Modeling the Parallel Execution of Black-Box Services

Gideon Mann
Google Inc.
New York, NY 10011

Mark Sandler
Google Inc.
New York, NY 10011

Darja Krushevskaja
Rutgers University
New Brunwick, NJ 08901

Sudipto Guha
University of Pennsylvania
Philadelphia, PA 19104

Eyal Even-dar
Final Inc.
Herzliya-Pituach, Israel 46733

Abstract

Services running in a data center frequently rely on RPCs to child services (e.g. storage, cache, authentication), and their latency depends crucially on latencies of those RPCs. However, even though service latency often comes exclusively from the time spent inside remote calls, it is difficult to determine parent latency since multithreading and asynchronous RPCs lead to complex and non-linear dependencies between service and RPC latencies. In this paper, we present a model that can be used to estimate parent latency given RPC latencies, where the parallel dependencies among of child services are modeled by an "execution flow", a direct acyclic graph. The model is learned from samples collected by a distributed tracing tool. Experiments demonstrate that these models are better able to predict top-level parent latency from child latency than state-of-the-art baselines such as linear regression and critical path analysis.

1 Introduction

In a modern datacenter front-end services are built on top of a rich ecosystem of lower level services (e.g. storage) typically accessed via RPCs. The latency of a front-end service has non-linear dependencies on the child services it relies upon, not in the least because of many services are multi-threaded and gain significant efficiency by executing calls to children services in parallel. This paper describes a method for learning a model for the parallelism of children services for a given service. This in turn enables performance bottleneck detection and "what-if" analysis. To the best of our knowledge no existing tool is able to perform this type of diagnosis, we give the overview of related work in Section 2.

In this paper we introduce a directed acyclic graph model to predict overall latency of black-box systems (Sections 3 and 4). This model, an "execution flow", encodes all feasible critical paths in a service. We demonstrate that execution flows can be learned from the call traces provided by a distributed tracing system and prediction accuracy based on flows is better than other state-of-the-art models, such as linear regression and critical path analysis (Section 5). We believe that execution flows capture a sweet spot in the analysis space in term of clarity, expressiveness, and prediction power. The added benefit of our model is that we do not need to analyze the underlying source code to build flows, but instead induce them by observing RPCs.

2 Related Work

Path profiling [3] is non-trivial even in a non-distributed setting. In a distributed setting, the network weather service [14] is one of the first tools to perform analysis with statistical learning techniques. It allows forecasting of service latency based on past latencies (not on the observed child latencies, as in this work). Since then, analysis using statistical techniques has became a prominent tool in debugging complex programs [7].

Stardust [12] is an early distributed profilers and collects execution traces for a service and sub-services. Similarly, Dapper [11] and X-Trace [5] are lightweight annotation-based tools that collect application-level RPC information by providing a few modules that are commonly linked in all services. Aguilera [1] and Magpie [4] perform more careful profiling but assume a nested call structure. In real systems, because of asynchronous calls and multithreading, this nested structure assumption is frequently violated.

In contrast, Pip [9] allows developers to encode arbitrary parallelism, but relies on explicit validators that check correspondence to a valid or invalid execution path. Similarly, work by Urgaonkar et al. [13] and Dejun et al. [6] address multi-tier systems and adds support for some amount of concurrency. However these models assume that each service follows a fixed M-tiered setup with limited variation.

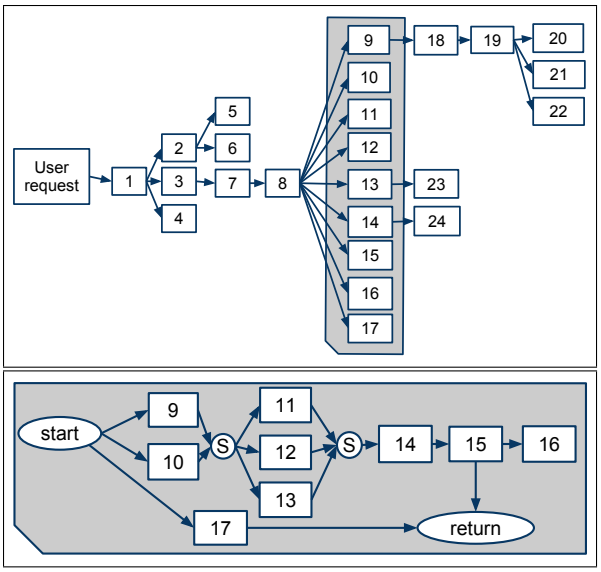

Figure 1: **(top)** The trace (call tree) of a service with relatively large stack. Note that it is impossible to tell the ordering of methods 9 through and 17 that are all called from service 8. **(bottom)** Induced execution flow (defined in Section 4) for service 8 describing its calls to children services (9 - 17). Unlike the above figure, edges indicate temporal dependencies, e.g service 11 starts only after services 9 and 10 have returned. We use a virtual node marked as "S" to denote a synchronization point.

Another line of work, which relies on latency models to diagnose performance problem is by Sambasivan et al [10]. They showed very interesting experimental results on applying simple latency models to detect performance problems. In contrast in this paper we concentrate on latency models and show their comparative accuracy.

3 Latency Analysis

A **service** is an arbitrary, potentially multi-threaded, program, running in a data center that can issue RPCs to **children services**. To avoid confusion we will refer to the issuing service as **parent**. The goal of this work is to build a model for the parent latency *given* the latencies of its children. Unlike [14], the value of the model isn't the raw predictions per se, but rather to gain a deeper understanding of a service which can be later used in evaluation of what-if scenarios and root cause analysis.

A distributed profiling tool collects a set of **traces**, where each trace is an augmented call tree for a service invocation. Each node in the call tree represents a RPC to a child service that generally would be running on a different machine or even in a different data center. Each node contains metadata about the service and the context of the request, such as the method name of service executed, size of request and response, and timing information. Figure 1 **(top)** depicts a trace where a user request initiated a sequence of calls. An edge in the graph indicates that one service called the other, e.g. parent service 1 calls children services 2,3, and 4.

Formally, for a particular invocation $\mathcal{I}$ of a parent service we assume the set of children services $m_1, m_2 \dots m_k$. We define the following functions on a particular trace: $L_{\mathcal{I}}(m_i)$ the latency of method m_i, and $P_{\mathcal{I}}(m_i)$ the preprocessing that the service had to do before RPC m_i can be called. The preprocessing time for RPC m_i is estimated as the time difference before the latest RPC that finishes before m_i and the start of m_i. During training, the system has access to the actual parent latency $L_{\mathcal{I}}$ and learns an estimator $\hat{L}_{\mathcal{I}}$ over $L_{\mathcal{I}}(m_i)$ and $P_{\mathcal{I}}(m_i)$.

The most simple models for the overall service latency are: **(1) purely sequential children**: $\hat{L}_I = \sum_{m_i} L_{\mathcal{I}}(m_i) + P_{\mathcal{I}}(m_i)$ **(2) purely parallel children**: $\hat{L}_I = \max_{m_i}(L_{\mathcal{I}}(m_i) + P_{\mathcal{I}}(m_i))$. If either of these models worked well then further analysis would be unnecessary. However, our experiments show that both of these methods have very poor accuracy indicating that there is indeed non-trivial internal flow structure, that controls the latency (see Table 1 in Section 5).

3.1 Linear Regression

The latency prediction problem can also be formulated as a classical regression problem: predict the latency of a parent service given latencies of children services. As a baseline model, we use the least squared error criterion to find the best model parameters w: $\hat{L}_I = \sum_i w_i L_{\mathcal{I}}(m_i)$. Note that the linear regression model itself encodes no information regarding relative order or dependencies between children services. Further, as opposed to our approach, it fails to generalize to detect the case when the services that were never been a performance bottleneck, suddenly become one, yet experiments show that it is a useful baseline.

3.2 Critical Paths

A *critical path* is defined as a subset of RPC calls to children services, such that decreasing the latency of any of the calls decreases the overall latency. Essentially, the critical path represents the blocking relationships between a sequence of siblings in a call tree.

To build a critical path model we use the following greedy search. Given a collection of calls $\{m_i\}$ for a partial trace $\mathcal{I}$, we find the RPC m_{i_1} that is the last to end before the service returns and include it in the path.

Then we iterate, by finding the RPC m_{i_l} that is the last to finish before $m_{i_{l-1}}$.

To use critical paths for prediction, for each particular service we collect all observed critical paths during the training. During the testing phase, for each trace $\mathcal{I}$, with latency function $L_\mathcal{I}$ defined on method set $\mathcal{M}$ we locate the set of critical paths $\mathcal{C}$ found during training, and compute $\hat{L}_\mathcal{I} = \max_{C \in \mathcal{C}_\mathcal{M}} \sum_{m_i \in C} P_\mathcal{I}(m_i) + L_\mathcal{I}(m_i)$ as an estimate of the parent latency.

For complex services, critical path analysis becomes brittle. For instance, consider a service that calls many different children services in parallel, waits for them to finish, and then issues another large batch of parallel calls. The critical path in this case would be the two calls that took the longest in each batch, but it can be easily seen that the total number of potential critical paths can grow exponentially. Thus as services grow in size, critical paths become inadequate for understanding the total latency.

4 Execution Flows

For a trace $\mathcal{I}$, we construct an **invocation graph** $G_\mathcal{I} = (\mathcal{M}, E_\mathcal{I})$ that encodes the relative order of the methods as they were executed in that particular trace. Note that since an invocation representa RPCs originating from a single machine. Therefore, despite the fact that clocks across machines are not synchronized, the individual invocations do in fact have reasonably synchronized timestamps.

An edge $(m_i, m_j) \in E_\mathcal{I}$ iff method m_i finished before method m_j[1]. A invocation graph may have more edges than dictated by the service, since it is possible that method m_i finishes before method m_j during that trace, but they actually are run in parallel and are not dependent on one another. These spurious constraints are resolved via gathering multiple traces.

For service $\mathcal{S}$ (or a collection of traces) the **execution flow** is a graph $G_\mathcal{S} = (\mathcal{M}, E_\mathcal{I})$ similar to the invocation graph. In an execution flow, there is an edge from node m_i to node m_j if and only if method m_i finishes before method m_j starts in all traces for that service (i.e. m_j is blocked by m_i). Figure 1 **(bottom)** illustrates execution flow of parent service 8 in the call tree above it.

Given an execution flow and latencies for each child service, we can predict the total latency by finding the longest blocking path through the graph – the effective critical path. In this manner, the execution flows compactly encode exponentially many possible critical paths, even those that have never been observed. Often a particular service might have several different execution flows corresponding to different input or different internal service state such as cache miss might cause a service to follow a different execution path.

Aggregate Flows

In this setting, we collect all invocation graphs and find the maximal graph G that is **consistent** with each of these distinct invocation graphs, where consistency is defined as follows: a G_a is consistent with G_b if $E_a \subseteq E_b$, and the maximal graph is simply $\cap_\mathcal{I} E_\mathcal{I}$. In this manner, if a service S requires two methods to be in a particular order, they will be in order in all instances and thus the algorithm can uncover the parallelism invariants.

Since the invocation graph is acyclic, at prediction time we perform topological sorting on all the nodes in the graph, so that if there is an edge from m_i to m_j then $i < j$. Then we compute c_i completion time for method m_i using dynamic programming. We iterate over all nodes to compute its completion time:

$$c(m_i) = P_\mathcal{I}(m_i) + L_\mathcal{I}(m_i) + \max_{j,s.t.(m_j,m_i)\in E} c(m_j).$$

The total time for the method is completion time of *return* node: $\hat{L}_\mathcal{I} = c(return)$.

Nearest-Neighbor Flow

A more robust estimator is to cluster traces together only if they have identical invocation graphs.[2] This can ameliorate modeling errors[3] and learning errors from noise in the underlying data collection and incorrect traces.

In this setting, we simplistically model each flow's latency using a normal distribution whose mean and variance are estimated from the observed traces, so $\hat{L}_\mathcal{S} \sim N(\bar{L}(\mathcal{M}), \bar{\sigma}(\mathcal{M}))$. This allows us to apply nearest neighbor by choose the nearest flow according to the metric:

$$d_\mathcal{I}(\mathcal{F}) = \sum_{i \in M} \frac{(L_\mathcal{I}(m_i) - \bar{L}_\mathcal{F}(m_i))^2}{\sigma^2_\mathcal{F}(m_i)}.$$

Given a flow, we then use the dynamic program above to estimate parent latency. From a statistical perspective, if the underlying latencies follow the Gaussian distribution[4], the flow which minimizes $d_\mathcal{I}(\mathcal{F})$ has the highest

[1] Similarly, every node m_i has an edge from `start` node to m_i and if the service waits for method m_j to finish before returning, there is an edge from m_j to `return`. Additionally, we omit all edges implied by the transitive closure since they do not introduce any new dependencies.

[2] Note that two instances may have two identical invocation graphs and completely different critical paths.

[3] E.g. when services violate the model assumptions such as having logical barriers where x of out y calls must return for timeouts or for replicated writes.

[4] We note that a gaussian distribution is likely to be incorrect for this data. Log-gaussian or a non-parametric approach might be better, we leave these experiments for future work.

probability of generating the latencies that we have observed.

5 Experimental Results

We compare the error of the algorithms, via a sample of all traces that were collected in a large datacenter (thousands of machines) over the course of two hours[5]. We collected traces from across all services running in that data center. In the data we use we accommodate clock skew, packet loss, and other hardware errors present in deployed systems. The data from the first hour was used for training of our latency predictors. The data from the second hour formed the test dataset. For testing, for each service invocation we measured how well the algorithms were able to reconstruct the total latency of the call given the RPC latencies.

In our sample, we ignored all parent services with fewer than five child services – these services are too simple in structure to benefit from our analysis. We compare six approaches for predicting total latency: (1) pure serial execution, (2) pure parallel execution, (3) linear regression, (4) best critical path, (5) aggregate flows, (6) nearest neighbor flows.

The graph on Figure 2 (*left*) plots the ratio of **estimated/actual** latencies for different methods, where all traces are ordered by that ratio in the increasing order. In essence, by looking at intersection points of each curve and lines $y = \alpha$ and $y = \beta$, we can compute the fraction of RPCs that have the error factor in the range $[\alpha, \beta]$ for each of the methods. Similarly, the graph in Figure 2 (right) plots the **discrepancy** (defined as $\frac{\max(estimated,actual)}{\min(estimated,actual)}$) and allows to find the fraction of all predictions that are within multiplicative factor of α of the true value. The latter graph depicts error symmetrically so that it is easier to visualize the cumulative estimation error. We summarize these graphs in Table 1.

The most notable conclusions to be taken from Table 1 and Figure 2 is that the **nearest neighbor flow** algorithm is the most effective at predicting parent service latency. As expected, **serial** and **parallel** approaches provide very crude approximations for lower and upper bounds on the total latency. The parallel approximation sometimes overestimates the latency due to times when the method returns without waiting for some children services to complete (e.g. a service may return before a write finishes).

The second conclusion is that the error from many of the other approaches is asymmetric. The **aggregated execution flow** tends to underestimate the latency which is expected, since the flow contains minimum number of temporal dependencies, and thus it produces a very optimistic estimate of the parent latency. Similarly **best critical path** tends to overestimates the latency because a certain path might be a critical path in one instance but not in another.

[5]For confidentiality reasons beyond our control, we are forbidden from disclosing the exact number of services, average service size and traces collected during this time period.

	50%	90%	95%	99%
Serial	0.43	11.44	62.05	∞
Parallel	0.60	4.57	21.96	∞
Aggregate Flw	0.00	1.11	3.60	102.18
Critical Path	0.00	0.85	1.81	37.40
Linear Regression	0.02	0.58	1.22	28.74
Nearest Neighbor Flw	0.00	0.36	0.73	4.39

Table 1: Relative errors for different approaches and different percentiles. Nearest Neighbor is best overall, and 99% of all its predictions have relative error less than 4.39 times true true latency.

Finally, **linear regression** performed surprisingly well provided that it does not infer any information about internal structure of the service. However it minimizes the prediction error and so "probabilistically" represents the true underlying latency. Even though the performance gains are slight, we believe that execution flows are more useful than linear models since (1) they have a reasonable model of the underlying process, (2) they are simple, and (3) they can be transparently understood by system developers. Furthermore, we expect future experiments will validate their utility.

6 Conclusions and Future Work

This paper has presented several approaches to predict latency of a service given child service latencies. Our experimental results indicate that the nearest neighbor flow based approac has superior accuracy when compared to other approaches and also (as opposed to linear regression) provides useful visualization of the service flow. However the main utility of latency models is not in the prediction of latency per se, but in enabling a new kind of diagnostic and forecasting applications. A few applications and future work include:

- **Measuring the impact of a change** Using the flow models, we can explore the impact of subtle changes, such as decreasing inner-cluster network or a change in a low-level method, on the latency on user-visible requests that are several software software layers removed. The models can further suggest whether or not a certain optimization or reconfiguration are worth implementing.

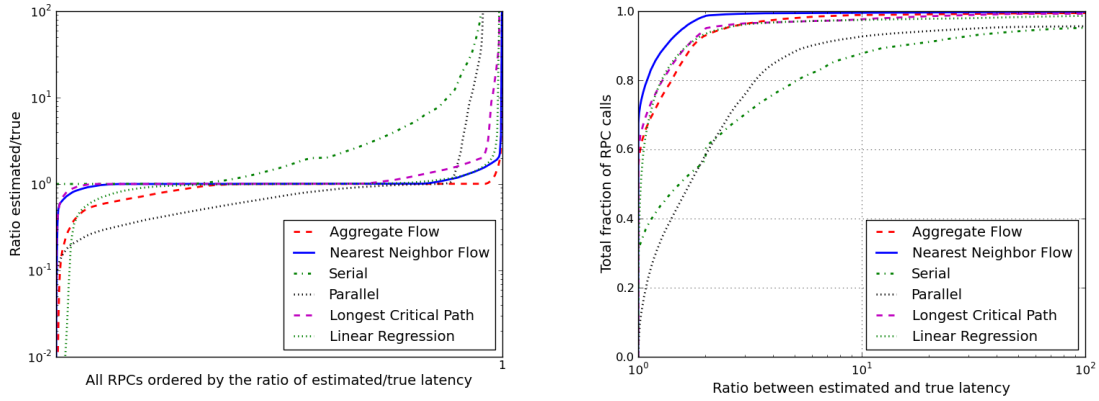

Figure 2: Distribution of estimation errors across all parent services. x-axis is normalized to represent traffic fraction. A perfect estimator would be the line $y = 1$ on both graphs. **(left)** The y-axis is the log-scaled ratio between estimated and actual latencies. The part of the curve below $y = 1$ are underestimated latencies, and the above it are overestimated. **(right)** The x-axis contains the log-scaled relative error between estimated and actual latencies (always > 1). To interpret, at $x = 1.3$ **Nearest Neighbor Flow** $\approx 90\%$, showing that 90% of all predictions have relative error less than 0.3.

- **Service allocation** Executions flows could facilitate research on meeting SLAs [2, 8] by allowing service owners to estimate viable SLAs given SLAs they have for the child methods, or what would be an optimal SLA with various software stacks, given the budget.

- **Root cause analysis and change detection** In highly distributed multi-tiered services change is a constant. Often service owners who run identical services in two different locations in two different data centers see extremely different behavior. Flows provide a mechanism to estimate which of the underlying factors (potentially nested within each other) are responsible for the difference in behavior.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003), ACM, pp. 74–89.

[2] APPLEBY, K., FAKHOURI, S., FONG, L., GOLDSZMIDT, G., KALANTAR, M., KRISHNAKUMAR, S., PAZEL, D., PERSHING, J., AND ROCHWERGER, B. Oceano - sla based management of a computing utility. In *Symposium on Integrated Network Management* (2001).

[3] BALL, T., AND LARUS, J. R. Efficient path profiling. In *MICRO* (1996), IEEE, pp. 46–57.

[4] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: online modelling and performance-aware systems. In *HotOS-IX* (2003), USENIX.

[5] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007), USENIX, pp. 271–284.

[6] JIANG, D., PIERRE, G., AND CHI, C.-H. Autonomous resource provisioning for multi-service web applications. In *WWW* (2010), ACM, pp. 471–480.

[7] LIBLIT, B. R. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[8] RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. Qos-driven server migration for internet data centers. In *Workshop on QoS* (2002), IEEE.

[9] REYNOLDS, P., KILLIAN, C., WIENER, J., MOGUL, J., SHAH, M., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006), pp. 115–128.

[10] SAMBASIVAN, R., ZHEN., A., ROSA, M. D., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., L, , AND GANGER, G. Diagnosing performance changes by comparing request flows. In *NSDI* (2011), USENIX.

[11] SIGELMAN, B., BARROSO, L., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google Inc, 2010.

[12] THERESKA, E., SALMON, B., STRUNK, J., WACHS, M., ABD-EL-MALEK, M., LOPEZ, J., AND GANGER, G. Stardust:tracking activity in a distributed storage system. In *SIGMETRICS* (2006), ACM.

[13] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS* (2005).

[14] WOLSKI, R. Dynamically forecasting network performance using the network weather service. *Cluster Computing 1* (January 1998), 119–132.