Hotspot Detection in a Service-Oriented Architecture

Pranay Anchuri*
Rensselaer Polytechnic Institute
Troy, New York, USA
anchupa@cs.rpi.edu

Roshan Sumbaly
Coursera Inc
Mountain View, CA, USA
roshan@coursera.org

Sam Shah
LinkedIn Corporation
Mountain View, CA, USA
samshah@linkedin.com

ABSTRACT

Large-scale websites are predominantly built as a service-oriented architecture. Here, services are specialized for a certain task, run on multiple machines, and communicate with each other to serve a user's request. Reducing latency and improving the cost to serve is quite important, but optimizing this service call graph is particularly challenging due to the volume of data and the graph's non-uniform and dynamic nature.

In this paper, we present a framework to detect hotspots in a service-oriented architecture. The framework is general, in that it can handle arbitrary objective functions. We show that finding the optimal set of hotspots for a metric, such as latency, is NP-complete and propose a greedy algorithm by relaxing some constraints. We use a pattern mining algorithm to rank hotspots based on the impact and consistency. Experiments on real world service call graphs from LinkedIn, the largest online professional social network, show that our algorithm consistently outperforms baseline methods.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Modeling Techniques; I.2.6 [Artificial Intelligence]: Learning; D.2.8 [Software Engineering]: Process Metrics

Keywords: call graph; monitoring; service-oriented architecture; hotspots

1. INTRODUCTION

Modern web architectures consist of a collection of services, which are a set of software components spread across multiple machines that respond to requests and map to a specific task. A service is an atomic unit of functionality. This permits easy abstraction and modularity, as well as independent scaling of components.

An incoming user request is load balanced to a front-end service, which fans out requests in parallel to other services to collect and process the data necessary to respond to the incoming request. The callee service of this request can also call other services, creating a call graph (service call graph or SCG) of requests. For example, LinkedIn, one of the largest online social networks, has a recommendation feature called "People You May Know" that attempts to find other members to connect with on the site. To show this module, several services are called: a web server wrapped as a service to receive and parse the member's request, a recommendation service that receives the member id from the web server and retrieves recommendations, and finally a profile service to collect metadata about the recommended members for decorating the web page shown to users.

Modern websites consist of dozens and often hundreds of services to encompass a breadth of functionality. LinkedIn, one of the largest online social networks, runs hundreds of services on thousands of machines in multiple data centers all over the world.

Engineering and operations teams are continually optimizing these services and the call graph to decrease latency, improve throughput, and reduce the cost to serve. However, with such a large and dynamically changing workload, it is difficult and time consuming to determine these hotspots in the call graph. Each page served usually touches dozens of services and machines, and is usually dependent on a particular member's attributes. For example, most content on a page is a function of the member's number of connections in the social graph. In addition, at any given moment, there is continual code deployment and A/B split testing, which further causes call graph changes. As a result, the call graphs vary for the same page.

In a service-oriented world, each service can call several other services before returning to its caller with the appropriate response. On one extreme, these subcalls can be made sequentially meaning that a service is called only after the previous one completed or is called in parallel, meaning that the services are called at the same instant or in a brief time span. Further the subsequent calls can be called either serially or in parallel. The service itself can also spend time performing internal computations. Understanding where a service spends most of its time is important in detecting hotspots to optimize the call graph. The latencies and the order of service calls vary widely across different requests, making it difficult to create a reasonable model to fit historical data.

In this work, we present a novel unsupervised algorithm that determines hotspots in a service-oriented architecture. It combines historical and latest service metrics data to rank hotspots in the call graph. This ranked list provides a simple vehicle to target which services to optimize. The algorithm works in two stages. In the first stage, the hotspot services are computed for each call graph by analyzing the impact each downstream service has on the request. In the second stage, the hotspots across call graphs are aggregated to find the best hotspots.

The contribution of this work is a new approach for detecting hotspots in a service-oriented architecture with respect to various objective functions. In particular, we show how it can be used to find services that will improve the latency of a functionality and

*Work performed while the author was an intern at LinkedIn Corp.

CIKM '14 November 03 - 07 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2661991 .

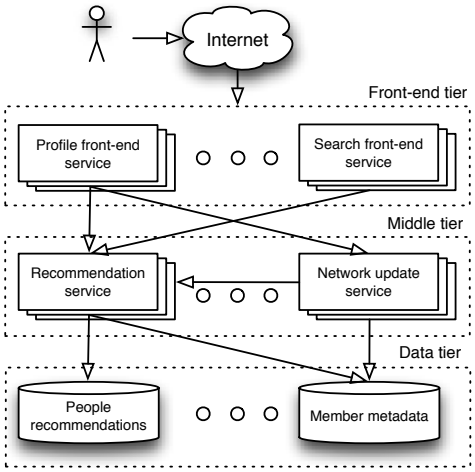


Figure 1. A generic website architecture consisting of multiple tiers of services

reduce the cost of serving a request. The algorithm can be used to analyze both the historical call graph data and real-time call graphs.

We evaluate our approach on millions of call graphs generated at LinkedIn. Our results show that our method consistently outperforms other baseline methods.

The rest of the paper is organized as follows. Section 2 provides background on service-oriented architectures and Section 3 showcases related work. Section 4 introduces notation used in this paper. We first show how to optimize latency in Section 6, followed by the cost to serve in Section 7, and then any arbitrary metric in Section 8. We evaluate our algorithm against current techniques in Section 9 and finally conclude in Section 10.

2. BACKGROUND

Websites are composed of multiple services that can be classified into 3 tiers: the front-end, middle, and data tier, with each service in their tiers exposing APIs for communication. Figure 1 presents the high-level architecture of LinkedIn. The *front-end tier*, consists of services catered to serving user requests by multiplexing them to the correct service in the *application tier*. There is generally a one-to-one mapping between user-facing functionality and a front-end service. For example, the search front-end service would only cater to requests coming from search pages. The *application tier* services communicate amongst themselves, and then finally call the *data tier* to retrieve data from databases.

Each service implements a common interface that generates common metrics such as latency, error count, etc. and sends them to a centralized metric collection system. Because these systems are running on various machines, every user request produces a random trace id that is injected at the front-end service, and then passed along by downstream services for the metric collection system to construct a call graph.

LinkedIn runs hundreds of services deployed on thousands of machines in multiple data centers. At any given time there are multiple versions of the site being shown to users with frequent code changes and deployments. As new features are added regularly, new services get added to all 3 tiers of the architecture, making the call graph complicated and error prone. Because services communicate with each other via pre-defined APIs, no single engineer has the complete domain knowledge of all APIs and their corresponding dependencies.

Hence, in this complicated call-graph it becomes very important to automatically detect hotspots with minimal human effort. The algorithm for detecting these hotspots should be domain agnostic and should cater to complicated call graphs that could potentially be affected by external factors such as call timeouts, hardware maintenance, or service hardware co-location.

3. RELATED WORK

Most of the existing work on diagnosing large-scale service-oriented architectures are aimed at modeling workflows [7, 17, 18, 21, 22] and latency [16, 19]. Mann et al. [16] compared the performance of several models in estimating service latency as a function of the calling RPCs latencies. Ostrowski et al. [19] developed a probabilistic model for end-to-end modeling of the *root* service's latency. This modeling enables the user to ask "what if" questions to understand the impact of changing a downstream service on the *root* service. Though these models work well in estimating impact, they cannot suggest the actual services to optimize. Further, most of these models are tailored for optimizing just one metric: latency; the generalized framework we present in this paper recommends the hotspot services with respect to various metrics.

Estimating link bandwidth bottlenecks in a network is a well-studied problem in the field of network monitoring: tools that detect the bottlenecks are widely used by network operators for effective utilization of network resources. Hu et al. [13] developed "Pathneck", a tool to detect bottlenecks along network paths by estimating the bandwidth of the links that make up that path. It uses a probing technique where the source sends a sequence of UDP packets containing load and measurement packets, called recursive packet train (RPT), along the paths that the user wants to analyze. At each node on the path, two ICMP packets are sent back to the source; bandwidth is estimated using the time gap between receipt of the two packets. Choke points along a path are defined as the links where the bandwidth is less than what is available on the path from the source to that link, and where the bottleneck node is the last choke point along the path. Hu et al. [14] also analyzed various properties of the bottlenecks like persistence and packet loss.

The tool developed by Harfoush et al. [12] analyzes how the packet train messages are handled and can detect bottlenecks in arbitrary subpaths. Other tools for detecting bottlenecks include "BFind" developed by Akella et al. [1] and the packet tailgating method by Ribeiro et al. [20]. Recently, there have been algorithms [10, 24] developed for bottleneck detection in sensor networks that face new challenges compared to traditional networks.

Betweenness centrality, primarily used in the context of social and transportation networks is a measure of the importance of a node. A node or edge with high betweenness value is part of many shortest paths between nodes in the network [4, 8]. These nodes and edges can be interpreted as bottlenecks in the network. Cuzzocrea et al. [5] uses the betweenness idea for controlling the topology in wireless sensor networks: given a set of nodes and quality of services (QoS) requirements, their algorithm suggests a topology that satisfies the requirements with minimum energy utilization.

For several reasons, detecting hotspots in SCG poses several unique challenges that make it infeasible to use any of the network monitoring approaches previously described.

Controlling metric. Most of the bottleneck detection algorithms construct packets to analyze the bandwidth. They have fine grained control over the length of the packet and the nodes involved. However, this is not possible in the case of SCG as metrics such as latency cannot be changed arbitrarily while keeping other factors constant.

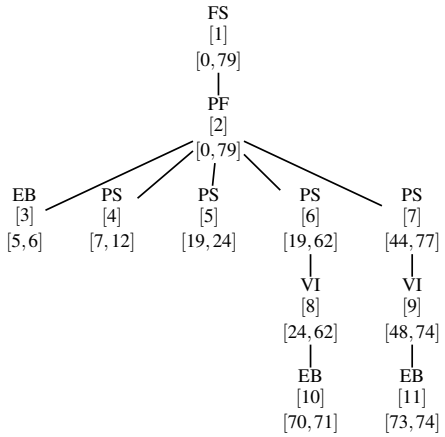


Figure 2. Example of an SCG. For each node, [id], name of the services (anonymized), and the [start, end] time of the service call that initiated the service is shown.

Network structure. Network monitoring tools usually analyze a fixed large network. In our case, the network is a SCG that is relatively small and has a variable structure for the same system functionality.

Parallel service calls. Parallel service calls introduce a new challenge that the traditional algorithms don't handle.

Goal of detecting hotspots. Our ultimate goal is to find the services, which if optimized, will impact the functionality the most. This goal doesn't always translate to finding the services that form the bottleneck. For example, there may not be any benefit in optimizing a service that has the maximum response time especially if its caller is waiting for a different service to be completed.

This work takes all these challenges into consideration to detect hotspots in service oriented architectures.

4. PRELIMINARIES

The LinkedIn website is composed of many user-facing functionalities. Examples of these include search, a news feed, a profile, and so on. Whenever the user requests a functionality F, multiple services call each other to serve the incoming request. The first service initiating the request is part of the front-end tier. The service that initiates a call is the caller, and the service it calls is the callee. The dependencies between the caller and the callee services in a given request can be modeled as a directed acyclic graph with a node for each service. We call it a Service Call Graph or SCG. We also refer to each request to F as an instance of F. The set of services called for serving F can vary across different instances because of various factors including caching, errors and so on. The hotspot detection algorithm has to account for the fact that SCG for a functionality doesn't have a unique representation. For each service call in an SCG various useful metrics are collected. These metrics help us find hotspot services with respect to different objectives.

For an instance of functionality F, the SCG G is denoted by a 3-tuple $G = (V, E, M)$, $V \in \mathbb{N}$ is the set of unique service identifiers, $E \subseteq V \times V$ is the set of service-service calls, and $M : E \to \mathbb{R}$ computes the metric associated with each service call. For every $(u,v) \in E$, v is a *subcall* of u, and u is the parent of v, denoted by $p(v) = u$. Since each node has a unique parent, the metric on edge (u,v) is considered a property of v.

In every SCG, the node without a parent is denoted by *root* and it usually corresponds to a service in the *front-end tier*. The degree of a service u, $d(u)$, is the total number of subcalls from u. Also, the trace of a service, $tr(u)$, is the sequence nodes on the path from *root* to u *i.e.,* $tr(u) =< u_0 = root, \ldots, u_h = u >$ such that $p(u_{i+1}) = u_i$.

Figure 2 shows an example SCG with the *latency* metric. For each node, the unique identifier, service name, and the start and end times of the service-service call is shown. The latency of an edge is the difference between the start and end times of the associated service call. In the example, $M((2,4)) = 12 - 7$, $p(4) = 2$, and $d(2) = 5$. Notice that the nodes $4, 5, 6$, and 7 are different instances of the same service.

Abstract Problem Statement: Given SCG instances $G_i = (V_i, E_i, M)$, of a functionality F over a time period and a small positive integer α. Find α hotspot services for F with respect to the metric M.

5. OUR APPROACH

In this section, we describe the various steps in our framework. For each step, we explain the rationale behind using a specific approach. The following shows three major steps in our hotspot detection framework.

Steps in hotspot detection framework
1. If necessary, compute a derived metric for each service call.
2. Find top-k hotspot services in every SCG instance.
3. Return frequently occurring hotspot services as α services to optimize.

5.1 Top-k hotspots in each SCG instance

In a broad sense, there are two ways to detect α hotspot services in service-oriented architecture. First, we find/construct a few summary G_i's representing the general trend of the service call metric across all SCG instances. The summary G_i's are then analyzed for computing α hotspot services. Second, we compute k (for a small positive integer k) services that form a hotspot in a given instance of G_i. We call this the top-k list for the SCG G_i. The top-k lists are further analyzed to compute α hotspot services.

We follow the second approach as it presents the following three advantages compared to the other approach.

Summarizing SCG. A problem similar to that of computing summary SCG occurs in the field of Bioinformatics in the form of constructing a consensus tree from a set of phylogenetic trees. A phylogenetic tree gives the evolutionary relationship between a set of species. By using different construction algorithms, different trees are obtained for the same set of species, and the *consensus* tree is a single tree that includes features from all the trees.

In our case, SCG plays the role of a phylogenetic tree and the services play the role of species. The summary tree which we are interested in is a consensus tree of the input SCG instances. However, there are two major differences between a phylogenetic tree and an SCG. First, in a phylogenetic tree, the species lie only at the leaf level, whereas in an SCG even the internal nodes represent a service. Second, edges in a phylogenetic tree usually don't carry any weights. In SCG, edges are always associated with a service call metric.

Additionally, most of the consensus tree construction methods start by computing the bipartitions of the phylogenetic trees [2, 23]. The consensus tree is then constructed from the bipartitions that are present in all the input trees. However, the number of bipartitions is significantly more in the case of SCG, because the bipartitions are subgraphs rather than a set. Also, comparing bipartitions requires solving graph isomorphism, a computationally intensive problem.

SCG	v_1	v_2
G_1	x_1	x_2
G_2	x_1	x_2
G_3	x_3	x_2
G_4	x_3	x_4
G_5	x_3	x_5

Table 1. Example showing top-k lists for various service call graph instances. The columns v_1 and v_2 denote the service names of top 2 services obtained by solving Equation 1 corresponding to SCGs $G_1, \ldots, G_5$.

Hence, constructing the summary SCG is both memory and computation intensive, especially, when the number of requests are in the order of millions per day.

Loss of information. Even if we had enough computation and memory resources to compute and compare the bipartitions, another challenge is to merge the bipartitions that have different metric values on the edges. There is an inherent loss of information if any summary statistics such as mean, median and so on of the metric values are used.

Online and offline algorithm. Our approach has an added benefit that it can be easily made into an online algorithm because hotspots are computed on a per instance basis. In the alternate approach, the summary SCG construction step cannot proceed until all the SCGs are collected.

Objective function. In summary, the goal of computing top-k hotspots is equivalent to solving the following objective function for each SCG instance.

$$\begin{aligned} &\max && f(S) \\ &\text{subject to} && |S| \leq \text{k},\ S \subseteq V \end{aligned} \tag{1}$$

The function f is constructed to reflect the changes in the value of the metric under consideration as the services in S are optimized. The set V consists of all the services in the specific SCG instance.

5.2 Frequent top-k hotspot services

We motivate the need for this step with an example. Table 1 shows top-k (k=2) hotspots in five SCG instances of a functionality F. Based on the frequency of occurrence in top-k lists, the services $\{x_2, x_3\}$ are the best candidates to optimize F. However, optimizing these services together might not have a huge impact on requests to F in general, because they co-occur only once in the five top-k lists. A better choice would be to optimize $\{x_1, x_2\}$, which occurs twice. Frequent itemset mining solves this exact problem. Therefore, the target services to optimize are the frequent sets of services in the top-k hotspot lists computed for SCG instances.

6. OPTIMIZING LATENCY

The objective of analyzing the SCGs of a functionality F with respect to the latency metric is to find α services, which if optimized, produce the greatest reduction in response time of requests to F. These services are the hotspot services for F. Following our three step approach, the first step is trivial. With the second step, we compute the top-k hotspots in each SCG.

We first define an appropriate objective function f (Equation 1) and then devise an algorithm to solve it efficiently. A scalable and efficient algorithm is required because the number of SCG instances are in the order of millions.

Key Idea: Assume each service in a SCG_i can be run $\theta(>1)$ times faster. We answer the following question: *What are the k services, if already optimized, that would have led to maximum reduction in response time of root in SCG_i ?* We are not interested in their effect on other SCG instances.

6.1 Effect of optimizing one service

First, we model the effects of optimizing a service.

Definitions: For any service v, the *start time* and *end time* is denoted by s_v and e_v respectively, $(e_v > s_v)$. The interval $[s_v, e_v]$ is its *active interval*. The *response time* equals the *length* of active interval, defined by the difference $e_v - s_v$. At any given instance during the active interval $[s_v, e_v]$, the service v is in one of the two states: either waiting for a subcall to return or performing internal computations.

An interval $[t_1, t_2]$ is a *waiting interval* of v if and only if the active interval of any its subcall is either completely disjoint or entirely contained within $[t_1, t_2]$. In other words, it is one of the maximal intervals during which v is waiting on a subcall. On the other hand, a *computing interval* is a maximal subinterval such that no subcall of v is active at any instant. These definitions imply that the computing and waiting (from now on referred to as CW) intervals are both disjoint and also that their union is the active interval.

Consider the SCG in the Figure 2. The start time of the service 2 is $s_2 = 0$ and its end time is $e_2 = 79$. The response time of the service 4, $r(4) = 12 - 7 = 5$. The CW intervals of 2 are {[0, 5],[6, 7],[12, 19],[77, 79]} and {[5, 6],[7, 12],[19, 77]}, respectively. Note that a computing interval is always followed by a (possibly empty) waiting interval.

6.1.1 Local effect of optimization

To begin with, we compute the effect of optimizing v on its own response time. Methods such as those proposed in Mann et al. [16] estimate this effect by precomputing a model based on the trace history [22]. But the question that we are trying to address here is *what-if* the service was optimized in a specific SCG instance, irrespective of the metrics in other SCGs. It is important to understand this distinction. We estimate the impact by dividing its active interval into blocks of CW interval pairs.

Let v be a service and $v_1, v_2, \ldots, v_{d(v)}$ be its subcalls. Without loss of generality, assume that the number of CW pairs is m and that the lengths of CW intervals (sorted based on start time) in the i^{th} pair are c_i and w_i, respectively. Based on the definition of CW intervals, the response time of v is written as :

$$r(v) = \sum_{i=1}^{i=m} (c_i + w_i) \tag{2}$$

Figure 3 shows the active intervals of v and its 6 subcalls, v_1 through v_6. The CW interval pairs of v are $([0,1],[1,8])$, $([8,9],[9,10])$, $([10,12],[12,15])$ and $([15,16],[16,16])$. In general, given the start and end times of the subcalls of a service, the CW pairs can be computed incrementally using a trivial linear time algorithm starting with the subcall that starts the earliest.

To quantify the local effect, assume that v is optimized by a factor $\theta(\theta > 1)$. We call θ the improvement factor. This optimization leads to smaller computation intervals because, theoretically, all the internal operations are performed θ times faster. In other words, an internal operation that takes a unit time before the optimization, now runs in θ^{-1} units of time. Therefore, the computation time of v is reduced by $(1-\theta^{-1}) \times \sum_{i=1}^{i=m} c_i$ and the active interval of a subcall v_j in the i^{th} CW pair is shifted by $(1-\theta^{-1}) \times \sum_{k=1}^{k=i} c_k$. The modi-

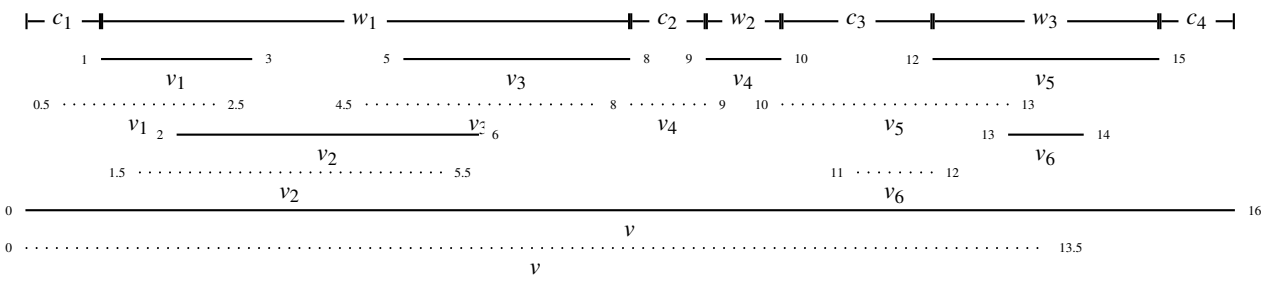


Figure 3. Active intervals before and after optimizing v by $\theta = 2$

fied start and end times (s', e') of v and its subcalls are as follows:

$$\begin{aligned} e'_v &= e_v - (1-\theta^{-1}) \times \sum_{i=1}^{i=m} c_i \\ s'_{v_j} &= s_{v_j} - (1-\theta^{-1}) \times \sum_{k=1}^{k=i} c_k \\ e'_{v_j} &= e_{v_j} - (1-\theta^{-1}) \times \sum_{k=1}^{k=i} c_k \end{aligned} \tag{3}$$

In Figure 3, the dotted lines shows the active intervals after optimizing the service v by $\theta = 2$.

6.1.2 Global effect of optimization

Optimizing a service v reduces its response time, $r(v)$. This effect propagates to its parent, and recursively all the way to the *root* along the trace, $tr(v)$. It is also possible that the active intervals of other services are shifted. We call this the global effect of optimization. However, sometimes the impact is not propagated to the parent. For instance, in Figure 3, optimizing the service v_6 has no impact on the response time of its parent, v, because the service v_5 is one of the bottlenecks for v. Now, we formalize the notion of global effect using the following assumption.

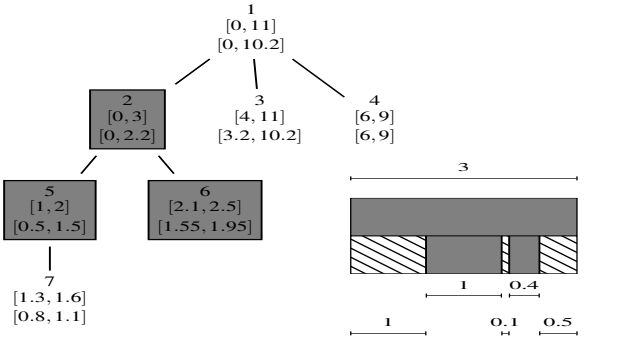
ASSUMPTION 1. *Let v be a service and $v_1, v_2, \ldots, v_{d(v)}$ be its subcalls. Let e_{v_i} and e'_{v_i} be the end times of the subcall v_i before and after optimization. The effect propagates to v (statement p), if and only if $\nexists v_j$ such that active interval (after optimization) of v_i overlaps with that of v_j (before optimization) (statement q) i.e., $p \Leftrightarrow q$.*

REASONING 1. $\mathbf{p} \rightarrow \mathbf{q}$. We will reason about the contrapositive *i.e.,* $\neg q \rightarrow \neg p$. Suppose, such a service v_j exists. As the active intervals overlap, one of the following two conditions hold true.

1. $e'_{v_i} < e_{v_j} < e_{v_i}$ or $e'_{v_i} < s_{v_j} < e_{v_i}$. In either case, if e_{v_i} is reduced to e'_{v_i}, the relative order of the subcalls v_i and v_j changes from v's perspective *i.e.,* v_j ends later than v_i. Because, our method is agnostic to how the service-service calls are made, we don't allow effects to be propagated if it changes the relationship between subcalls. So, the effect is not propagated to the parent, v.
2. $[e'_{v_i}, e_{v_i}] \in [s_{v_j}, e_{v_j}]$. In this case, the effect is not propagated because the subcall v_j forms a bottleneck.

$\mathbf{q} \rightarrow \mathbf{p}$. In this case, shifting the active intervals of all subcalls by $e_{v_i} - e'_{v_i}$ retains the order of other subcalls. This is because the active interval of v_i doesn't overlap with that of any v_j. Hence, the effect can be propagated to v if no subcall except v_i is active during the interval $[e'_{v_i}, e_{v_i}]$. □

In summary, Assumption 1 restricts the impact propagation to the cases where the order of the subcalls is maintained for all the services. It is required because we want the algorithm to be agnostic to the underlying dependencies between the subcalls and their parent service.



(a). Example SCG **(b).** Timeline of a service 2

Figure 4. Example showing global effect of optimization. The service 2 is optimized by $\theta = 2$.

6.1.3 Active intervals after optimization

We now combine the local effects (Section 6.1.1) and global effects (Section 6.1.2) to compute the effect of optimizing a service on the *root* service. We define the impact on *root* as follows.

$$impact(v) = \begin{cases} (1-\theta^{-1}) \times \sum_{i=1}^{i=m} c_i & \text{if Assump. 1 true} \\ & \forall u \in tr(v) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

m is the number of CW pairs and θ is the optimization factor. It can be seen from Equation 4 that the impact is propagated only if Assumption 1 holds true for all the services on the path from v to *root* in the SCG. The active intervals of the services are updated as follows: If the service is a descendant of v_j, and v_j belongs to the i^{th} CW interval of v, then its active interval is shifted by $(1-\theta^{-1}) \times \sum_{i=1}^{i=j} c_i$. Otherwise, the active intervals of all services that start later than s_v are shifted by $impact(v)$.

Figure 4 shows an example of a SCG instance. For each service, the active intervals before and after optimizing the service 2 by $\theta = 2$ are shown. The timeline and CW intervals of 2 are shown in the Figure 4a. Notice that the computation time (dashed boxes) of the service 2 is 1.6. Therefore, the response time of the *root* is reduced by 0.8. But, the response times of 5 and 7 are reduced by 0.5 because they descend from a subcall in the first CW interval pair of 2.

6.2 Top-k services in one call graph

In this section, we derive an objective function for the latency metric, and present an algorithm to compute the optimal subset

of services. We assume each service can be optimized by θ, and compute the top-k hotspots in a SCG. The algorithm can be easily extended to cases where the optimization factor θ differs for different services.

LEMMA 1. *Let x and y be two services that have a non-zero impact on the root. The total impact after optimizing x and y is the same irrespective of the order in which the services are optimized.*

Proof Sketch:

The interesting case is when x is a descendant of y in the SCG or vice versa. Without loss of generality, let y be a descendant of x. Then $[s_y, e_y] \in [s_x, e_x]$. Irrespective of the order in which the services are optimized, the end times of the services x and y are $(e_x - impact(x) - impact(y))$ and $(e_y - impact(x) - impact(y))$, respectively. The lemma is easy to prove in other cases. □

We can also see that the effects are additive. For a subset S of services, the total impact of optimizing all the services in S is just the sum of individual impacts (computed as in Equation 4). Therefore, the objective function $f(S)$ can be defined as:

$$f(S) = \sum_{v \in S} impact(v) \tag{5}$$

THEOREM 1. *Computing S, $|S| = k$, that maximizes $f(S)$ is NP-Hard.*

Proof Sketch: We prove this using a reduction from the following variant of the subset sum problem, which is an NP-hard problem [15].

Subset sum: Let $x_1, \ldots, x_n$ be positive integers, M be the maximum bound, and k be the maximum size of the subset. Find a subset S, $|S| \leq k$, that maximizes $\sum_{x_i \in S} x_i, \sum_{x_i \in S} x_i \leq M$. The problem is equivalent to computing top-k hotspots for the SCG shown in the Figure 5.

The variable t depends on θ as : $t = \theta \times (\theta - 1)^{-1}$. It is chosen such that a computation that takes t units before optimization will be a unit faster after the optimization. ε is a small positive constant. We make the following observations from the SCG construction.

- A and B have 0 impact because of Assumption 1.
- Each of the i nodes, $1 \leq i \leq n$, has a computation time of $x_i \times t$, and optimizing it by θ reduces the response time of *root* by x_i. Therefore, optimizing node i has the same effect as picking x_i in the subset.
- i' nodes make sure that each i node is picked at most once.

Moreover, the maximum possible reduction in the response time of *root*, without violating Assumption 1, is M. Therefore, if there exists a polynomial time algortihm for maximizing the objective function f then it can be used to solve the subset problem in polynomial time. Now, the equivalence between the two problems can be established easily. □

We now relax Assumption 1 so that computing optimal S becomes tractable.

ASSUMPTION 2. *Let v be a service and $v_1, v_2, \ldots, v_{d(v)}$ be its subcalls. Let e_{v_i} and e'_{v_i} be the end times of v_i. Further, $I^* = [e^*_{v_i}, e_{v_i}]$ be the maximum subinterval of $[e'_{v_i}, e_{v_i}]$ such that $\nexists v_j$, v_j is active during I^*. By optimizing v_i, the response time of v is reduced by $e_{v_i} - e^*_{v_i}$.*

REASONING 2. The assumption can be thought of as a fractional version of the Assumption 1. Instead of restricting the impact on v to be either 0 or $e_{v_i} - e'_{v_i}$, this allows for a partial effect $e_{v_i} - e^*_{v_i}$ and has the property that the relative order of the services remains intact. It can be proved using the same arguments as in Assumption 1. □

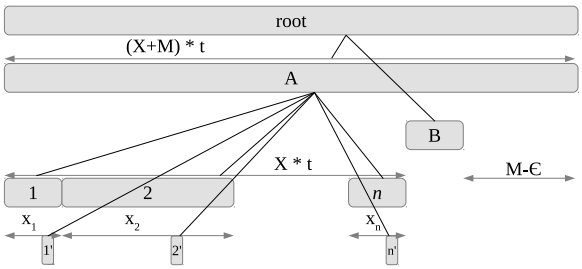


Figure 5. SCG for which computing the optimal k subset is NP-Hard

Computing optimal subset S. With Assumption 2, the subset S that maximizes Equation 5 can be computed using a greedy iterative approach. In each iteration, pick the service that has the maximum impact on the *root* service and modify the active intervals as described in Section 6.1.3. The services picked in this process are the top-k hotspots. This procedure returns the optimal set because Assumption 2 reduces the subset problem to its fractional version, which has an optimal greedy algorithm.

6.3 Frequent subsets of services

The last step in our hotspot detection framework is computing frequent service(s) from the top-k hotspot lists of each SCG.

We used a frequent pattern mining approach to find α services to optimize to produce the greatest reduction on the latency of F. Frequent itemset mining is one of the fundamental data mining tasks and has been widely studied in the literature. Its main goal is to find an item or set of items that occur frequently in a list of transactions. We direct the reader to Goethals [9] and Han et al. [11] for an overview of frequent pattern mining methods.

Let I be a set of items. A transaction is just a subset of I. Given a list of n transactions $T_1, T_2, \ldots, T_n$, the support of a set S is the fraction of transactions that are a super set of S *i.e.,* $support(S) = \frac{|\{T_i | S \subseteq T_i\}|}{n}$. We say S is *frequent* $\iff support(S) \geq minsup$ for some user given threshold *minsup*. S is also called a maximal set if $\nexists S'$ such that S' is frequent and $S' \supset S$. The algorithm by Borgelt [3] returns all maximal sets from a given database of transactions and *minsup*. In our case, each top-k list becomes a transaction and I is the set of all services. The maximal sets of services are the hotspots for F.

6.3.1 Ranking

We present three different schemes to rank maximal sets F.

Frequency Ranking. Maximal sets are ranked by their support.

Impact Ranking. Frequency based ranking ignores the impact of the services in the maximal set. The average impact, $\widetilde{f}(S)$, of a maximal set S can be defined as follows:

$$\widetilde{f}(S) = \sum \frac{f(S, G_i)}{support(S)}, S \subseteq T_i$$

where the $impact(S, G_i)$ in G_i is computed using the Equation 5, and T_i is the top-k list for G_i. In this ranking scheme, the maximal sets are ranked by their average impact.

Coverage based Ranking. We say a maximal set covers all SCG instances in which it is a subset of the top-k list. In this ranking scheme, we try to maximize the number of top-k lists covered using a minimum number of maximal sets. This is similar to solving set cover problem [15] where each top-k list is an element that has to be covered. Computing the minimum set cover is an NP-complete problem and has no polynomial time algorithm unless P = NP. How-

ever, there is a greedy $O(\log n)$ approximation algorithm [6]. In our experiments, we used impact ranking.

7. OPTIMIZING COST TO SERVE

At LinkedIn, and many other web properties of sufficient scale, data is both stored and processed in a distributed fashion. Each service can call the same services on multiple machines. Informally, the number of such calls is called its multiplicity. For example, a service handling a user search request may scatter service calls to several machines to gather and then merge the relevant data. Without loss of generality, we assume that the number of machines used by a service is a proxy for the cost that it contributes to a request. Therefore, the hotspot services with respect to *cost to serve* metric are the services that use maximum number of machines. Though it seems like a straightforward counting problem, the main challenge lies in the construction of SCG with an appropriate metric for each service-service call.

We briefly describe how our framework finds services that reduce the cost of serving a request.

7.1 Computing multiplicity metric

To construct the SCG with the multiplicity metric, we first need a way to group equivalent subcalls made by a service.

DEFINITION 1. *Let u be a service, and v_1, v_2 be any two subcalls. The services v_1, v_2 are equivalent, denoted by $v_1 \equiv v_2 \iff v_1 = v_2$ and $\exists$ a bijective function $f : X = subcalls(v_1) \longrightarrow Y = subcalls(v_2)$, $\forall x \in X, x \equiv f(x)$.*

In other words, a pair of services are equivalent if and only if all the downstream paths originating from them are isomorphic. For example, in the SCG shown in Figure 6 the *root* node service A calls two instances of service B with node ids $2,4$. Each of these services calls other services identically. Therefore, the nodes are equivalent *i.e.,* $B_2 \equiv B_4$. We can group the subcalls of a service into a set of equivalence classes such that all pairs of services in a class are equivalent. Let E be an equivalent class of subcalls from a service u. Then, the $|E|$ equivalent services can be replaced by a service v with multiplicity metric as $M((u,v)) = |E|$. As in the case of latency, we consider it a property of the subcall service v.

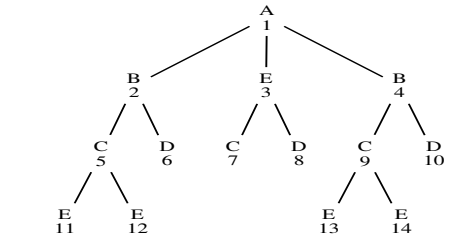
The metric $M(v)$ gives us the multiplicity v for each instance of its parent $p(v) = u$. Similarly, $M(u)$ is the multiplicity in terms of its parent $p(u)$. Therefore, the cost contributed by a service v, $impact(v)$, is:

$$mul(v) = \prod_{u \in tr(v)} M(u) \tag{6}$$

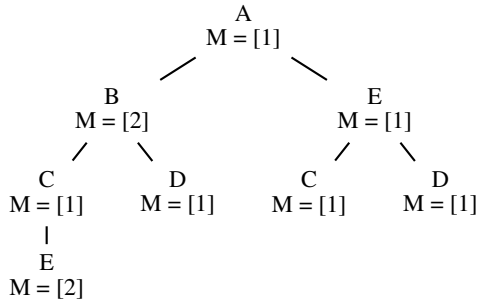
To compute the multiplicity metric, the nodes are processed in level order, and equivalence classes of subcalls are computed. All the services in an equivalent class are replaced by a single service with the metric value equal to the size of the equivalent class. The multiplicity of *root*, and services x with $d(x) = 0$ being 1. Figures 6a and 6b show an example of an SCG without metric and with the metric M. In the first level, the equivalence classes of subcalls $\{2,3,4\}$ are $\{2,3\}$ and $\{4\}$. Therefore, subtree rooted at 4 is deleted and $M(2) = 2$. Similarly, in the third iteration the node 12 is deleted and $M(11) = 2$. The number of machines used by the service E with $p(E) = C$ equals the product $1 \times 2 \times 1 \times 2 = 4$.

7.2 Detecting hotspots

Given that we have the SCG with the multiplicity metric, we can use our hotspot detection framework to find services that use the most machines for a given functionality F in the system. Computing the top-k list is trivial, as it is just the first k services sorted in the decreasing order $mul(v)$ computed using the Equation 6. From the top-k lists of each SCG, we compute and rank the α services using the frequent itemset mining approach as described in Section 6.3.



(a). SCG without metric



(b). SCG with multiplicity metric

Figure 6. Example showing the construction of service call graph with number of machines as the metric.

8. OPTIMIZING ARBITRARY METRICS

To summarize our discussion, we now show how our framework can be extended to detect hotspots with arbitrary metrics and objective functions. The first step is to log or compute the appropriate metric for each node in the SCG. The next step is to devise an objective function f that reflects the goal, and solve the optimization problem in Equation 1. In the case of latency $f(S) = \sum_{v \in S} impact(v)$, and for multiplicity $f(S) = \sum_{v \in S} \prod_{u \in tr(v)} M(v)$. Additional constraints such as Assumptions 1 or 2 may be enforced on the feasible solutions. The optimal set S is the top-k list in a given SCG instance. Once the top-k lists are computed for all SCG instances, any frequent mining algorithm can be used to mine the α hotspot services. Additionally, various ranking schemes as described in Section 6.3.1 can be used to rank these hotspots.

9. EXPERIMENTS

We performed several experiments to show the effectiveness of our hotspot detection framework. We show the results for latency because it is one of the most important metrics to optimize.

A practical problem arises in testing whether the projected improvements in the metrics are correct because the projections are based on the assumption that the underlying services can be optimized. Therefore, we used indirect methods to show that the hotspot services predicted are true hotspot services in the LinkedIn website. Due to the lack of space, we focus our experiments on two of the most requested functionalities on LinkedIn denoted by F_A and F_B.

The hotspot detection system consists of three components.

Batch Processing. The algorithm is run periodically to construct SCG from the service calls, mine top-k services in each SCG, and finally compute the α services for all the functionalities in the sys-

Figure 7. Example data from web application UI.

Functionality	|F|	$\|E\|$	$d(v)$	Max $d(u)$
Home	10.2 M	16.90	1.88	9.02
Mailbox	3.33 M	23.31	1.9	8.88
Profile	3.14 M	17.31	1.86	11.04
Feed	1.75 M	16.29	1.87	8.97

Table 2. Number of incoming requests (in millions), number of service calls, average degree of non-leaf services, and maximum degree of a non-leaf service for various functionalities. The numbers are scaled by a constant and averaged over a day.

tem. The computed results are stored in a distributed key-value store keyed by the functionality F and date.

Online. We also developed an online version of the system to detect hotspots in real time. In this, the stream processing engine computes the top-k in each SCG after all the service calls return to their caller.

Web Application. as a web application. The tool is useful for all the developers to target the services to optimize. A user of the system has the option of choosing the functionality F and the time frame to analyze. The application retrieves a ranked list of hotspots for F along with potential improvement if the services were optimized.

Figure 7 is an example of how the results are presented to the user. The table shows the possible reduction in the response time of *root* by optimizing hotspots with different improvement factors (in percentages). The services are shown in the order of their impact ranking as defined in the Section 6.3.1.

The preprocessing scripts and the SCG construction steps run on a Hadoop cluster. We have developed a Java-based library for processing SCG with respect to a user-preferred metric. Mining of frequent services from the top-k lists is relatively easy in terms of the computational complexity and is performed on a single node. The value of minimum support, *minsup*, we used is equal to 25% of the number of SCG instances.

9.1 Statistics

Table 2 shows the the number of incoming requests we are mining. All the numbers shown are scaled down by a constant factor (>1). It shows the average number of requests, number of service calls, average degree of non-leaf services that is services v such that $d(v) > 0$, and the maximum degree of non-leaf services for some of the most commonly requested system functionalities. One of the most requested functionalities *Home* is requested on an average of 10.2 million times per day. There are about 17 inter-service calls in a request to *Home*.

Fraction of unique instances	Number of services
(0.88, 1.0]	40
(0.76, 0.88]	6
(0.64, 0.76]	2
(0.53, 0.64]	1
(0, 0.53]	1

Table 3. Fraction of unique SCG instances for 50 most requested functionalities in the system.

API	Maximum parallel calls
Get connections	17
Edit profile	12
Updates	12

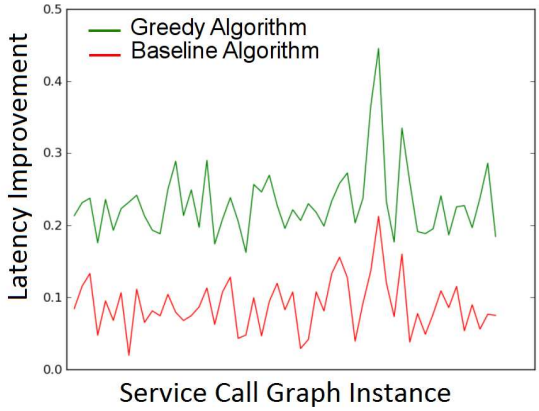
Table 4. Maximum number of active parallel calls during typical API requests.

One of the main challenges in mining SCGs is handling the changes in the structure of the call graphs called for serving the same functionality. This is not just an exception, but the norm in typical service-oriented architectures. We computed the fraction of unique SCGs for 50 most requested services . Table 3 shows the number of functionalities in various intervals. For example, in 40 out of the 50 functionalities, at least 88% of the SCG instances are structurally unique. Note that the uniqueness is computed only based on structure—response times are not considered. Inode t can be seen most the SCG instances are unique, reinforcing the complexity of the mining hotspots problem.

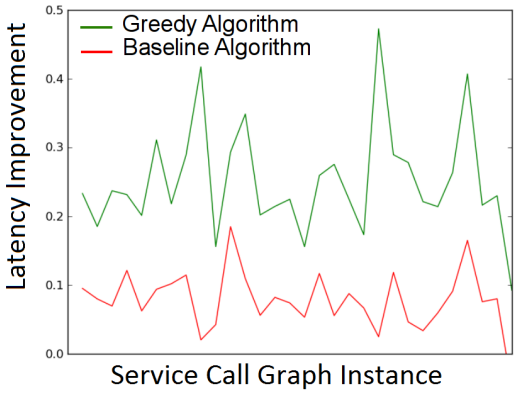
As shown in Section 6, overlapping subcalls pose a serious challenge in computing the top-k hotspots. It is especially observed for the latency metric. Table 4 shows the maximum number of overlapping subcalls for some downstream service in various functionalities.

9.2 Comparison against baseline approach

We show the effectiveness of our greedy top-k service computation to a baseline approach. One way of selecting top-k services in a SCG is to select the first k services sorted based on the metric under consideration. In other words, the top-k services are the k services that have the maximum response time or use the maximum number of machines when the goal is reduce to latency or number of machines used, respectively. Figure 8 shows the fractional improvement in the response time of F_A and F_B instances by optimizing the top-k services returned by our greedy approach and the baseline algorithm. The improvement factor $\theta = 2$ and $k = 3$. The maximum improvement possible is $(1 - \theta^{-1}) = 0.5$. It can be

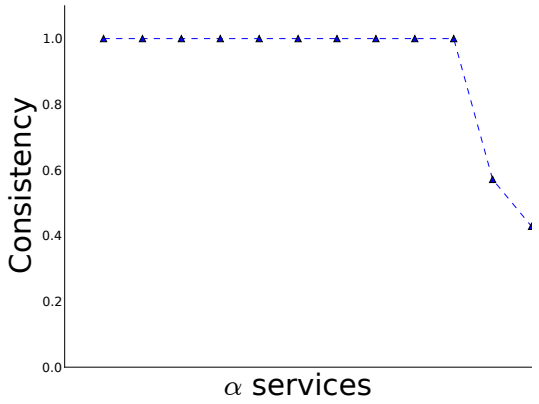


(a). Improvement in response time of F_A

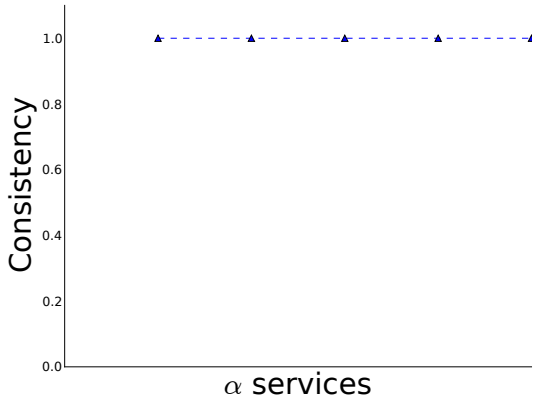


(b). Improvement in response time of F_B

Figure 8. The improvement in latency if the top-k services are computed using our greedy approach and baseline algorithm for F_A and F_B. In both cases, the vertical axis shows the fractional reduction in the latency of *root* if top-k services are optimized by 2.



(a). Consistency of α services in F_A



(b). Consistency of α services in F_B

Figure 9. Consistency of the α services for different values of θ for F_A and F_B. For F_B, the α (hotspot) services are exactly the same for all values of θ.

seen that optimizing the services returned by our algorithm has a greater impact in reducing the response time of F_A and F_B. On an average, the impact of top-k services computed using our approach has twice the impact than those computed using the baseline. We consistently obtained an improvement of nearly 0.3 compared to the maximum of 0.5 by optimizing only $k = 3$ services.

9.3 Impact of θ and ranking services

One of the challenges in applying our framework is deciding on an appropriate improvement factor θ for computing the top-k services in SCG instance. Also, the improvement factors are not always achievable in practice. For example, it is extremely difficult to reduce the latency by $\theta = 2$ for many services, especially where the limitations are due to underlying hardware constraints. However, we found in our experiments that the hotspot services detected are usually the same irrespective of the θ value used.

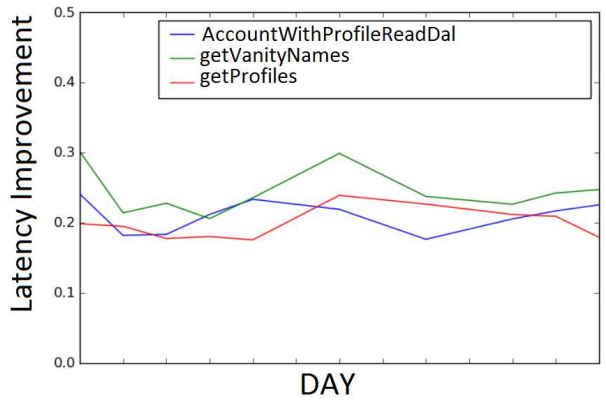
Let $\theta^* = \{\theta_1, \theta_2, \ldots, \theta_m\}$, be a set of optimization factors and H_i be the set of top-k hotspots using θ_i as the optimization factor. For any $h \in \bigcup H_i$, we define its consistency as follows:

$$consistency(h) = \frac{|\{H_i | h \in H_i\}|}{m}$$

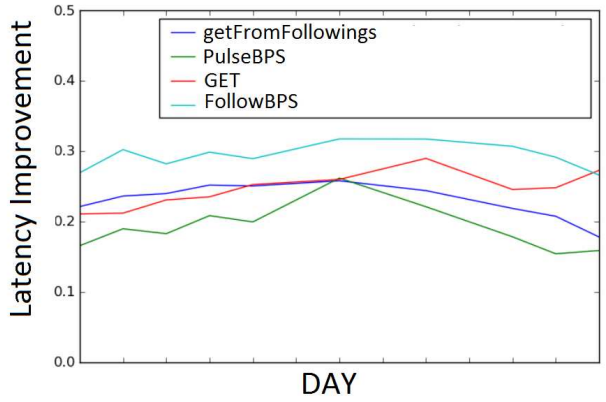
We can see that the higher the value of consistency, the lesser is the effect of improvement factor. Figure 9 shows the consistency of the hotspot services for F_A and F_B. On the horizontal axis are the hotspot services and the vertical axis shows their consistency values. We used 7 different θ values in the interval $(1, 2]$. A consistency value of 1 indicates that the hotspot services returned by our algorithm are the same for all 7 different values of θ. We can come to a similar conclusion by observing the projected improvement values for various values of θ_i in Figure 7.

9.4 Consistency over a time period

The offline version of the algorithm runs every day computing the α (hotspot) services to optimize for each F in the system. The output returned by the hotspot detection framework is meaningful if it returns similar results over a period of time. This gives us additional confidence that the hotspot services are indeed the best services to optimize. We analyzed the results of our algorithm over a period of 12 days. The hotspot services computed each day were not only similar but also have a similar impact on the F if they were optimized. For example, Figures 10a and 10b show the fractional improvement in the response time of requests to F_A and F_B. The



(a). Improvement in latency of F_A



(b). Improvement in latency of F_B

Figure 10. Fractional reduction in latencies of F_A, F_B over a period of 12 days. The hotspot services shown in the legend are not only consistent but also have similar average impact on the latency of *root*.

improvement factor $\theta = 2$. The improvement on a given day is averaged across all SCG instances.

10. CONCLUSION

In this paper, we addressed the problem of finding hotspots in a service-oriented architecture. This problem is challenging due to the service call graph's non-uniform and dynamic nature.

Our approach is to find hotspots in a specific request and then use a frequent pattern mining approach to compute and rank these hotspot services. This method can be used to detect hotspots in terms of latency, cost to serve, or any other metric. Our approach is scalable and can run both online or offline. We evaluate our algorithm on production data from LinkedIn, which shows significant improvement over baseline models.

References

[1] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *IMC*, pages 101–114, 2003.

[2] N. Amenta, F. Clarke, and K. S. John. A linear-time majority tree algorithm. In *Algorithms in Bioinformatics*, volume 2812 of *Lecture Notes in Computer Science*, pages 216–227. Springer Berlin Heidelberg, 2003.

[3] C. Borgelt. Efficient implementations of apriori and eclat. In *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003)*, page 90, 2003.

[4] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2), 2008.

[5] A. Cuzzocrea, D. Katsaros, Y. Manolopoulos, and A. Papadimitriou. EBC: A topology control algorithm for achieving high qos in sensor networks. In *QSHINE*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 613–626, 2009.

[6] U. Feige. A threshold of Ln N for approximating set cover. *J. ACM*, 45(4):634–652, July 1998.

[7] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[8] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[9] B. Goethals. Survey on frequent pattern mining. Technical report, 2003.

[10] H. Gou and Y. Yoo. Distributed bottleneck node detection in wireless sensor network. In *CIT*, pages 218–224. IEEE Computer Society, 2010.

[11] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, Aug. 2007.

[12] K. Harfoush, A. Bestavros, and J. W. Byers. Measuring bottleneck bandwidth of targeted path segments. In *INFOCOM*, 2003.

[13] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In *SIGCOMM*, pages 41–54, New York, NY, USA, 2004. ACM.

[14] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. A measurement study of internet bottlenecks. In *INFOCOM*, pages 1689–1700. IEEE, 2005.

[15] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[16] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the parallel execution of black-box services. In *HotCloud*, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[17] M. Marzolla and R. Mirandola. Performance prediction of web service workflows. In S. Overhage, C. A. Szyperski, R. Reussner, and J. A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2007.

[18] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, G. Yann-Gael, B. Baudry, and J.-M. J'ez'equel. Specification and Detection of SOA Antipatterns. In F. T. Chengfei Liu, Heiko Ludwig, editor, *Proccedings of the International Conference on Service Oriented Computing (ICSOC)*, Shanghai, Chine, Nov. 2012.

[19] K. Ostrowski, G. Mann, and M. Sandler. Diagnosing latency in multi-tier black-box services. In *LADIS*, 2011.

[20] V. J. Ribeiro, R. H. Riedi, and R. G. Baraniuk. Spatio-temporal available bandwidth estimation for high-speed networks. In *In Proc. of the First Bandwidth Estimation Workshop*, 2003.

[21] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[22] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[23] S. Sul and T. L. Williams. Fast hashing algorithms to summarize large collections of evolutionary trees, 2008.

[24] C. Wang, S. Wang, and Y. Wei. A coarse-grained bottleneck detection method with data buffer mechanism for wireless sensor networks. In *2nd International Conference on Computer Science and Network Technology (ICCSNT)*, pages 2015–2018, 2012.