jobs, simulates the effect of adding additional resources, and dynamically responds to slower jobs to avoid SLO violations. ARIA [43] is a similar project that uses analytical equations instead of simulation to predict the effects of more resources in the control loop. Mantri [12] also works to achieve more predictable completion times by monitoring the progress of jobs and restarting or duplicating slower tasks with network-aware placement. Scarlett [13] proactively prevents delayed jobs by learning content access patterns and making better replication and placement decisions to avoid contention hotspots.

Other projects use resource usage statistics (without end-to-end traces) in virtual servers to inform migration strategies that can allieviate hotspots and avoid SLO violations. Sandpiper [46] showed the benefits of using OS- and application-level statistics to decide when and where to migrate overloaded VMs. The 1000 Islands project [47] also limited interference from services that oversubscribed the same bottleneck resources by dynamically adjusting resource allocations on each physical server. Both of these cases focus on workload interference rather than downstream request delays. The former project predicts workload requirements based on historical values, but requires threshold tuning of hotspot detection and migration parameters, while the latter assumes perfect knowledge of workload requirements.

References

[1] Private correspondence with a Facebook engineer. 7

[2] Apache Flume. http://flume.apache.org/. 5, 10

[3] Apache Hadoop. http://hadoop.apache.org/. 2

[4] Apache HBase. http://hbase.apache.org/. 2, 5

[5] Apache Thrift. http://thrift.apache.org/. 8

[6] Apache Tomcat. http://tomcat.apache.org/. 8

[7] Linux Kernal-based Virtual Machine (KVM). http://www.linux-kvm.org. 13

[8] Software Engineering Advice from Building Large-Scale Distributed Systems. Stanford CS295 class lecture, Spring, 2007. 2

[9] Twitter Zipkin. https://github.com/twitter/zipkin. 3

[10] Tarek Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13:2002, 2001. 14

[11] Amazon EC2, http://aws.amazon.com/ec2/. 13

[12] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010. 15

[13] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM. 15

[14] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online Modelling and Performance-Aware Systems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, Berkeley, CA, USA, 2003. USENIX Association. 10

[15] James Cipar, Lianghong Xu, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A. Kozuch, and Gregory R. Ganger. JackRabbit: Improved agility in elastic distributed storage. Technical report, Carnegie Mellon University, October 2012. 8

[16] Yixin Diao, Joseph L. Hellerstein, Senior Member, Sujay Parekh, Student Member, Rean Griffith, Gail E. Kaiser, Senior Member, and Dan Phung. A control theory foundation for self-managing computing systems. *IEEE journal*, 23:2213–2222, 2005. 14

[17] Ronald P. Doyle. Model-based resource provisioning in a web service utility. In *In Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS*, 2003. 14

[18] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*. ACM, 2012. 15

[19] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association. 5, 10

[20] Gene F. Franklin, David J. Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001. 14

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association. 7

[22] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems*. 2012. 5, 12

[23] Abhinav Kamra, Vishal Misra, and Erich Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *In International Workshop on Quality of Service (IWQoS*, June 2004. 14

[24] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, November 2005. 14

[25] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104, June 2013. 14

[26] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, James Cipar, Elie Krevat, Michael Stroucken, Julio Lopez, and Gregory R. Ganger. Tashi: Location-aware Cluster Management. In *Workshop on Automated Control for Datacenters and Clouds*, June 2009. 13

[27] Darja Krushevskaja and Mark Sandler. Understanding latency variations of black box services. In *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pages 703–714, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee. 14

[28] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. In *in Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–261. Kluwer Academic Publisher, 2003. 14

[29] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM. 14

[30] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association. 14

[31] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011)*, 2011. 3, 14

[32] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007. ACM. 14

[33] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011. 3

[34] Spec sfs97 (2.0). http://www.spec.org/sfs97. 3

[35] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows , Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report. 3, 14

[36] Louis P. Slothouber. A model of web server performance. In *In Proceedings of the Fifth International World Wide Web Conference*, 1996. 14

[37] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 31–44, New York, NY, USA, 2007. ACM. 7

[38] Eno Thereska and Gregory R. Ganger. Ironmodel: robust performance models in the wild. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '08, pages 253–264, New York, NY, USA, 2008. ACM. 10

[39] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '06/Performance '06, pages 3–14, New York, NY, USA, 2006. ACM. 3, 10, 14

[40] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association. 7, 14

[41] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 291–302, New York, NY, USA, 2005. ACM. 14

[42] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1–1:39, March 2008. 11, 14

[43] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *ICAC'11*, pages 235–244, 2011. 15

[44] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In *Proceedings of the 16th IFIP/IEEE Ambient Networks international conference on Distributed Systems: operations and Management*, DSOM'05, pages 133–144, Berlin, Heidelberg, 2005. Springer-Verlag. 14

[45] John Wilkes. Cluster management at Google, 2011. http://research.google.com/university/relations/facultysummit2011/. 12

[46] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association. 15

[47] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret Mckee, Chris Hyser, Daniel Gmach, Robert Gardner, Tom Christian, and Ludmila Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12:45–57, March 2009. 15