The Mystery Machine: End-to-end performance analysis of large-scale Internet services

Michael Chow[*], *David Meisner*[†], *Jason Flinn*[*], *Daniel Peek*[†], *Thomas F. Wenisch*[*]
University of Michigan[*] *Facebook, Inc.*[†]

Abstract

Current debugging and optimization methods scale poorly to deal with the complexity of modern Internet services, in which a single request triggers parallel execution of numerous heterogeneous software components over a distributed set of computers. The Achilles' heel of current methods is the need for a complete and accurate model of the system under observation: producing such a model is challenging because it requires either assimilating the collective knowledge of hundreds of programmers responsible for the individual components or restricting the ways in which components interact.

Fortunately, the scale of modern Internet services offers a compensating benefit: the sheer volume of requests serviced means that, even at low sampling rates, one can gather a tremendous amount of empirical performance observations and apply "big data" techniques to analyze those observations. In this paper, we show how one can automatically construct a model of request execution from pre-existing component logs by generating a large number of potential hypotheses about program behavior and rejecting hypotheses contradicted by the empirical observations. We also show how one can validate potential performance improvements without costly implementation effort by leveraging the variation in component behavior that arises naturally over large numbers of requests to measure the impact of optimizing individual components or changing scheduling behavior.

We validate our methodology by analyzing performance traces of over 1.3 million requests to Facebook servers. We present a detailed study of the factors that affect the end-to-end latency of such requests. We also use our methodology to suggest and validate a scheduling optimization for improving Facebook request latency.

1 Introduction

There is a rich history of systems that understand, optimize, and troubleshoot software performance, both in practice and in the research literature. Yet, most of these prior systems deal poorly with the complexities that arise from modern Internet service infrastructure. Complexity comes partially from *scale*; a single Web request may trigger the execution of hundreds of executable components running in parallel on many different computers. Complexity also arises from *heterogeneity*; executable components are often written in different languages, communicate through a wide variety of channels, and run in execution environments that range from third-party browsers to open-source middleware to in-house, custom platforms.

In this paper, we develop performance analysis tools for measuring and uncovering performance insights about complex, heterogeneous distributed systems. We apply these tools to the Facebook Web pipeline. Specifically, we measure *end-to-end performance* from the point when a user initiates a page load in a client Web browser, through server-side processing, network transmission, and JavaScript execution, to the point when the client Web browser finishes rendering the page.

Fundamentally, analyzing the performance of concurrent systems requires a model of application behavior that includes the causal relationships between components; e.g., *happens-before* ordering and mutual exclusion. While the techniques for performing such analysis (e.g., critical path analysis) are well-understood, prior systems make assumptions about the ease of generating the causal model that simply do not hold in many large-scale, heterogeneous distributed systems such as the one we study in this paper.

Many prior systems assume that one can generate such a model by comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [1, 3, 13, 18, 22, 24, 28]. This is a reasonable assumption if the software architecture is homogeneous; for instance, Dapper [28] instruments a small set of middleware components that are widely used within Google.

However, many systems are like the Facebook systems we study; they grow organically over time in a culture that favors innovation over standardization (e.g., "move fast and break things" is a well-known Facebook slogan). There is broad diversity in programming languages, communication middleware, execution environments, and scheduling mechanisms. Adding instrumentation retroactively to such an infrastructure is a Herculean task. Further, the end-to-end pipeline includes client software such as Web browsers, and adding detailed instrumentation to all such software is not feasible.

Other prior systems rely on a user-supplied schema that expresses the causal model of application behavior [6, 31]. This approach runs afoul of the scale of modern Internet services. To obtain a detailed model of end-to-end request processing, one must assemble the collective knowledge of hundreds of programmers responsible for the individual components that are involved in request processing. Further, any such model soon grows stale due to the constant evolution of the system under observation, and so constant updating is required.

Consequently, we develop a technique that generates a causal model of system behavior without the need to add substantial new instrumentation or manually generate a schema of application behavior. Instead, we generate the model via large-scale reasoning over individual software component logs. Our key observation is that the sheer volume of requests handled by modern services allows us to gather observations of the order in which messages are logged over a tremendous number of requests. We can then hypothesize and confirm relationships among those messages. We demonstrate the efficacy of this technique with an implementation that analyzes over 1.3 million Facebook requests to generate a comprehensive model of end-to-end request processing.

Logging is an almost-universally deployed tool for analysis of production software. Indeed, although there was no comprehensive tracing infrastructure at Facebook prior to our work, almost all software components had some individual tracing mechanism. By relying on only a minimum common content for component log messages (a request identifier, a host identifier, a host-local timestamp, and a unique event label), we unified the output from diverse component logs into a unified tracing system called *ÜberTrace*.

ÜberTrace's objective is to monitor *end-to-end request latency*, which we define to be the time that elapses from the moment the user initiates a Facebook Web request to the moment when the resulting page finishes rendering. *ÜberTrace* monitors a diverse set of activities that occur on the client, in the network and proxy layers, and on servers in Facebook data centers. These activities exhibit a high degree of concurrency.

To understand concurrent component interactions, we construct a causality model from a large corpus of *ÜberTrace* traces. We generate a cross-product of possible hypotheses for relationships among the individual component events according to standard patterns (currently, happens-before, mutual exclusive, and first-in-first-out relationships). We assume that a relationship holds until we observe an explicit contradiction. Our results show that this process requires traces of hundreds of thousands of requests to converge on a model. However, for a service such as Facebook, it is trivial to gather traces at this scale even at extremely low sampling frequencies. Further, the analysis scales well and runs as a parallel Hadoop job.

Thus, our analysis framework, *The Mystery Machine* derives its causal model solely from empirical observations that utilize only the existing heterogeneous component logs. *The Mystery Machine* uses this model to perform standard analyses, such as identifying critical paths, slack analysis, and outlier detection.

In this paper, we also present a detailed case study of performance optimization based on results from *The Mystery Machine*. First, we note that whereas the average request workload shows a balance between client, server, and network time on the critical path, there is wide variance in this balance across individual requests. In particular, we demonstrate that Facebook servers have considerable slack when processing some requests, but they have almost no slack for other requests. This observation suggests that end-to-end latency would be improved by having servers produce elements of the response as they are needed, rather than trying to produce all elements as fast as possible. We conjecture that this just-in-time approach to response generation will improve the end-to-end latency of requests with no slack while not substantially degrading the latency of requests that currently have considerable slack.

Implementing such an optimization is a formidable task, requiring substantial programming effort. To help justify this cost by partially validating our conjecture, we use *The Mystery Machine* to perform a "what-if" analysis. We use the inherent variation in server processing time that arises naturally over a large number of requests to show that increasing server latency has little effect on end-to-end latency when slack is high. Yet, increasing server latency has an almost linear effect on end-to-end latency when slack is low. Further, we show that slack can be predicted with reasonable accuracy. Thus, the case study demonstrates two separate benefits of *The Mystery Machine*: (1) it can identify opportunities for performance improvement, and (2) it can provide preliminary evidence about the efficacy of hypothesized improvements prior to costly implementation.

2 Background

In the early days of the Web, a request could often be modeled as a single logical thread of control in which a client executed an RPC to a single Web server. Those halcyon days are over.

At Facebook, the end-to-end path from button click to final render spans a diverse set of systems. Many components of the request are under Facebook's control, but several components are not (e.g., the external network and the client's Web browser). Yet, users care little about who is responsible for each component; they simply desire that their content loads with acceptable delay.

A request begins on a client with a user action to retrieve some piece of content (e.g., a news feed). After

DNS resolution, the request is routed to an Edge Load Balancer (ELB) [16]. ELBs are geo-distributed so as to allow TCP sessions to be established closer to the user and avoid excessive latency during TCP handshake and SSL termination. ELBs also provide a point of indirection for better load balancing, acting as a proxy between the user and data center.

Once a request is routed to a particular data center, a Software Load Balancer routes it to one of many possible Web servers, each of which runs the HipHop Virtual Machine runtime [35]. Request execution on the Web server triggers many RPCs to caching layers that include Memcache [20] and TAO [7]. Requests also occasionally access databases.

RPC responses pass through the load-balancing layers on their way back to the client. On the client, the exact order and manner of rendering a Web page are dependent on the implementation details of the user's browser. However, in general, there will be a Cascading Style Sheet (CSS) download stage and a Document Object Model rendering stage, followed by a JavaScript execution stage.

As with all modern Internet services, to achieve latency objectives, the handling of an individual request exhibits a high degree of concurrency. Tens to hundreds of individual components execute in parallel over a distributed set of computers, including both server and client machines. Such concurrency makes performance analysis and debugging complex. Fortunately, standard techniques such as critical path analysis and slack analysis can tame this complexity. However, all such analyses need a model of the causal dependencies in the system being analyzed. Our work fills this need.

3 *ÜberTrace*: End-to-end Request Tracing

As discussed in the prior section, request execution at Facebook involves many software components. Prior to our work, almost all of these components had logging mechanisms used for debugging and optimizing the individual components. In fact, our results show that individual components are almost always well-optimized *when considered in isolation*.

Yet, there existed no complete and detailed instrumentation for monitoring the end-to-end performance of Facebook requests. Such end-to-end monitoring is vital because individual components can be well-optimized in isolation yet still miss opportunities to improve performance when components interact. Indeed, the opportunities for performance improvement we identify all involve the interaction of multiple components.

Thus, the first step in our work was to unify the individual logging systems at Facebook into a single end-to-end performance tracing tool, dubbed *ÜberTrace*. Our basic approach is to define a minimal schema for the information contained in a log message, and then map existing log messages to that schema.

ÜberTrace requires that log messages contain at least:

1. A unique request identifier.

2. The executing computer (e.g., the client or a particular server)

3. A timestamp that uses the local clock of the executing computer

4. An event name (e.g., "start of DOM rendering").

5. A task name, where a task is defined to be a distributed thread of control.

ÜberTrace requires that each <event, task> tuple is unique, which implies that there are no cycles that would cause a tuple to appear multiple times. Although this assumption is not valid for all execution environments, it holds at Facebook given how requests are processed. We believe that it is also a reasonable assumption for similar Internet service pipelines.

Since all log timestamps are in relation to local clocks, *ÜberTrace* translates them to estimated global clock values by compensating for clock skew. *ÜberTrace* looks for the common RPC pattern of communication in which the thread of control in an individual task passes from one computer (called the client to simplify this explanation) to another, executes on the second computer (called the server), and returns to the client. *ÜberTrace* calculates the server execution time by subtracting the latest and earliest server timestamps (according to the server's local clock) nested within the client RPC. It then calculates the client-observed execution time by subtracting the client timestamps that immediately succeed and precede the RPC. The difference between the client and server intervals is the estimated network round-trip time (RTT) between the client and server. By assuming that request and response delays are symmetric, *ÜberTrace* calculates clock skew such that, after clock-skew adjustment, the first server timestamp in the pattern is exactly 1/2 RTT after the previous client timestamp for the task.

The above methodology is subject to normal variation in network performance. In addition, the imprecision of using existing log messages rather than instrumenting communication points can add uncertainty. For instance, the first logged server message could occur only after substantial server execution has already completed, leading to an under-estimation of server processing time and an over-estimation of RTT. *ÜberTrace* compensates by calculating multiple estimates. Since there are many request and response messages during the processing of a higher-level request, it makes separate RTT and clock skew calculations for each pair in the cross-product of requests. It then uses the calculation that yields the lowest observed RTT.

Timecard [23] used a similar approach to reconcile timestamps and identified the need to account for the effects of TCP slow start. Our use of multiple RTT estimates accomplishes this. Some messages such as the initial request are a single packet and so are not affected by slow start. Other messages such as the later responses occur after slow start has terminated. Pairing two such messages will therefore yield a lower RTT estimate. Since we take the minimum of the observed RTTs and use its corresponding skew estimate, we get an estimate that is not perturbed by slow start.

Due to performance considerations, Facebook logging systems use statistical sampling to monitor only a small percentage of requests. *ÜberTrace* must ensure that the individual logging systems choose the same set of requests to monitor; otherwise the probability of all logging systems independently choosing to monitor the same request would be vanishingly small, making it infeasible to build a detailed picture of end-to-end latency. Therefore, *ÜberTrace* propagates the decision about whether or not to monitor a request from the initial logging component that makes such a decision through all logging systems along the path of the request, ensuring that the request is completely logged. The decision to log a request is made when the request is received at the Facebook Web server; the decision is included as part of the per-request metadata that is read by all subsequent components. *ÜberTrace* uses a global identifier to collect the individual log messages, extracts the data items enumerated above, and stores each message as a record in a relational database.

We made minimal changes to existing logging systems in order to map existing log messages to the *ÜberTrace* schema. We modified log messages to use the same global identifier, and we made the event or task name more human-readable. We added no additional log messages. Because we reused existing component logging and required only a minimal schema, these logging changes required approximately one person-month of effort.

4 *The Mystery Machine*

The Mystery Machine uses the traces generated by *ÜberTrace* to create a causal model of how software components interact during the end-to-end processing of a Facebook request. It then uses the causal model to perform several types of distributed systems performance analysis: finding the critical path, quantifying slack for segments not on the critical path, and identifying segments that are correlated with performance anomalies. *The Mystery Machine* enables more targeted analysis by exporting its results through a relational database and graphical query tools.

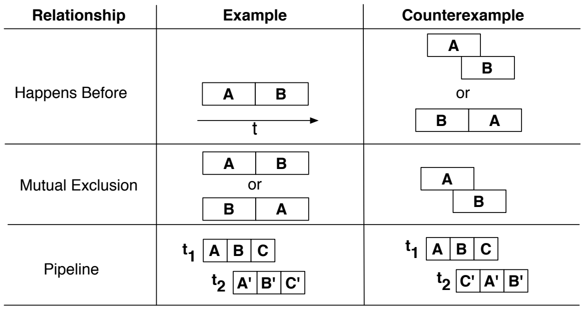


Figure 1: Causal Relationships. This figure depicts examples of the three kinds of causal relationship we consider. Happens-before relationships are when one segment (A) always finishes in its entirety before another segment (B) begins. FIFO relationships exist when a sequence of segments each have a happens-before relationship with another sequence in the same order. A mutual exclusion relationship exists when two segments never overlap.

4.1 Causal Relationships Model

To generate a causal model, *The Mystery Machine* first transforms each trace from a collection of logged events to a collection of *segments*, which we define to be the execution interval between two consecutive logged events for the same task. A segment is labeled by the tuple <task, start_event, end_event>, and the segment duration is the time interval between the two events.

Next, *The Mystery Machine* identifies causal relationships. Currently, it looks for three types of relationships:

1. **Happens-before** ($\rightarrow$) We say that segment A happens-before segment B ($A \rightarrow B$) if the start event timestamp for B is greater than or equal to the end event timestamp for A in all requests.

2. **Mutual exclusion** ($\vee$) Segments A and B are mutually exclusive ($A \vee B$) if their time intervals never overlap.

3. **Pipeline** ($\gg$) Given two tasks, t_1 and t_2, there exists a data dependency between pairs of segments of the two tasks. Further, the segment that operates on data element d_1 precedes the segment that operates on data element d_2 in task t_1 if and only if the segment that operates on d_1 precedes the segment that operates on d_2 in task t_2 for all such pairs of segments. In other words, the segments preserve a FIFO ordering in how data is produced by the first task and consumed by the second task.

We summarize these relationships in Figure 1. For each relationship we provide a valid example and at least one counterexample that would contradict the hypothesis.

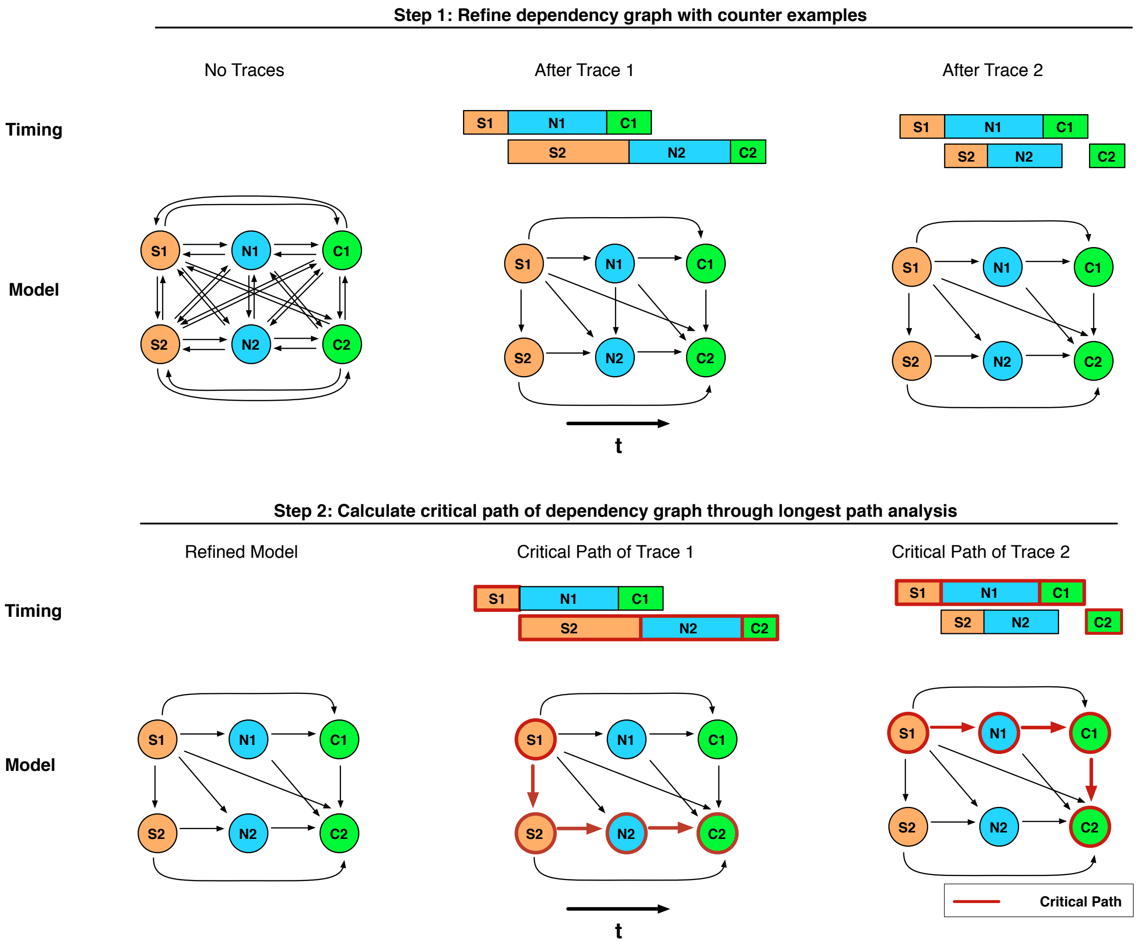


Figure 2: Dependency model generation and critical path calculation. This figure provides an example of discovering the true dependency model through iterative refinement. We show only a few segments and relationships for the sake of simplicity. Without any traces, the dependency model is a fully connected graph. By eliminating dependency edges invalidated by counterexamples, we arrive at the true model. With a refined model, we can reprocess the same traces and derive the critical path for each.

We use techniques from the race detection literature to map these static relationships to dynamic happens-before relationships. Note that mutual exclusion is a static property; e.g., two components A and B that share a lock are mutually exclusive. Dynamically, for a particular request, this relationship becomes a happens-before relationship: either A → B or B → A, depending on the order of execution. Pipeline relationships are similar. Thus, for any given request, all of these static relationships can be expressed as dynamic causal relationships between pairs of segments.

4.2 Algorithm

The Mystery Machine uses iterative refinement to infer causal relationships. It first generates all possible hypotheses for causal relationships among segments. Then, it iterates through a corpus of traces and rejects a hypothesis if it finds a counterexample in any trace.

Step 1 of Figure 2 illustrates this process. We depict the set of hypotheses as a graph where nodes are segments (”S” nodes are server segments, ”N” nodes are network segments and ”C” nodes are client segments) and edges are hypothesized relationships. For the sake of simplicity, we restrict this example to consider only happens-before relationships; an arrow from A to B shows a hypothesized “A happens before B” relationship.

The “No Traces” column shows that all possible relationships are initially hypothesized; this is a large number because the possible relationships scale quadratically as the number of segments increases. Several hypotheses are eliminated by observed contradictions in the first request. For example, since S2 happens after S1, the hypothesized relationship, S2 → S1, is removed. Further traces must be processed to complete the model. For instance, the second request eliminates the hypothesized relationship, N1 → N2. Additional traces prune new hypotheses due to the natural perturbation in timing of segment processing; e.g., perhaps the second user had less friends, allowing the network segments to overlap due to

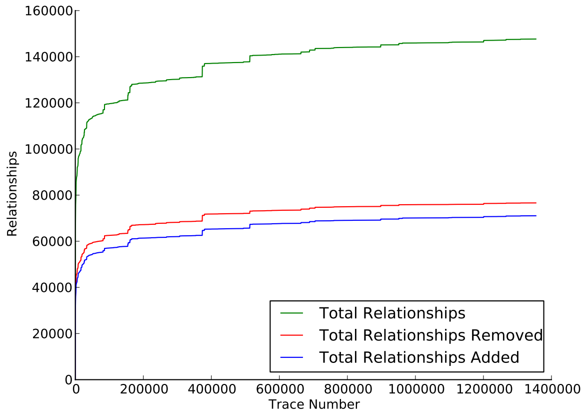


Figure 3: Hypothesis Refinement. This graph shows the growth of number of hypothesized relationships as a function of requests analyzed. As more requests are analyzed, the rate at which new relationships are discovered and removed decreases and eventually reaches a steady-state. The total number of relationships increases over time due to code changes and the addition of new features.

shorter server processing time.

The Mystery Machine assumes that the natural variation in timing that arises over large numbers of traces is sufficient to expose counterexamples for incorrect relationships. Figure 3 provides evidence supporting this hypothesis from traces of over 1.3 million requests to the Facebook home page gathered over 30 days. As the number of traces analyzed increases, the observation of new counterexamples diminishes, leaving behind only true relationships. Note that the number of total relationships changes over time because developers are continually adding new segments to the pipeline.

4.3 Validation

To validate the causal model produced by the Mystery Machine, we confirmed several specific relationships identified by the Mystery Machine. Although we could not validate the entire model due to its size, we did substantial validation of two of the more intricate components: the interplay between JavaScript execution on the client and the dependencies involved in delivering data to the client. These components have 42 and 84 segments, respectively, as well as 2,583 and 10,458 identified casual relationships.

We confirmed these specific relationships by examining source code, inserting assertions to confirm model-derived hypotheses, and consulting relevant subsystem experts. For example, the system discovered the specific, pipelined schedule according to which page content is delivered to the client. Further, the model correctly reflects that JavaScript segments are mutually exclusive (a known property of the JavaScript execution engine) and identified ordering constraints arising from synchronization.

4.4 Analysis

Once *The Mystery Machine* has produced the causal model of segment relationships, it can perform several types of performance analysis.

4.4.1 Critical Path

Critical path analysis is a classic technique for understanding how individual components of a parallel execution impact end-to-end latency [22, 32]. The critical path is defined to be the set of segments for which a differential increase in segment execution time would result in the same differential increase in end-to-end latency.

The Mystery Machine calculates the critical path on a per-request basis. It represents all segments in a request as a directed acyclic graph in which the segments are vertices with weight equal to the segment duration. It adds an edge between all vertices for which the corresponding segments have a causal relationship. Then, it performs a transitive reduction in which all edges A → C are recursively removed if there exists a path consisting of A → B and B → C that links the two nodes.

Finally, *The Mystery Machine* performs a longest-path analysis to find the critical path from the first event in the request (the initiation of the request) to the last event (which is typically the termination of some JavaScript execution). The length of the critical path is the end-to-end latency of the entire request. If there are equal-length critical paths, the first discovered path is chosen.

We illustrate the critical path calculation for the two example requests in Step 2 of Figure 2. Each request has a different critical path even though the dependency graph is the same for both. The critical path of the first request is {S1, S2, N2, C2}. Because S2 has a long duration, all dependencies for N2 and C2 have been met before they start, leaving them on the critical path. The critical path of the second request is {S1, N1, C1, C2}. In this case, S2 and N2 could have longer durations and not affect end-to-end latency because C2 must wait for C1 to finish.

Typically, we ask *The Mystery Machine* to calculate critical paths for large numbers of traces and aggregate the results. For instance, we might ask how often a given segment falls on the critical path or the average percentage of the critical path represented by each segment.

4.4.2 Slack

Critical path analysis is useful for determining where to focus optimization effort; however, it does not provide any information about the importance of latency for segments off the critical path. *The Mystery Machine* provides this information via slack analysis.

We define slack to be the amount by which the duration of a segment may increase without increasing the

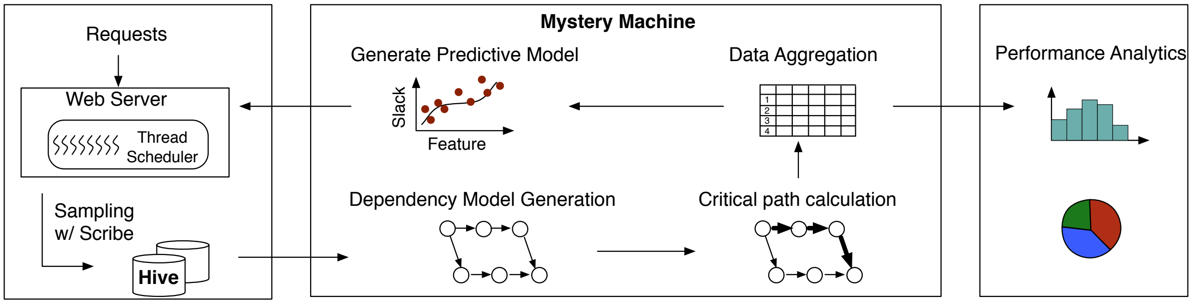


Figure 4: *The Mystery Machine* data pipeline.

end-to-end latency of the request, assuming that the duration of all other segments remains constant. By this definition, segments on the critical path have no slack because increasing their latency will increase the end-to-end latency of the request.

To calculate the slack for a given segment, S, *The Mystery Machine* calculates CP_{start}, the critical path length from the first event in the request to the start of S and CP_{end} the critical path length from the end of S to the last event in the request. Given the critical path length for the entire request (CP) and the duration of segment S (D_S), the slack for S is CP - CP_{start} - D_S - CP_{end}. *The Mystery Machine*'s slack analysis calculates and reports this value for every segment. As with critical path results, slack results are typically aggregated over a large number of traces.

4.4.3 Anomaly detection

One special form of aggregation supported by *The Mystery Machine* is anomaly analysis. To perform this analysis, it first classifies requests according to end-to-end latency to identify a set of outlier requests. Currently, outliers are defined to be requests that are in the top 5% of end-to-end latency. Then, it performs a separate aggregation of critical path or slack data for each set of requests identified by the classifiers. Finally, it performs a differential comparison to identify segments with proportionally greater representation in the outlier set of requests than in the non-outlier set. For instance, we have used this analysis to identify a set of segments that correlated with high latency requests. Inspection revealed that these segments were in fact debugging components that had been returned in response to some user requests.

4.5 Implementation

We designed *The Mystery Machine* to automatically and continuously analyze production traffic at scale over long time periods. It is implemented as a large-scale data processing pipeline, as depicted in Figure 4.

ÜberTrace continuously samples a small fraction of requests for end-to-end tracing. Trace data is collected by the Web servers handling these requests, which write them to Scribe, Facebook's distributed logging service. The trace logs are stored in tables in a large-scale data warehousing infrastructure called Hive [30]. While Scribe and Hive are the in-house analysis tools used at Facebook, their use is not fundamental to our system.

The Mystery Machine runs periodic processing jobs that read trace data from Hive and calculate or refine the causal model based on those traces. The calculation of the causal model is compute-intensive because the number of possible hypotheses is quadratic with the number of segments and because model refinement requires traces of hundreds of thousands of requests. Therefore, our implementation parallelizes this step as a Hadoop job running on a compute cluster. Infrequently occurring testing and debugging segments are automatically removed from the model; these follow a well-defined naming convention that can be detected with a single regular expression. The initial calculation of the model analyzed traces of over 1.3 million requests collected over 30 days. On a Hadoop cluster, it took less than 2 hours to derive a model from these traces.

In practice, the model must be recomputed periodically in order to detect changes in relationships. Parallelizing the computation made it feasible to recompute the model every night as a regularly-scheduled batch job.

In addition to the three types of analysis described above, *The Mystery Machine* supports on-demand user queries by exporting results to Facebook's in-house analytic tools, which can aggregate, pivot, and drill down into the results. We used these tools to categorize results by browser, connection speed, and other such dimensions; we share some of this data in Section 5.

4.6 Discussion

A key characteristic of *The Mystery Machine* is that it discovers dependencies automatically, which is critical because Facebook's request processing is constantly evolving. As described previously, *The Mystery Machine* assumes a hypothesized relationship between two segments until it finds a counterexample. Over time, new segments are added as the site evolves and new features are added. *The Mystery Machine* automatically finds the dependencies introduced by the new segments by hy-

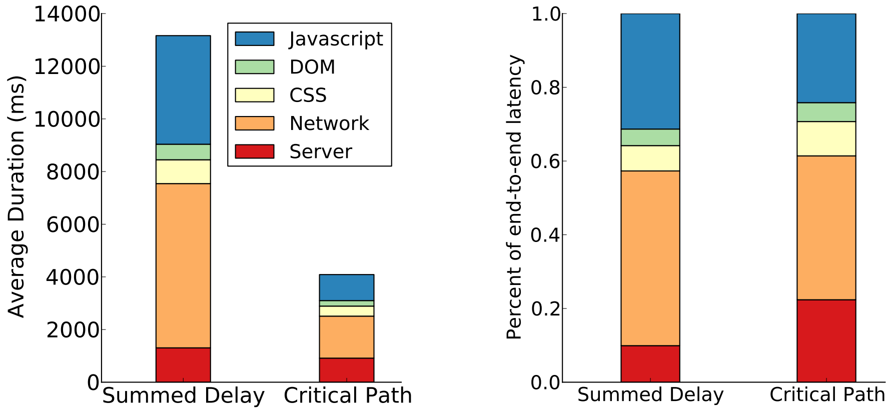


Figure 5: Mean End-to-End Performance Breakdown. Simply summing delay measured at each system component ("Summed Delay") ignores overlap and underestimates the importance of server latency relative to the actual mean critical path ("Critical Path").

pothesizing new possible relationships and removing relationships in which a counterexample is found. This is shown in Figure 3 by the increase in number of total relationships over time. To account for segments that are eliminated and invariants that are added, one can simply run a new Hadoop job to generate the model over a different time window of traces.

Excluding new segments, the rate at which new relationships are added levels off. The rate at which relationships are removed due to counterexamples also levels off. Thus, the model converges on a set of true dependencies.

The Mystery Machine relies on *ÜberTrace* for complete log messages. Log messages, however, may be missing for two reasons: the component does no logging at all for a segment of its execution or the component logs messages for some requests but not others. In the first case, *The Mystery Machine* cannot identify causal relationships involving the unlogged segment, but causal relationships among all other segments will be identified correctly. When a segment is missing, the model overestimates the concurrency in the system, which would affect the critical path/slack analysis if the true critical path includes the unlogged segment. In the second case, *The Mystery Machine* would require more traces in order to discover counterexamples. This is equivalent to changing the sampling frequency.

5 Results

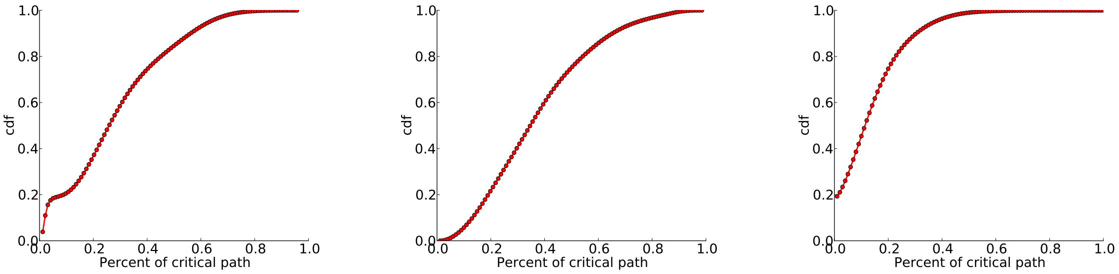
We demonstrate the utility of *The Mystery Machine* with two case studies. First, we demonstrate its use for aggregate performance characterization. We study live traffic, stratifying the data to identify factors that influence which system components contribute to the critical path. We find that the critical path can shift between three major components (servers, network, and client) and that these shifts correlate with the client type and network connection quality.

This variation suggests one possible performance optimization for Facebook servers: provide differentiated service by prioritizing service for connections where the server has no slack while deprioritizing those where network and client latency will likely dominate. Our second case study demonstrates how the natural variance across a large trace set enables testing of such performance hypotheses without expensive modifications to the system under observation. Since an implementation that provided differential services would require large-scale effort to thread through hundreds of server components, we use our dataset to first determine whether such an optimization is likely to be successful. We find that slack, as detected by *The Mystery Machine*, indeed indicates that slower server processing time minimally impacts end-to-end latency. We also find that slack tends to remain stable for a particular user across multiple Facebook sessions, so the observed slack of past connections can be used to predict the slack of the current connection.

5.1 Characterizing End-to-End Performance

In our first case study, we characterize the end-to-end performance critical path of Web accesses to the `home.php` Facebook endpoint. *The Mystery Machine* analyzes traces of over 1.3 million Web accesses collected over 30 days in July and August 2013.

Importance of critical path analysis. Figure 5 shows mean time breakdowns over the entire trace dataset. The breakdown is shown in absolute time in the left graph, and as a percent of total time on the right. We assign segments to one of five categories: *Server* for segments on a Facebook Web server or any internal service accessed from the Web server over RPC, *Network* for segments in which data traverses the network, *DOM* for



(a) Server **(b)** Network **(c)** Client

Figure 6: Cumulative distribution of the fraction of the critical path attributable to server, network, and client portions

browser segments that parse the document object model, *CSS* for segments processing cascading style sheets, and *JavaScript* for JavaScript segments. Each graph includes two bars: one showing the stacked sum of total processing time in each component ignoring all concurrency ("Summed Delay") and the other the critical path as identified by *The Mystery Machine* ("Critical Path").

On average, network delays account for the largest fraction of the critical path, but client and server processing are both significant. JavaScript execution remains a major bottleneck in current Web browsers, particularly since the JavaScript execution model admits little concurrency. The comparison of the total delay and critical path bars reveals the importance of *The Mystery Machine*—by examining only the total latency breakdown (e.g., if an engineer were profiling only one system component), one might overestimate the importance of network latency and JavaScript processing on end-to-end performance. In fact, the server and other client processing segments are frequently critical, and the overall critical path is relatively balanced across server, client, and network.

High variance in the critical path. Although analyzing the average case is instructive, it grossly oversimplifies the performance picture for the `home.php` endpoint. There are massive sources of latency variance over the population of requests, including the performance of the client device, the size of the user's friend list, the kind of network connection, server load, Memcache misses, etc. Figure 6 shows the cumulative distribution of the fraction of the critical path attributable to server, network, and client segments over all requests. The key revelation of these distributions is that the critical path shifts drastically across requests—any of the three components can dominate delay, accounting for more than half of the critical path in a non-negligible fraction of requests.

Variance is greatest in the contribution of the network to the critical path, as evidenced by the fact that its CDF has the least curvature. It is not surprising that network delays vary so greatly since the trace data set includes accesses to Facebook over all sorts of networks, from high-speed broadband to cellular networks and even some dial-up connections. Client processing always accounts for at least 20% of the critical path. After content delivery, there is a global barrier in the browser before the JavaScript engine begins running the executable components of the page, hence, JavaScript execution is a factor in performance measurement. However, the client rarely accounts for more than 40% of the critical path. It is unusual for the server to account for less than 20% of the critical path because the initial request processing before the server begins to transmit any data is always critical. Noticing this high variance in the critical path was very valuable to us because it triggered the idea of differentiated services that we explore in Section 5.2.

Stratification by connection type. We first consider stratifying by the type of network over which a user connects to Facebook's system, as it is clear one would expect network latency to differ, for example, between cable modem and wireless connections. Facebook's edge load balancing system tags each incoming request with a network type. These tags are derived from the network type recorded in the Autonomous System Number database for the Internet service provider responsible for the originating IP address. Figure 7 illustrates the critical path breakdown, in absolute time, for the four largest connection type categories. Each bar is annotated with the fraction of all requests that fall within that connection type (only a subset of connection types are shown, so the percentages do not sum to 100%).

Perhaps unsurprisingly, these coarse network type classifications correlate only loosely to the actual performance of the network connection. Mobile connections show a higher average network critical path than the other displayed connection types, but the data is otherwise inconclusive. We conclude that the network type reported by the ASN is not very helpful for making performance predictions.

Stratification by client platform. The client platform is included in the HTTP headers transmitted by the browser along with each request, and is therefore also available at the beginning of request processing. The

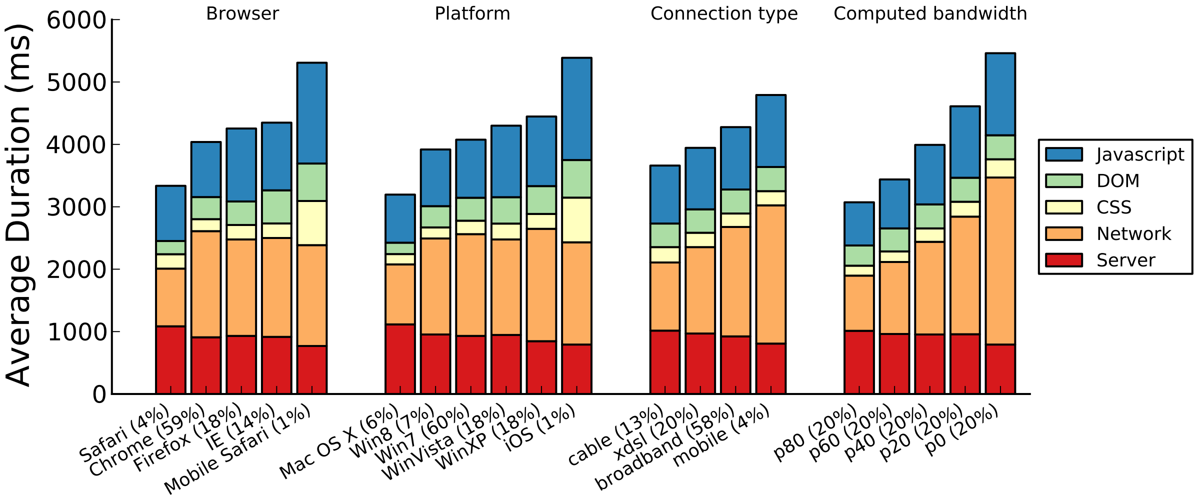


Figure 7: Critical path breakdowns stratified by browser, platform, connection type, and computed bandwidth

client operating system is a hint to the kind of client device, which in turn may suggest relative client performance. Figure 7 shows a critical path breakdown for the five most common client platforms in our traces, again annotated with the fraction of requests represented by the bar. Note that we are considering only Web browser requests, so requests initiated by Facebook cell phone apps are not included. The most striking feature of the graph is that Mac OS X users (a small minority of Facebook connections at only 7.1%) tend to connect to Facebook from faster networks than Windows users. We also see that the bulk of connecting Windows users still run Windows 7, and many installations of Windows XP remain deployed. Client processing time has improved markedly over the various generations of Windows. Nevertheless, the breakdowns are all quite similar, and we again find insufficient predictive power for differentiating service time by platform.

Stratification by browser. The browser type is also indicated in the HTTP headers transmitted with a request. In Figure 7, we see critical paths for the four most popular browsers. Safari is an outlier, but this category is strongly correlated with the Mac OS X category. Chrome appears to offer slightly better JavaScript performance than the other browsers.

Stratification by measured network bandwidth. All of the preceding stratifications only loosely correlate to performance—ASN is a poor indication of network connection quality, and browser and OS do not provide a reliable indication of client performance. We provide one more example stratification where we subdivide the population of requests into five categories directly from the measured network bandwidth, which can be deduced from our traces based on network time and bytes transmitted. Each of the categories are equally sized to represent 20% of requests, sorted by increasing bandwidth (p80 is the quintile with the highest observed bandwidth). As one would expect, network critical path is strongly correlated to measured network bandwidth. Higher bandwidth connections also tend to come from more capable clients; low-performance clients (e.g., smart phones) often connect over poor networks (3G and Edge networks).

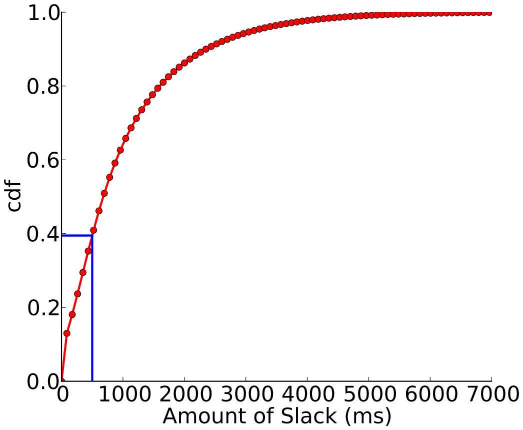


Figure 8: Slack CDF for Last Data Item. Nearly 20% of traces exhibit considerable slack—over 2 s—for the server segment that generates the last pagelet transmitted to the client. Conversely, nearly 20% of traces exhibit little (< 250 ms) slack.

5.2 Differentiated Service using Slack

Our second case study uses *The Mystery Machine* to perform early exploration of a potential performance optimization—differentiated service—without undertaking the expense of implementing the optimization.

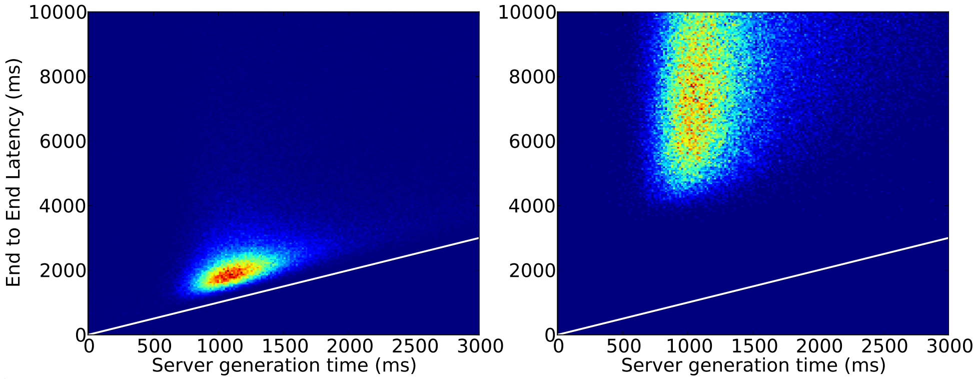


Figure 9: Server vs. End-to-end Latency. For the traces with slack below 25ms (left graph), there is strong correlation (clustering near $y = x$) between server and end-to-end latency. The correlation is much weaker (wide dispersal above $y = x$) for the traces with slack above 2.5s (right graph).

The characterization in the preceding section reveals that there is enormous variation in the relative importance of client, server, and network performance over the population of Facebook requests. For some requests, server segments form the bulk of the critical path. For these requests, any increase in server latency will result in a commensurate increase in end-to-end latency and a worse user experience. However, after the initial critical segment, many connections are limited by the speed at which data can be delivered over the network or rendered by the client. For these connections, server execution can be delayed to produce data as needed, rather than as soon as possible, without affecting the critical path or the end-to-end request latency.

We use *The Mystery Machine* to directly measure the slack in server processing time available in our trace dataset. For simplicity of explanation, we will use the generic term "slack" in this section to refer to the slack in server processing time only, excluding slack available in any other types of segments.

Figure 8 shows the cumulative distribution of slack for the last data item sent by the server to the client. The graph is annotated with a vertical line at 500 ms of slack. For the purposes of this analysis, we have selected 500 ms as a reasonable cut-off between connections for which service should be provided with best effort ($<$ 500 ms slack), and connections for which service can be deprioritized ($>$ 500 ms). However, in practice, the best cut-off will depend on the implementation mechanism used to deprioritize service. More than 60% of all connections exhibit more than 500 ms of slack, indicating substantial opportunity to defer server processing. We find that slack typically increases monotonically during server processing as data items are sent to the client during a request. Thus, we conclude that slack is best consumed equally as several segments execute, as opposed to consuming all slack at the start or end of processing.

Validating Slack Estimates It is difficult to directly validate *The Mystery Machine*'s slack estimates, as we can only compute slack once a request has been fully processed. Hence, we cannot retrospectively delay server segments to consume the slack and confirm that the end-to-end latency is unchanged. Such an experiment is difficult even under highly controlled circumstances, since it would require precisely reproducing the conditions of a request over and over while selectively delaying only a few server segments.

Instead, we turn again to the vastness of our trace data set and the natural variance therein to confirm that slack estimates hold predictive power. Intuitively, small slack implies that server latency is strongly correlated to end-to-end latency; indeed, with a slack of zero we expect any increase in server latency to delay end-to-end latency by the same amount. Conversely, when slack is large, we expect little correlation between server latency and end-to-end latency; increases in server latency are largely hidden by other concurrent delays. We validate our notion of slack by directly measuring the correlation of server and end-to-end latency.

Figure 9 provides an intuitive view of the relationship for which we are testing. Each graph is a heat map of server generation time vs. end-to-end latency. The left graph includes only requests with the lowest measured slack, below 25 ms. There are slightly over 115,000 such requests in this data set. For these requests, we expect a strong correlation between server time and end-to-end time. We find that this subset of requests is tightly clustered just above the $y = x$ (indicated by the line in the figure), indicating a strong correlation. The right figure includes roughly 100,000 requests with the greatest slack (above 2500 ms). For these, we expect no particular relationship between server time and end-to-end time (except that end-to-end time must be at least as large as slack, since this is an invariant of request processing).

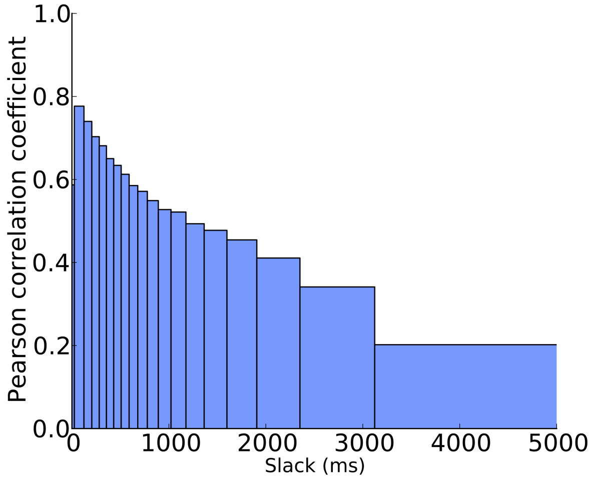


Figure 10: Server–End-to-end Latency Correlation vs. Slack. As reported slack increases, the correlation between total server processing time and end-to-end latency weakens, since a growing fraction of server segments are non-critical.

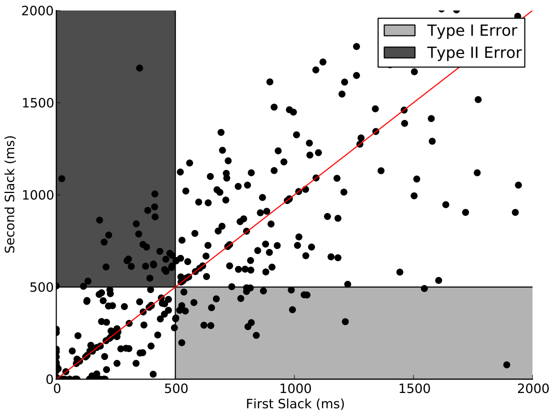


Figure 11: Historical Slack as Classifier. The clustering around the line $y = x$ shows that slack is relatively stable over time. The history-based classifier is correct 83% of the time. A type I error is a false positive, reporting slack as available when it is not. A type II error is a false negative.

Indeed, we find the requests dispersed in a large cloud above $y = x$, with no correlation visually apparent.

We provide a more rigorous validation of the slack estimate in Figure 10. Here, we show the correlation coefficient between server time and end-to-end time for equally sized buckets of requests sorted by increasing slack. Each block in the graph corresponds to 5% of our sample, or roughly 57,000 requests (buckets are not equally spaced since the slack distribution is heavy-tailed). As expected, the correlation coefficient between server and end-to-end latency is quite high, nearly 0.8, when slack is low. It drops to 0.2 for the requests with the largest slack.

Predicting Slack. We have found that slack is predictive of the degree to which server latency impacts end-to-end latency. However, *The Mystery Machine* can discover slack only through a retrospective analysis. To be useful in a deployed system, we must predict the availability or lack of slack for a particular connection as server processing begins.

One mechanism to predict slack is to recall the slack a particular user experienced in a prior connection to Facebook. Previous slack was found to be more useful in predicting future slack than any other feature we studied. Most users connect to Facebook using the same device and over the same network connection repeatedly. Hence, their client and network performance are likely to remain stable over time. The user id is included as part of the request, and slack could be easily associated with the user id via a persistent cookie or by storing the most recent slack estimate in Memcache [20].

We test the hypothesis that slack remains stable over time by finding all instances within our trace dataset where we have multiple requests associated with the same user id. Since the request sampling rate is exceedingly low, and the active user population is so large, selecting the same user for tracing more than once is a relatively rare event. Nevertheless, again because of the massive volume of traces collected over the course of 30 days of sampling, we have traced more than 1000 repeat users. We test a simple classifier that predicts a user will experience a slack greater than 500 ms if the slack on their most recent preceding connection was also greater than 500 ms. Figure 11 illustrates the result. The graph shows a scatter plot of the first slack and second slack in each pair; the line at $y = x$ indicates slack was identical between the two connections. Our simple history-based classifier predicts the presence or absence of slack correctly 83% of the time. The shaded regions of the graph indicate cases where we have misclassified a connection. A type I error indicates a prediction that there is slack available for a connection when in fact server performance turns out to be critical–8% of requests fall in this category. Conversely, a type II error indicates a prediction that a connection will not have slack when in fact it does, and represents a missed opportunity to throttle service—9% of requests fall in this category.

Note that achieving these results does not require frequent sampling. The repeated accesses we study are often several weeks separated in time, and, of course, it is likely that there have been many intervening unsampled requests by the same user. Sampling each user once every few weeks would therefore be sufficient.

Potential Impact. We have shown that a potential performance optimization would be to offer differentiated service based on the predicted amount of slack available per connection. Deciding which connections to service is equivalent to real-time scheduling with deadlines.

By using predicted slack as a scheduling deadline, we can improve average response time in a manner similar to the earliest deadline first real-time scheduling algorithm. Connections with considerable slack can be given a lower priority without affecting end-to-end latency. However, connections with little slack should see an improvement in end-to-end latency because they are given scheduling priority. Therefore, average latency should improve. We have also shown that prior slack values are a good predictor of future slack. When new connections are received, historical values can be retrieved and used in scheduling decisions. Since calculating slack is much less complex than servicing the actual Facebook request, it should be feasible to recalculate the slack for each user approximately once per month.

6 Related Work

Critical path analysis is an intuitive technique for understanding the performance of systems with concurrent activity. It has been applied in a wide variety of areas such as processor design [26], distributed systems [5], and Internet and mobile applications [22, 32].

Deriving the critical path requires knowing causal dependencies between components throughout the entire end-to-end system. A model of causal dependencies can be derived from comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [1, 3, 9, 13, 15, 18, 22, 24, 28]. In contrast to these prior systems, *The Mystery Machine* is targeted at environments where adding comprehensive new instrumentation to an existing system would be too time-consuming due to heterogeneity (e.g., at Facebook, there a great number of scheduling, communication, and synchronization schemes used during end-to-end request processing) and deployment feasibility (e.g., it is not feasible to add new instrumentation to client machines or third-party Web browser code). Instead, *The Mystery Machine* extracts a causal model from already-existing log messages, relying only a minimal schema for such messages.

Sherlock [4] also uses a "big data" approach to build a causal model. However, it relies on detailed packet traces, not log messages. Packet traces would not serve our purpose: it is infeasible to collect them on user clients, and they reveal nothing about the interaction of software components that run on the same computer (e.g., JavaScript), which is a major focus of our work. Observing a packet sent between A and B inherently implies some causal relationship, while *The Mystery Machine* must infer such relationships by observing if the order of log messages from A and B obey a hypothesized invariant. Hence, Sherlock's algorithm is fundamentally different: it reasons based on temporal locality and infers probabilistic relationships; in contrast, *The Mystery Machine* uses only message order to derive invariants (though timings are used for critical path and slack analysis).

The lprof tool [36] also analyzes log messages to reconstruct the ordering of logged events in a request. It supplements logs with static analysis to discover dependencies between log points and uses those dependencies to differentiate events among requests. Since static analysis is difficult to scale to heterogeneous production environments, *The Mystery Machine* used some manual modifications to map events to traces and leverages a large sample size and natural variation in ordering to infer causal dependencies between events in a request.

In other domains, hypothesizing likely invariants and eliminating those contradicted by observations has proven to be a successful technique. For instance, likely invariants have been used for fault localization [25] and diagnosing software errors [12, 21]. *The Mystery Machine* applies this technique to a new domain.

Many other systems have looked at the notion of critical path in Web services. WebProphet [17] infers Web object dependencies by injecting delays into the loading of Web objects to deduce the true dependencies between Web objects. *The Mystery Machine* instead leverages a large sample size and the natural variation of timings to infer the causal dependencies between segments. WProf [32] modifies the browser to learn browser page load dependencies. It also injects delays and uses a series of test pages to learn the dependencies and applies a critical path analysis. *The Mystery Machine* looks at end-to-end latency from the server to the client. It automatically deduces a dependency model by analyzing a large set of requests. Google Pagespeed Insight [14] profiles a page load and reports its best estimate of the critical path from the client's perspective. *The Mystery Machine* traces a Web request from the server through the client, enabling it to deduce the end-to-end critical path.

Chen et al. [11] analyzed end-to-end latency of a search service. They also analyzed variation along the server, network, and client components. *The Mystery Machine* analyzes end-to-end latency using critical path analysis, which allows for attributing latency to specific components and performing slack analysis.

Many other systems have looked at automatically discovering service dependencies in distributed systems by analyzing network traffic. Orion [10] passively observes network packets and relies on discovering service dependencies by correlating spikes in network delays. *The Mystery Machine* uses a minimum common content tracing infrastructure finds counterexamples to disprove causal relationship dependencies. WISE [29] answers "what-if" questions in CDN configuration. It uses machine learning techniques to derive important features that affect user response time and uses correlation to derive dependencies between these features. Butkiewicz et al. [8] measured which network and client features best predicted Web page load times across thousands of websites. They produced a predictive model from these features across a diverse set of Web pages. *The Mystery Machine* aims to characterize the end-to-end latency in a single complex Web service with a heterogeneous client base and server environment.

The technique of using logs for analysis has been applied to error diagnosis [2, 34, 33] and debugging performance issues [19, 27].

7 Conclusion

It is challenging to understand an end-to-end request in a highly-concurrent, large-scale distributed system. Analyzing performance requires a causal model of the system, which *The Mystery Machine* produces from observations of component logs. *The Mystery Machine* uses a large number of observed request traces in order to validate hypotheses about causal relationships.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Willy Zwaenepoel, for comments that improved this paper. We also thank Claudiu Gheorghe, James Ide, and Okay Zed for their help and support in understanding the Facebook infrastructure. This research was partially supported by NSF awards CNS-1017148 and CNS-1421441. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing NSF, Michigan, Facebook, or the U.S. government.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[2] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206, Big Sky, MT, October 2009.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[4] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via interface of multi-level dependencies. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, August 2007.

[5] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *Proceedings of the ACM Conference on Computer Communications (SIGCOMM)*, Stockholm, Sweden, August/September 2000.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[8] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011.

[9] Anupam Chanda, Alan L. Cox, and Willy Zwanepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, Lisboa, Portugal, March 2007.

[10] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[11] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *Proceedings of the 2013 ACM Conference on Computer Communications*, Hong Kong, China, August 2013.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), February 2001.

[13] 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 Symposium on Networked Systems Design and Implementation*, pages 271–284, Cambridge, MA, April 2007.

[14] Google. Google Pagespeed Insight. `https://developers.google.com/speed/pagespeed/`.

[15] Eric Koskinen and John Jannotti. Borderpatrol: Isolating events for precise black-box tracing. In *Proceedings of the 3rd ACM European Conference on Computer Systems*, April 2008.

[16] Adam Lazur. Building a billion user load balancer. In *Velocity Web Performance and Operations Conference*, Santa Clara, CA, June 2013.

[17] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. Webprophet: Automating performance prediction for web services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, April 2010.

[18] 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 Workshop on Hot Topics in Cloud Computing (HotCloud)*, Portland, OR, June 2011.

[19] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2012.

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, April 2013.

[21] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

[22] Lenin Ravindranath, Jitendra Padjye, Sharad Agrawal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[23] Lenin Ravindranath, Jitendra Pahye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, October 2013.

[24] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, pages 115–128, San Jose, CA, May 2006.

[25] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[26] Ali Ghassan Saidi. *Full-System Critical-Path Analysis and Performance Prediction*. PhD thesis, Department of Computer Science and Engineering, University of Michigan, 2009.

[27] 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 *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, pages 43–56, Boston, MA, March 2011.

[28] 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, Google research, 2010.

[29] Mukarram Bin Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with wise. In *Proceedings of the 2008 ACM Conference on Computer Communications*, Seattle, WA, August 2008.

[30] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – a warehousing solution over a map-reduce framework. In *35th International Conference on Very Large Data Bases (VLDB)*, Lyon, France, August 2009.

[31] 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 Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, Banff, AB, June 2005.

[32] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, April 2013.

[33] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[34] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Pittsburgh, PA, March 2010.

[35] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop compiler for PHP. *ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, October 2012.

[36] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.