Scaling causality analysis for production systems

by

Michael C. Chow

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Jason Flinn. I could not have asked for a better mentor. I would not be able to achieve this without his constant guidance and support. I would like to thank the rest of my thesis committee members: Professors Michael Cafarella, Thomas Wenisch, and Viswanath Nagarajan. Their input and advice throughout this process helped considerably. I have been particularly blessed with the opportunity to work with many great mentors: Professor Peter Chen, Mona Attariyan, David Meisner, Dan Peek, and Kaushik Veeraraghavan. I would also like to thank all of my friends throughout the years in the Computer Science department. Finally, I would like to thank my mother, my father, and my sisters, Christina, Melissa, and Stephanie, for their love and support throughout this entire process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Scaling causality analysis for production systems

by

Michael C. Chow

Chair: Jason Flinn

Causality analysis reveals how program values influence each other. It is important for debugging, optimizing, and understanding the execution of programs. This thesis scales causality analysis to production systems consisting of desktop and server applications as well as large-scale Internet services. This enables developers to employ causality analysis to debug and optimize complex, modern software systems. This thesis shows that it is possible to scale causality analysis to both fine-grained instruction level analysis and analysis of Internet scale distributed systems with thousands of discrete software components by developing and employing automated methods to observe and reason about causality.

First, we observe causality at a fine-grained instruction level by developing the first taint tracking framework to support tracking millions of input sources. We also introduce flexible taint tracking to allow for scoping different queries and dynamic filtering of inputs, outputs, and relationships.

Next, we introduce the Mystery Machine, which uses a "big data" approach to discover causal relationships between software components in a large-scale Internet service. We leverage the fact that large-scale Internet services receive a large number of requests in order to observe counterexamples to hypothesized causal relationships. Using discovered casual relationships, we identify the critical path for request execution and use the critical path analysis to explore potential scheduling optimizations.

Finally, we explore using causality to make data-quality tradeoffs in Internet services. A data-quality tradeoff is an explicit decision by a software component to return lower-fidelity data in order to improve response time or minimize resource usage. We perform a study of data-quality tradeoffs in a large-scale Internet service to show the pervasiveness of these tradeoffs. We develop DQBarge, a system that enables better data-quality tradeoffs by propagating critical information along the causal path of request processing. Our evaluation shows that DQBarge helps Internet services mitigate load spikes, improve utilization of spare resources, and implement dynamic capacity planning.

CHAPTER I

Introduction

Software systems have become increasingly complex to reason about. This is exhibited by the number of outages and bugs introduced by either errors from operators or the developers of these systems. For example, Jim Gray's classic study [38] attributes 42% of outages to system administration and 25% to software faults. Oppenheimer et al. [67] found that large-scale Internet services, which handle millions of requests a day with hundreds of software components, still suffer from these same trends. They also found that the increased complexity of these systems caused outages due to operator errors to take longer to repair. Li et. al [53] found that over 29,000 reported bugs in open source software systems were due to semantic bugs, or bugs that arose from a programmer's lack of understanding of the program design requirements. Over 80% of their reported bugs were of this nature and the reported time to diagnose and troubleshoot them was almost twice as long as other bugs. Li et. al also found that many of these bugs can lead to security and privacy vulnerabilities causing unintended consequences. Additionally, Enck et al. [34] found that in 30 Android applications, two-thirds of the applications used sensitive data suspiciously, such as sending a user's location to advertising servers without the user's knowledge.

These bugs and outages arise because complexity makes it difficult for developers, operators, and users to understand software systems, debug errors like misconfigurations, and determine how a program uses the data it reads. The complexity of software systems makes fundamentally understanding and observing the *causality* of program values difficult. Causality describes how a program value influences another according to the data flow or control flow of the program.

Observing and understanding causality is fundamental for developers and users to properly debug and optimize their programs. For example, developers have used causality for debugging programs by tracing execution through causally related events [59]. Fine-grained causality analysis at the byte-level has been used to detect privacy leaks [34, 69, 55]. It has also been applied to debugging failures and performance issues stemming from misconfigurations [8, 7]. Additionally, there are a large number of systems that rely on tracking or inferring causality in order to understand and troubleshoot large-scale distributed systems [3, 13, 23, 36, 58, 76, 99, 57]. In performance debugging, understanding causality is fundamental to deriving the critical path of execution. It has been applied in a wide variety of areas such as processor design [82], distributed systems [12], and Internet and mobile applications [76, 95]. Causality is used in performing what-if analysis and predicting the impact of software changes and optimizations [88, 25, 11].

However, due to the complexity of software systems, observing causality at a granularity that is useful for debugging and troubleshooting is challenging. At one end of the spectrum, causality can be tracked at a very fine granularity at the byte level. Fine-grained analysis, however, requires analyzing the execution at the machine instruction level. At this granularity, it requires analyzing the billions of instructions that make up complex programs, which can lead to prohibitively high runtime overheads. The high performance overhead for fine-grained analysis is unacceptable for online use in production systems. Therefore, offline analysis is needed. Tracking causality at byte-level granularity requires tracking every single byte of interest that a program reads or writes. Observing the causality between inputs and outputs at byte granularity requires new techniques that scale to tracking the millions of causal relationships between bytes as the program executes.

At the other end of the spectrum, tracing causality in large-scale systems is difficult due to the sheer size of the system and the large number of components that make up the system. Observing causality in large-scale distributed systems can be accomplished by comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [3, 7, 36, 58, 76, 78, 87]. This assumes that all components within the system are homogeneous; Dapper [87], for instance, instruments a small set of middleware components that are widely used within Google. However, many systems grow organically over time. This can result in a 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 may include client software such as Web browsers, and adding detailed instrumentation to all such software is not feasible. Thus, scalable methods for observing causality that include automated instrumentation and inference are needed for these systems.

Modern Internet services often involve hundreds of distinct software components cooperating to handle a single user request. Each component must balance the competing goals of minimizing service response time and maximizing the quality of the service provided. This leads to low-level components making data-quality tradeoffs, which we define to be explicit decisions to return lower-fidelity data in order to improve response time or minimize resource usage. The complexity of these large-scale systems makes it difficult for low-level components to make decisions without additional information. We hypothesize that propagating data along the causal path of request processing can help these systems can make better decisions that lead to improved data-quality tradeoffs.

The goal of this research is to provide methods for observing causality at useful granularities for debugging, understanding program execution, and optimizing production systems. We consider two different granularities at which causality is observed. The first is at the instruction level. Fine-grained causality analysis at the instruction level allows us to observe causality between the input bytes and output bytes of machine-level instructions. This allows us to examine byte-level relationships between program values with applications such as understanding the provenance of program data. The second is at the level of individual software components in a large-scale Internet service. This allows us to reason about large-scale Internet services from end-to-end and perform analyses such as critical path analysis. This thesis focuses on solving some of the challenges associated with scaling causality analysis to production systems with the goal of using causality to automate the burden of debugging and optimization in modern, complex software systems. Thus, the following statement summarizes this thesis:

Causality analysis requires observing how a large number of program values influence each other. It is possible to scale causality analysis to both fine-grained instruction level analysis and Internet scale distributed systems by developing automated methods to observe and reason about causality.

The first part of this thesis shows how to scale taint tracking to millions of program inputs. Taint tracking, or dynamic information flow tracking, is a fine-grained causality analysis with many practical applications, such as detecting privacy leaks [62], misconfigurations [8, 7], and understanding provenance [32]. We have designed and implemented a taint tracking framework that is able to track millions of inputs. The challenge of tracking large numbers of inputs in a taint tracking framework is keeping track of all intermediate causal dependencies; we use a data structure called the *merge log* for this purpose. We also introduce the concept of a flexible taint tracking framework with multiple *linkage functions*. Linkage functions describe different causal relationships between inputs and outputs.

In the second part of the thesis, we introduce a “big data” approach that discovers causal relationships between software components in a large-scale Internet service. Our approach, called the Mystery Machine, hypothesizes all possible casual relationships between each pair-wise software component. We leverage the fact that large-scale Internet services receive a large number of requests in order to observe counterexamples to the hypothesized causal relationships. The remaining relationships are the true casual relationships. We use this approach to discover happens-before, mutual exclusion, and pipeline relationships among the components and to construct a dependency graph among the components. Using discovered casual relationships, we identify the critical path for request execution. The critical path describes the list of components where increasing the latency of that component also increase the end-to-end latency. We perform a brief survey of how requests are broken down by critical path for Facebook request traffic. Then, we use the critical path and resulting slack analysis to explore a potential scheduling optimization.

In the last part of the thesis, we explore using causality in order to make better data-quality tradeoffs in Internet services. A *data-quality tradeoff* is an explicit decision by a software component to return lower-fidelity data in order to improve response time or minimize resource usage. First, we performed a study of data-quality tradeoffs at Facebook in order to show the pervasiveness of these decisions. The results of our study show that most data-quality tradeoffs are suboptimal in two ways. First, most data-quality tradeoffs are reactive, instead of proactive, and rely on timeouts to make tradeoffs. Reactive tradeoffs waste resources and exacerbate system overload. Second, software components make these tradeoffs with only local information because they lack higher-level knowledge such as the provenance of data, system load, and whether they are on the critical request path. Using these results from the study, we introduce a system, DQBarge, which propagates information along the causal path of processing in order to make better tradeoffs. We show how DQBarge can be used to handle load spikes, utilize spare resources, and dynamic capacity planning. By leveraging the propagation of causal information, DQBarge can make better data quality tradeoffs.

1.1 Roadmap

The remainder of the thesis proceeds as follows. Chapter II discusses how we scale the intraprocess taint tracking to millions of input bytes. Chapter III describes how we scale causality analysis in order to discover causal relationships in a large-scale Internet service. Chapter IV describes how causality can be used in order to make better data-quality tradeoffs in these Internet services. Chapter V discusses related work. Chapter VI concludes with directions for future work and summarizes the contributions of this thesis.

CHAPTER II

Scaling intraprocess taint analysis

2.1 Introduction

Dynamic information flow tracking (DIFT), sometimes called taint tracking, is a means of tracking the causality of data as a program executes. It is a technique widely used in computer security [62], privacy [34], information provenance [32], application debugging [80], and troubleshooting misconfigurations [8, 7]. DIFT instruments the program as it executes; it has been implemented in many different runtimes, such as dynamic language interpreters [72], virtual machines [34], binary instruction instrumentation [62], and hardware [29]. However, current taint tracking systems have limitations running in a production environment. First, prior systems have been limited in the number of taint labels or inputs that can be simultaneously tracked. Prior implementations have supported only up to 8 taint labels [48]. In order to perform detailed taint analysis, we would ideally support many orders of magnitude more taint labels than supported by current systems. This would allows us to track taint from millions of input bytes to millions of output bytes. Second, as the number of taint labels increase, taint tracking systems impose a large runtime overhead, making them infeasible to run in a production environment.

Our goal is to provide a scalable taint tracking framework that scales to millions of input sources. This will enable fine-grain provenance queries that allow users to derive the sources at byte granularity.

Traditionally, taint tracking has been used for online security checks in order to detect the leaking of sensitive data. Sources of sensitive data, such as password files, are tainted as they are read into the program. If a sink, such as an output to a network socket, contains tainted data, then the program may be sending out sensitive data. At runtime, this behavior can be detected and stopped. We refer to these queries, as *forward queries*. In forward queries, the sensitive sources are known ahead of time and the user wishes to know which sinks are causally affected. In other words, forward queries answer the question of what data is affected by a source.

In this thesis we focus on enabling *backward* queries. In these queries, the user is interested in what sources affected a sink. These queries are not run for online security checking, but rather, they are run for offline forensic analysis. Backward queries answer the question of what sources influenced a given output or program value.

Creating a taint tracking system to support backward queries poses several challenges that do not exist in answering forward queries. In order to answer a forward query, only the sources of interest need to be tracked. Therefore only a single bit or a predefined small set of bits are required to answer the query: was the data derived from a tainted data source or not? However, in order to answer a backward query, all unique sources of input need to be tracked as the program executes so that the specific sources can be determined at each sink. This is challenging because taint tracking now requires keeping a taint set for each memory location, and because taint sets grow to be larger as the number of unique sources increases. Resolving taint sets at runtime leads to a large amount of overhead.

This work makes the following contributions. Our taint tracking framework is the first taint tracking framework that supports tracking millions of input sources, and the *merge log* is the key to this scaling. The merge log is a data structure that enables efficient operations on taint sets with a large number of distinct sources. Second, we use deterministic replay in order to provide low runtime overhead while providing the ability to perform offline forensic taint tracking queries. Finally, we provide a *flexible* taint tracking framework that supports different propagation functions, called linkage functions, that describe different lineages of data. Our flexible taint tracking framework allows for scoping different queries by using various input and output filters. Our evaluation shows how our framework can support tracking up to tens of millions of unique sources in several desktop and server applications while providing low online overhead.

2.2 Dynamic information flow tracking background

Dynamic information flow tracking (DIFT) traces how data flows through a program by instrumenting an executing program. Taint tracking reveals which sources (inputs) causally affect which sinks (outputs) according to a propagation function. Sources of input are typically external program inputs such as bytes read from a file or network socket. Sinks are typically external outputs such as data written to a file or network socket. Taint tracking works by assigning a *taint identifier* to each unique source. As the program executes, the analysis code maintains a taint set for each location of memory. The taint set represents the sources that have affected that particular byte of memory at the point of the execution. At an output sink, taint tracking identifies the set of taint identifiers that have affected each byte that is output. Taint sets are updated for every instruction executed for the locations that the instruction affects. The propagation function determines how these taint sets are updated for each instruction. For example, using the basic data flow propagation function for an instruction that does an add, $x = y + z$, the updated taint set for x would be the union of the taint sets of y and z. Propagation functions can express a variety of causal relationships between the source and destination operands of instructions. In section 2.3, we discuss how we leverage different propagation functions in order to track various definitions of data provenance.

2.3 Flexible taint tracking

The goal of our work is to provide a taint tracking framework that allows expressive, detailed queries on the provenance of data in programs.

First, the user may only be concerned with certain outputs of a program; e.g. only program output sent to unknown destinations. Second, the user may only be concerned about data from certain inputs of a program; e.g. data from files that match a certain regex or only the data received from a network socket. Our taint tracking framework is *flexible* in the ability to allow users to scope the inputs and outputs of queries.

The user may wish to know not only what sources of data affected an output but also how the sources influenced the output. For example, the user may query what bytes of the output were directly copied from an input source. In another query, the user may ask what bytes of the output were computed from an input source. In order to accomplish this, our taint tracking framework is *flexible* in its propagation functions. It supports queries that track different types of causal relationships between data. We define several *linkage functions* that specify how outputs are influenced by inputs. We provide several common linkage functions described below. Additional linkage functions can be defined according to how an input influences its output. Our predefined linkage functions are:

- **Copy.** An input influences an output only if the output copies the value of the input (e.g., via a move instruction).

- **Data flow.** An input influences an output if the input is used to calculate the value of the output (e.g., via an add instruction).

- **Index.** An input influences an output if the input is used to calculate the output or if the input is used as an index to load a value used to calculate the output (e.g., via an array or lookup table index).

Our taint tracking framework tracks causality at a byte-level granularity. This allows the user to understand the inputs that have influenced any byte that a program outputs. Our taint tracking tool works at the machine instruction level using dynamic binary instrumentation to instrument every instruction and propagate taint according to the specified linkage function. Taint is propagated using a set of simple taint operations:

- **Set.** The set operation sets the taint label for a specified location.
- **Clear.** The clear operation clears the taint label at a given location, setting the taint label for that location to be the null taint.
- **Copy.** The copy operation sets the taint label at the destination location to be the same as the source's taint label.
- **Merge.** The merge operation takes two sets of taint labels and returns the union of the taint sets. It is used to merge dependencies.

Linkage functions describe, for a particular instruction, the set of taint operations that relate the instruction's inputs and outputs. Table 2.1 shows an example program and the subsequent taint operations for each of the supported linkage functions.

The copy linkage only tracks data copies. Therefore, for memory copy instructions, such as `mov`, we execute a corresponding copy taint operation for each byte in order to propagate the taint label for that byte. In the copy relationship, data operations such as `add` result in clear taint operations. In Table 2.1, instructions 4 and 5 show that the copy linkage clears taints, but the copy linkage maintains move and copy relationships in instruction 6. The data flow relationship propagates taint if the input

Instructions	Copy	Data	Index
1. A = read()	A: IN_0	A: IN_0	A: IN_0
2. B = read()	B: IN_1	B: IN_1	B: IN_1
3. C = read()	C: IN_2	C: IN_2	C: IN_2
4. D = A + B	D: {}	D: $\{IN_0, IN_1\}$	D: $\{IN_0, IN_1\}$
5. E = C + D	E: {}	E: $\{IN_0, IN_1, IN_2\}$	E: $\{IN_0, IN_1, IN_2\}$
6. F = C	F: $\{IN_2\}$	F: $\{IN_2\}$	F: $\{IN_2\}$
7. Y = [1, 2, 3]	Y: {}	Y: {}	Y: {}
8. Z = Y[D]	Z: {}	Z: {}	Z: $\{IN_0, IN_1\}$
9. write(Z)	OUT_0: {}	OUT_0: {}	OUT_0: $\{IN_0, IN_1\}$

Table 2.1: **Linkage function examples** Taint tracking analysis of an example program with the different linkage functions.

of an instruction is used to calculate the value of the output. In Table 2.1, for the data flow relationship, instructions 4 and 5 merge the taint sets of the instructions' inputs. The data flow analysis is a superset of the copy analysis so instruction 6 still maintains the relationships propagated in the copy analysis. The index linkage function propagates taint if the input is used to calculate an output or if the input is used as an index to load a value to calculate the output, such as via an array or lookup table index. Instruction 8 shows using an instruction's input as an index, so that relationship is propagated through to the instruction's output.

Our taint tracking tools use Pin [56] binary instrumentation to dynamically instrument a program as it executes. We chose Pin because it is a flexible and well-documented tool. We make use of Pin's dynamic binary instrumentation facility in order to dynamically insert taint propagation logic to track and propagate taint for every x86 instruction executed. Each linkage function is implemented as a separate Pin tool that inserts the proper taint operation(s) for each x86 instruction. Our taint tracking framework provides an interface of taint operations that a linkage function Pin tool should implement. Implementing a new linkage tool only necessitates implementing the functions of the interface, i.e, providing an implementation for each taint operation.

There are several challenges with implementing a taint tracking framework using dynamic binary instrumentation. In order to track causality at such a fine-granularity, all instructions of the application need to be instrumented so that the relationships between inputs and outputs can be tracked. Instrumenting all instructions of an application can lead to prohibitively high online overhead. Additionally, in order to efficiently implement these taint operations, we need to be able to compactly represent sets of taints, lookup the taint at any program location efficiently, and merge two taint sets efficiently. We describe how we implement these operations in an efficient manner in the next section.

2.4 Efficient taint representation

Every source byte that is tracked is assigned an integer taint identifier when it is read into the program. We track all sources of input to the program from external sources through the kernel by intercepting all input system calls using Pin. Additionally, environment variables and arguments passed into the program via the `execve` system call are tracked as sources of input.

We maintain a mapping of taint labels to the bytes read via input system calls. Taint labels are represented as integers. A taint label represents a unique source of input. The locations of the taint creation are initially set to their corresponding taint label. For example, the taint at the address of each byte of a `read` buffer is set to the newly created taint labels.

In order to efficiently track taint as the system propagates, we require data structures that allow us to quickly look up the taint of any location in the process's address space or CPU registers. We maintain an array of taint labels for each CPU register for each thread of execution. In order to efficiently map the process's address space, we use a two-level page table structure that shadows the process's address space. We choose to map the address space using a page table since the address spaces of most processes are sparse.

Next, we need data structures that allow for fast implementation of all of the taint operations (clear, set, copy, and merge). It is important that these operations are fast since the number of taint operations that are executed is on the same order of magnitude as the number of machine instructions executed. The clear and set taint operations are made efficient using the page table structure for memory locations. These operations are both $O(1)$ operations in the number of input bytes. Using an array for CPU registers also is an $O(1)$ operation for clears and sets on registers. However, supporting the merge taint operation requires additional data structures in order to be fast.

When there is a limited number of taints, a taint label can be represented as a bit vector or array. For example, using a bit vector representation, a 32-bit integer representation of a taint label supports 32 possible input labels, with each bit position representing a taint label. This representation makes set and merge operations efficient. An initial set operation can be accomplished by performing a bitwise shift to set the appropriate bit. A merge operation is accomplished by performing a bitwise OR operation on the taint bit vector representations. However, the drawbacks of this approach is that only a finite number of taints are representable.

Alternatively, a set of taints could be represented as a list of taint labels. However, during the program execution, in order to propagate taint, we need to have a mapping of all taint locations to set of taints. This representation is not very space efficient; each location points to a separate list. There is also high memory overhead as nodes in each list need to be allocated and deallocated as taint is added or removed from the taint set. Also, this representation means that merging taints requires merging two lists, which is a slow operation.

Thus, the `merge` operation adds an additional complication in choosing how to represent taints; we not only wish to efficiently represent all input taint labels but we also need to efficiently represent all possible sets of taints while the program executes.

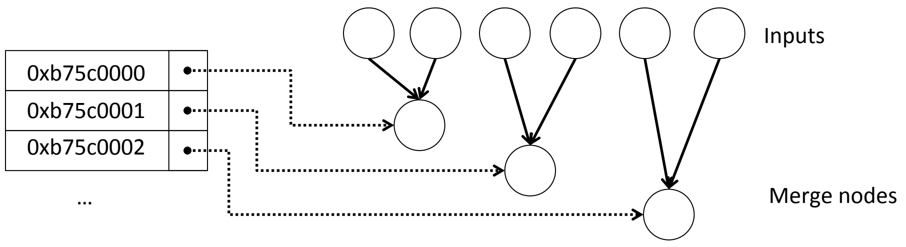


Figure 2.1: **Taint graph.** Representation of how sets of taint labels are represented as a directed acyclic graph. A pointer to a node in the graph only needs to be stored in order to represent a taint set. In order to resolve what input taint labels are in the taint set, we walk upwards from the pointer until all root nodes have been resolved.

Our insight is that taint sets only need to be resolved at certain outputs. Only at these outputs do we have to determine the set of input taint labels that have influenced the output. Thus, we simply need to propagate enough information so that the taint set can be calculated when needed.

In order to efficiently support the `merge` operation, we use an idea proposed by Ruwase et al. [80] in which sets of taint values are represented by a binary tree. This data structure takes advantage of the observation that all possible taint states must be derived from an input taint label. For example, in Table 2.2, instruction 4 requires merging the taint sets for the inputs A and B. The taint set $\{IN_0, IN_1\}$, is the result of the merge of the input taint sets. Each merge set can be represented as a pointer to its two parent taint sets. Thus, all merge states can be encoded in a directed, acyclic graph where root nodes are inputs. Intermediate nodes represent taint labels resulting from merge operations. In order to resolve the inputs that a particular output depends on, we can perform a depth-first traversal of the root node entry to discover the entire input set. Thus, instead of storing the taint set for every memory location, all possible taint sets are encoded in the merge log. Then, instead of storing the entire set of taints, we store a pointer into the merge log that represents the set of taints that have influenced that location. A logical representation of this is shown in Figure 2.1.

Instructions	Data	Merge log
1. A = read()	A: IN_0	
2. B = read()	B: IN_1	
3. C = read()	C: IN_2	
4. D = A + B	D: $M[0]$	$M[0] = \{IN_0, IN_1\}$
5. E = C + D	E: $M[1]$	$M[1] = \{IN_2, M[0]\}$
6. F = C	F: IN_2	
7. Y = [1, 2, 3]	Y: $\emptyset$	
8. Z = Y[D]	Z: $\emptyset$	
9. write(Z)	OUT_0: {}	

Table 2.2: **Merge log example** An example of how the merge log tracks and resolves the union of taint sets in a data flow analysis.

Each entry in the merge log represents a binary tree of taint identifiers rooted at that node. The merge log itself is an append-only log with each new entry representing a new merge operation. The merge log leverages the property that the resulting taint set of a merge operation is the union of two existing taint sets. Each entry in the merge log represents the result of a merge operation. It contains two pointers to the previous taint sets. A representation of the merge log is show in Figure 2.2. Table 2.2 walks through an example program showing the contents of the merge log as the program executes a data flow analysis. In order to make the merge log efficient, merge operations are restricted to be binary operations with two inputs. For merges that require more than two inputs, these merges are broken up into multiple pairwise merge operations.

Our design differs from Ruwase et al.'s algorithm in that Ruwase et al. applied the merge log to abstract taint values. They stored a symbolic representation of where the taint for a particular location was derived from. Their design computed the concrete taint value by resolving the symbolic representation for different tainted inputs. The symbolic representation was applied to parallelize sequential taint tracking. Our enhancements to the merge log allows our taint tracking framework to scale to millions of dependencies. Our merge log keeps track of concrete taint values and defers resolving the union of taint sets. The merge log uses much less memory than

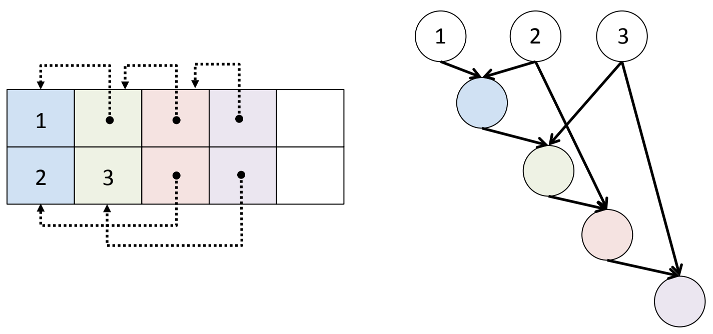


Figure 2.2: **Merge log.** The left diagram shows the merge log. Each entry contains the two pointers to the taint sets that were merged. Each entry in the merge log refers to either a previous entry in the merge log or an input taint label. The right diagram shows the directed acyclic graph of merge operations that the merge log encodes.

storing a set for each location. Memory usage is proportional to the number of merge operations rather than the total size of all taint sets for every location. The cost of using a merge log is that a tree traversal must be performed when resolving a root node to a set of source identifiers.

There are, however, some tradeoffs in this design. By encoding taint sets as a graph, the same taint set can exist in the graph if the taint set were created using a different ordering of merge operations. This can lead to duplicate nodes in the graph. We found that, in practice, this occurs rarely due to the nature of data flow analysis. Since every taint set is encoded in this graph, in order to resolve the input taint labels, the graph needs to be walked upwards for every single output byte. We can amortize the cost of walking the graph by memoizing the resolution of intermediate merge nodes.

Several optimizations are made in the merge operation. Merges with the null taint set return the non-null taint set and do not create an entry in the merge log. Merges between the same taint set (i.e., represented by the same pointer) return the same taint set without creating a new merge log entry. Since taint sets are represented by a pointer in the merge log implementation, the two merge operands are first ordered before merging. The two pointers are then non-associatively hashed, for performance, and stored in a hash table with the resulting merged taint set. Therefore, repeated merges of the same taint sets do not create a new merge log entry.

Another added benefit of using the merge log is the deferral of resolving taints to an offline phase. For each output of interest, only the pointer into the merge log needs to be saved in order to resolve it. Since the merge log is append-only, at the end of execution all of the information to resolve a taint identifier in the merge log still exists. This allows us to reduce the amount of instrumentation that needs to be added at runtime, improving performance.

2.5 Interplay of deterministic replay with taint tracking

Another challenge with dynamic taint tracking is the large runtime overhead associated with tracking a large number of taint inputs. Deterministic replay is a tried and tested technique of faithfully reproducing an execution of a program. Deterministic replay can be implemented at several abstraction layers such as the virtual machine monitor [33], the operating system kernel [92], or a user-level library [71]. We wish that our deterministic replay system provide a low runtime overhead while allowing us to dynamically analyze the running execution. For this reason, we chose to use an operating system level deterministic replay system [92].

We needed to make modifications to the replay implementation because of our desire to use Pin to insert binary instrumentation into replayed executions. This implementation faces a substantial challenge: from the point of view of the replay system, the replayed execution is not the same as the recorded execution because it contains additional binary instrumentation not present during recording. While Pin is transparent to the application being instrumented, it is *not* transparent to lower layers such as the OS.

Our replay system is *instrumentation-aware*; it compensates for the divergences in replayed execution caused by dynamic instrumentation. Pin makes many system calls, so our replay system allocates a memory area that allows analysis tools run by Pin to inform the replay kernel which system calls are initiated by the application (and should be replayed from the log) and which are initiated by Pin or the analysis tool (and should execute normally).

It also compensates for interference between resources requested by the recorded application and resources requested by Pin or an analysis tool. For instance, Pin might request that the kernel `mmap` a free region of memory. If the kernel grants Pin an arbitrary region, it might later be unable to reproduce the effects of a recorded application `mmap` that returns the same region. Our replay system avoids this trap by initially scanning the replay log to identify all regions that will be requested by the recorded application and pre-allocating them so that Pin does not ask for them and the kernel does not return them.

The replay system avoid avoids conflicts for signal handlers in a similar manner. Since Pin allows for the ability to perform an arbitrary action on receipt of a signal, it must intercept all signal delivered to the application process first. This is at odds with the replay system, which deterministically replays the delivery of signals to the process. Our replay system avoid conflicts for signal handlers by allowing Pin to register its signal handlers and signal masks during replay. Then, all signals, including replayed signals, are delivered to Pin.

Finally, the replay system must avoid deadlock. The replay system adds synchronization to reproduce the same order of system calls, synchronization events, and racing instructions seen during recording. Pin adds synchronization to ensure that application operations such as memory allocation are executed atomically with the Pin code that monitors those events. Our replay system initially deadlocked because it was unaware of Pin locking. To compensate, it now only blocks threads when it knows Pin is not holding a lock; e.g., rather than block threads executing a system call, it blocks them prior to the instruction that follows the system call.

2.6 Evaluation

In this section we evaluate how effective the linkage tools are at achieving our goals. In particular, we will answer the questions:

- What is the runtime overhead of the linkage tools?
- How much additional work is required to go from forward dynamic taint analysis to backward dynamic taint analysis?

We evaluated on taint tracking framework on a 3.1GHz Xeon server with 8GB of RAM. We used a deterministic replay system based on the 32-bit Linux kernel 3.5.7.13. Since our deterministic replay system schedules threads to execute on a single core when replaying execution, our analysis uses only a single core.

2.6.1 Runtime overhead

First, we will look at the runtime overhead of the tools. Since all of the analysis is performed offline, the runtime overhead is comprised of recording a process's execution.

We measured online overhead by comparing the throughput and latency of Apache, lighttpd, postfix, and Postgre when they are recorded by our deterministic replay system to results when the applications run on default Linux without recording. For Apache and lighttpd, we used ab to send 5000 requests for a 35KB static Web page with a concurrency of 50 requests at a time over an isolated network. For Postfix, we used smtp-source to send 1000 64KB mail messages. For PostgreSQL, we used pgbench to measure the number of transactions completed in 60 seconds with a concurrency of 10 transactions at a time. Each transaction has one `SELECT`, three `UPDATE`s, and one `INSERT` command.

Our system adds an average of 2.3% throughput overhead: 0.1% for Apache, 4.7% for Postfix, 3.5% for PostgreSQL, and 0.8% for lighttpd. These values include the cost of logging data races previously detected by our offline data race detector. This overhead is consistent with similar deterministic replay approaches [28]. Latency overheads for Apache, PostgreSQL, and lighttpd are equivalent to the respective throughput overheads; Postfix has no meaningful latency measure since its processing is asynchronous. The recording log sizes were 2.8MB for Apache, 1.6MB for lighttpd, 321MB for PostgreSQL, and 15MB for Postfix. Apache and lighttpd have smaller logs because they use `sendfile` to avoid copying data.

2.6.2 Scaling the linkage tools

Next, we look at the performance of the copy, data flow, and index linkage tools in analyzing the input and output relationships in a set of programs and how they scale to an increasing size of inputs and number of dependencies. We evaluated our tools on 7 commonly used desktop programs and server workloads:

- **Gzip** – Zip a large file
- **Ghostscript** – Convert a research poster .ps to .pdf
- **Evince** – Open and view a research paper
- **Mongodb** – Yahoo cloud server benchmark
- **Nginx** – Serve static content
- **OpenOffice** – Edit a conference presentation
- **Firefox** – A long Facebook browsing section

We show the time to replay each benchmark without instrumentation as a baseline. Note that replay time is not equivalent to the original application execution time; instead, replay time can be one to two orders of magnitude faster [32] because deterministic replay omits all user think-time (for interactive applications), network delays and idle time (for servers), and external output.

For Gzip, Evince, Mongodb, Nginx, all of the linkage queries ask for all dependencies between command-line, network, and file-system inputs to all such outputs.

Running the all-to-all query for Ghostscript, OpenOffice, and Firefox produces a large amount of data, so the queries were refined to produce manageable results. The Ghostscript query only considers file system data from the input file, the OpenOffice query only considers file system data from a portion of the input file, and the Firefox query only considers cookies data to be sources and network output to specific sites to be outputs.

First, we implemented a forward taint tracking tool to establish a baseline for evaluating the performance of our tools. It uses a single bit to track tainted or untainted data and works with our deterministic replay system. We compare the performance of our forward taint tracking implementation with an existing taint tracking implementation, libdft [48]. We used the baseline libdft tool that uses a single bit to track tainted or untainted data. It tracks all input from the file system, except for shared libraries, and all incoming network data. We ran the base libdft implementation 5 times on our computational benchmarks, Gzip and Ghostscript. As noted previously, running libdft on our interactive and server benchmarks without deterministic replay, on live execution, does not provide a sound comparison as replay time is not equivalent to application execution time. For Gzip, the libdft analysis took on average 18.71 seconds with a standard deviation of 0.15. For Ghostscript, the libdft analysis took on average 19.90 seconds with a standard deviation of 0.30. This is 8.6 times longer than normal execution for Gzip and 46.3 times longer for Ghostscript. In comparison, our forward taint tracking tool took 1.9 times longer than libdft for both Gzip and Ghostscript. Unlike libdft, our forward taint tracking analysis is done offline using deterministic replay.

Our forward taint tracking tool is slower than libdft for two main reasons. First, libdft has a more efficient forward taint-tracking implementation and has made optimizations for faster Pin instrumentation. Libdft has optimized its taint tracking analysis code to avoid branches and keeps its analysis code short so that Pin can

Benchmark	Execution time (s)	libdft (s)	Replay + null Pin (s)	Replay + forward (s)
gzip	2.18 ± 0.03	18.71 ± 0.15	9.23 ± 0.07	35.11 ± 0.23
ghostscript	0.43 ± 0.01	19.90 ± 0.3	19.60 ± 0.38	38.47 ± 0.98

Table 2.3: **Comparison with libdft** This table show the evaluation of an existing taint tracking framework, libdft, compared with our forward taint tracking implementation. Each result in the average analysis time taken over 5 runs with the standard deviation.

inline it. Second, as previously stated, our forward taint tracking tool has been modified to work with our deterministic replay system. The required modifications are described in section 2.5. These modifications require that Pin insert additional instrumentation to support our deterministic replay system; this adds overhead compared with running libdft on live execution. To measure this overhead, we implemented a null Pin tool. The null Pin tool adds the minimum instrumentation to support our deterministic replay system; it is the lower bound on any Pin tool run with our deterministic replay system. Over 5 runs, the null Pin tool took on average 9.23 seconds with a standard deviation of 0.07 for Gzip. For Ghostscript, it took an average of 19.60 seconds with a standard deviation of 0.38. Adding our forward taint tracking instrumentation increases the analysis time by 3.80 times (to 35.11 seconds) for Gzip and 1.96 times (to 38.47 seconds) for Ghostscript. Table 2.3 provides a summary of the overhead of libdft on live execution compared with the replay time of Gzip and Ghostscript with the null Pin tool and our forward taint tracking tool.

Next, we compare the additional analysis time of our linkage tools compared with our baseline forward taint tracking tool. Table 2.4 shows the results for each workload evaluated under each linkage tool. For each benchmark, we show the number of input bytes considered in each query and the number of input-output dependencies each query produces. The replay time is the time to re-execute the program without any analysis.

Our linkage tools, on average across all 7 benchmarks, track the relationships between 15,330,432.3 unique inputs to 48,338,413.3 unique outputs. Our tools were able to track over 64 million unique inputs for Gzip and all but one benchmark had millions of unique inputs. This is many orders of magnitude greater than prior systems. Across all benchmarks, the copy, data flow, and index linkage tools took only 37.6%, 44.1%, and 182.3% longer than the forward tool, respectively. Additionally, we show the flexibility of our taint tracking system. Our framework allows us to track different relationships using the copy, data, and index linkages without drastically changing the analysis time of each tool. The data flow tool took, on average, 9.6% longer than the copy tool and the index tool took, on average, 68.1% longer than the data flow tool.

The index linkage produces the longest queries as the index linkage function tracks a significantly larger number of dependencies. The longest index linkage query was on Firefox with a query runtime of approximately 82 minutes. However, as mentioned previously, these queries are performed offline using deterministic replay. Since more complicated linkage functions produce a greater number of dependencies, we envision users employing these queries on targeted inputs.

2.6.3 Scalability of the merge log and limitations

Next we discuss the scalability of the merge log. Table 2.5 shows the merge log size for each of the previous benchmarks. The copy linkage does not produce a merge log since it performs no taint merge operations. The largest merge logs were with the index linkage analyzing mongo and Firefox with 1.7GB and 1.3GB merge log sizes, respectively. In all of these queries the merge log fits in the 32-bit process address space. The current implementation keeps the merge log in memory for the duration of the analysis. The Ghostscript, OpenOffice, and Firefox queries only considered a subset of inputs since running the all-to-all queries would produce a merge log that exceeded the process address space.

The merge log scales taint tracking to millions of inputs by enabling a merge operation with an online cost comparable to a bitmask merge operation used by prior systems. However, using the merge log results in an extra step of computation where the outputs need to be resolved to inputs by traversing the graph encoded in the merge log. Our system can become bottlenecked whenever traversing this graph becomes prohibitively expensive since resolving every byte of output requires performing a traversal of the graph.

One case where the traversal becomes prohibitively expensive is when the merge log no longer fits in the process address space. Since the merge log is an append-only data structure, segments of the merge log can be stored once the memory limit has been reached and a new merge log segment can be allocated. However, a traversal of the merge log can now span multiple segments that do not fit in memory. Therefore, a traversal of the graph could require loading and unloading segments of the merge log into memory. This would lead to a substantial slowdown.

A potential solution for solving this bottleneck would be to move the processing of the merge log to a machine with a 64-bit address space so that the entire merge log can fit in memory. Since the merge log is a directed acyclic graph, we can also leverage existing graph databases that are optimized for large graphs and high query throughput [79, 66]. Using these systems, we could potentially scale the processing of the merge log to a large cluster. This would scale both the time to resolve a single byte of output and allow for an increased parallelization of resolving outputs.

Another case where traversals can become expensive is if an output byte depends on a large number of input bytes, so a traversal requires visiting many intermediate nodes. Applications that perform operations on binary data often result in a large number of data operations to track the propagation of taint. This results in a pattern where a large number of input bytes have a relationship with a large number of output bytes. For example, the encrypting of data can lead to such behavior. This taint explosion is common with taint tracking. Previous systems have mitigated taint explosion by using taint abstraction [8]. Common well-defined library functions are abstracted away using manual annotations describing how the inputs to the functions affect the output. Therefore using taint abstraction techniques that are aware of the binary formats can significantly reduce the analysis runtime (by eliding taint propagation for these functions) and reduce the merge log size.

Another bottleneck of the merge log is that it is produced sequentially. Taint tracking, by definition, is sequential since the analysis examines dependencies as a program executes. The dependencies are recorded in the merge log as the program executes. Because of this, the analysis is bound by the execution time of the program. Even though we defer the analysis offline using deterministic replay, the analysis requires the entire program to be re-executed in order to track dependencies between inputs and outputs. For long running applications, such as a web server serving traffic for days, this would require performing the analysis for the duration of the program's execution. Parallelizing the taint analysis so that it can be completed faster is non-trivial and we leave this as future work [75].

2.7 Conclusion

In this chapter of this thesis, we have proposed a set of linkage tools used for understanding the relationships between inputs and outputs of a process. These linkage tools make use of fine-grained instruction-level taint tracking that scales to a large set of inputs. This enables determining of fine-grained, byte-level provenance of the output of a process. We enable the use of such analysis by modifying a deterministic replay system to be instrumentation-aware. We show that this allows for low online runtime overhead by performing the analysis offline. Finally, we show the effectiveness of our tools by analyzing a set of commonly used programs.

Benchmark	Replay Time (s)	Input bytes	Output bytes	Linkage	Runtime (s)	Dependencies
gzip	2.07	64,352,941	48,791,393	Forward	35.11 ± 0.23	36,586,765
				Copy	66.65 ± 0.18	36,586,765
				Data	67.71 ± 0.22	36,586,765
				Index	74.60 ± 0.54	36,586,765
ghostscript	0.67	2,514,067	176,009	Forward	38.47 ± 0.98	14,682,254
				Copy	54.82 ± 1.05	100,363
				Data	56.34 ± 1.23	14,682,254
				Index	190.35 ± 0.82	685,205,917
evince	7.81	10,302,848	104,061,604	Forward	124.28 ± 1.76	895,684
				Copy	143.80 ± 1.43	144,670
				Data	153.28 ± 3.93	895,684
				Index	395.14 ± 3.65	70,842,186
nginx	2.88	10,412,627	35,000,000	Forward	83.01 ± 2.50	5,000,000
				Copy	112.61 ± 0.97	5,000,000
				Data	116.71 ± 1.03	5,000,000
				Index	120.44 ± 0.41	5,000,000
mongo	15.63	8,863,855	116,592,809	Forward	133.68 ± 2.93	76,042,962
				Copy	190.06 ± 0.74	76,042,962
				Data	201.69 ± 3.16	76,042,962
				Index	266.22 ± 0.47	76,538,822
openoffice	5.17	9,946,659	32,110,959	Forward	259.61 ± 3.5	1,764,494
				Copy	282.02 ± 0.57	1,764,446
				Data	292.56 ± 2.07	1,764,494
				Index	294.88 ± 4.85	1,764,504
firefox	45.53	920,029	1,636,119	Forward	1002.85 ± 16.7	131,476
				Copy	1295.23 ± 9.34	131,476
				Data	1337.74 ± 9.09	131,476
				Index	4945.06 ± 24.69	7,137,269

Table 2.4: **Linkage tool evaluation** This tables show the evaluation of our taint tracking framework with our predefined linkage tools on several desktop and server applications. It shows the number of unique inputs, output, and dependencies each linkage tool tracked as well as the duration of the analysis time for each linkage function. Each of the analysis time shown is the average analysis time taken over 5 runs with the standard deviation.

Benchmark	Linkage tool	Merge log size (bytes)
gzip	Data	1,944
	Index	127,844,552
ghostscript	Data	13,978,144
	Index	51,765,000
evince	Data	6,853,304
	Index	18,693,064
nginx	Data	16,012,056
	Index	16,012,416
mongo	Data	299,222,288
	Index	1,715,635,776
openoffice	Data	5,931,368
	Index	5,925,264
firefox	Data	39,469,248
	Index	1,343,744,416

Table 2.5: **Merge log size** This table shows the size of the merge log structure at the end of the benchmark execution.

CHAPTER III

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

In this part of the thesis, 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.

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.

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.

3.1 Background

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.

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.

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 [3, 7, 36, 58, 76, 78, 87]. This is a reasonable assumption if the software architecture is homogeneous; for instance, Dapper [87] 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 [13, 91]. 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.

3.2 Life of a Facebook request

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) [52]. 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 [98]. Request execution on the Web server triggers many RPCs to caching layers that include Memcache [64] and TAO [16]. 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.

3.3 *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 [77] 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.

3.4 *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.

3.4.1 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

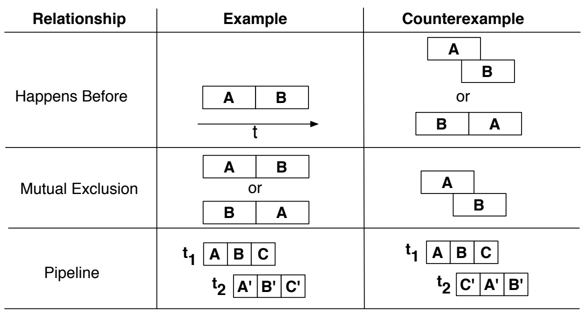


Figure 3.1: **Causal Relationships.** This figure depicts examples of the three kinds of causal relationship we consider. Happens-before relationships are when one segment (A) always finishes in its entirety before another segment (B) begins. FIFO relationships exist when a sequence of segments each have a happens-before relationship with another sequence in the same order. A mutual exclusion relationship exists when two segments never overlap.

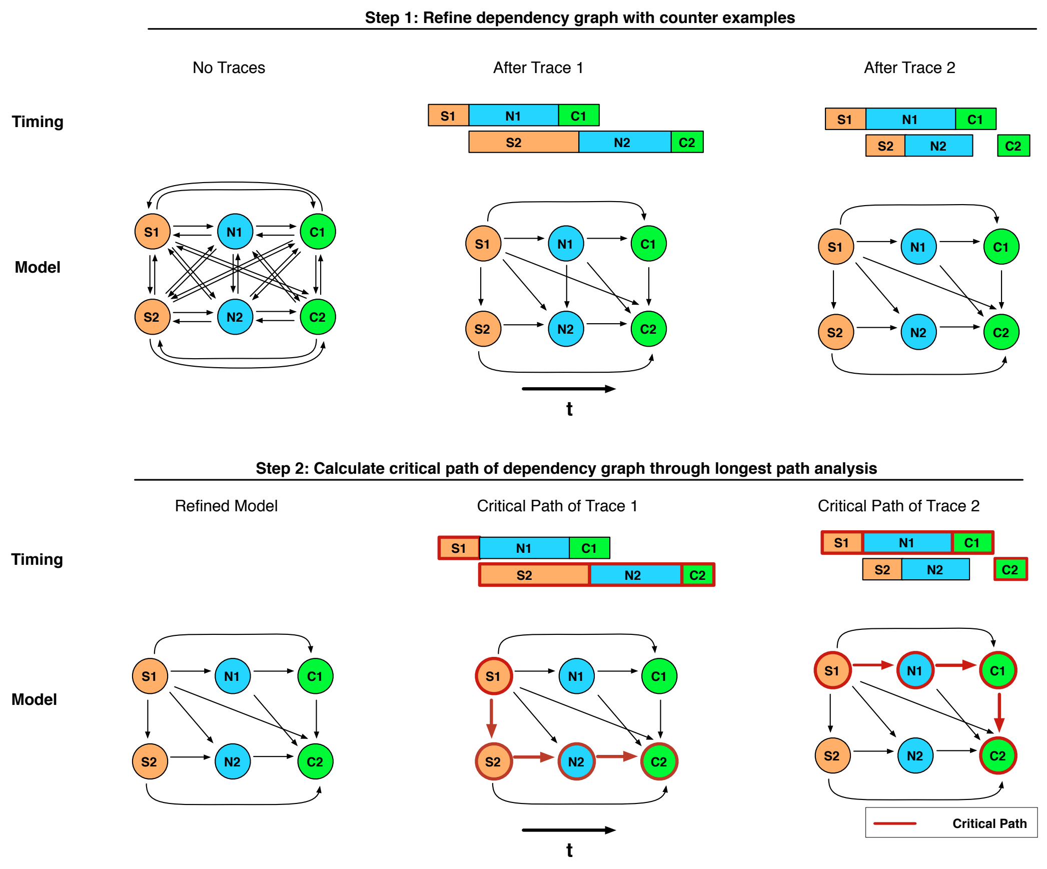
event timestamp for A in all requests.

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 3.1. For each relationship we provide a valid example and at least one counterexample that would contradict the hypothesis.

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 re-



Step 1: Refine dependency graph with counter examples
No Traces
After Trace 1
After Trace 2
Timing
Model
S1
N1
C1
S2
N2
C2
t
Step 2: Calculate critical path of dependency graph through longest path analysis
Refined Model
Critical Path of Trace 1
Critical Path of Trace 2
Critical Path

Figure 3.2: **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.

lationship: either A $\rightarrow$ B or B $\rightarrow$ 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.

3.4.2 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 3.2 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 $\rightarrow$ S1, is removed. Further traces must be processed to complete the model. For instance, the second request eliminates the hypothesized relationship, N1 $\rightarrow$ 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 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 3.3 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.

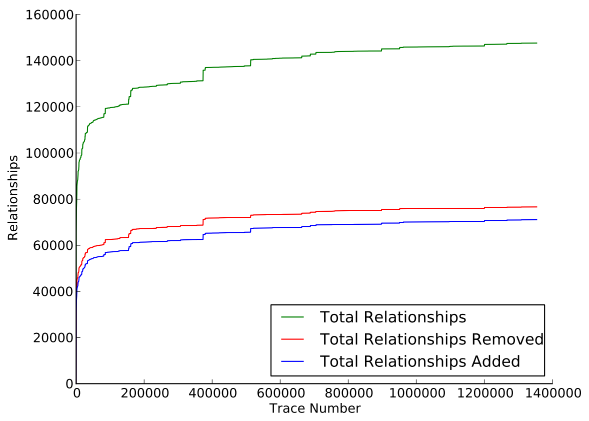


Figure 3.3: **Hypothesis Refinement.** This graph shows the growth of number of hypothesized relationships as a function of requests analyzed. As more requests are analyzed, the rate at which new relationships are discovered and removed decreases and eventually reaches a steady-state. The total number of relationships increases over time due to code changes and the addition of new features.

3.4.3 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.

3.4.4 Analysis

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

3.4.5 Critical Path

Critical path analysis is a classic technique for understanding how individual components of a parallel execution impact end-to-end latency [76, 95]. 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 $\rightarrow$ C are recursively removed if there exists a path consisting of A $\rightarrow$ B and B $\rightarrow$ 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 3.2. 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.

3.4.6 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 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.

3.4.7 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

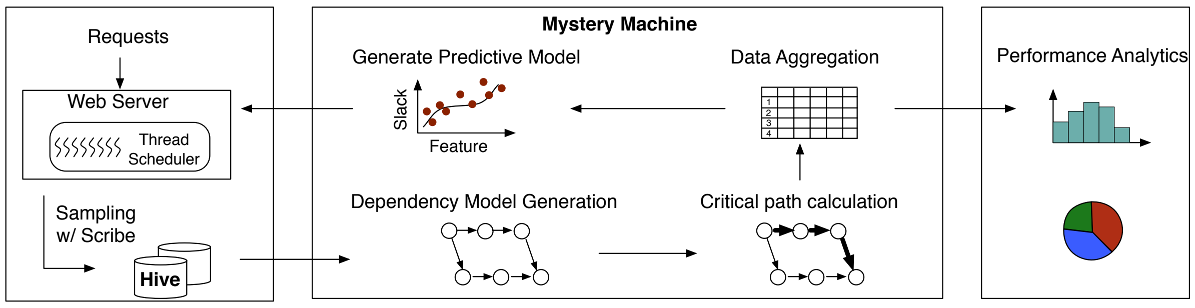


Figure 3.4: ***The Mystery Machine* data pipeline.**

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.

3.5 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 3.4.

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 [90]. 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 3.7.

3.6 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 hypothesizing new possible relationships and removing relationships in which a counterexample is found. This is shown in Figure 3.3 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.

3.7 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.

3.8 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 3.5 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 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

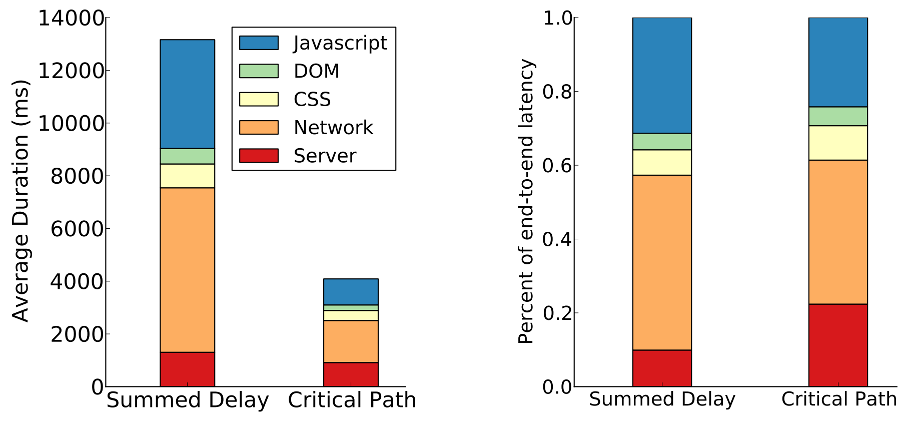
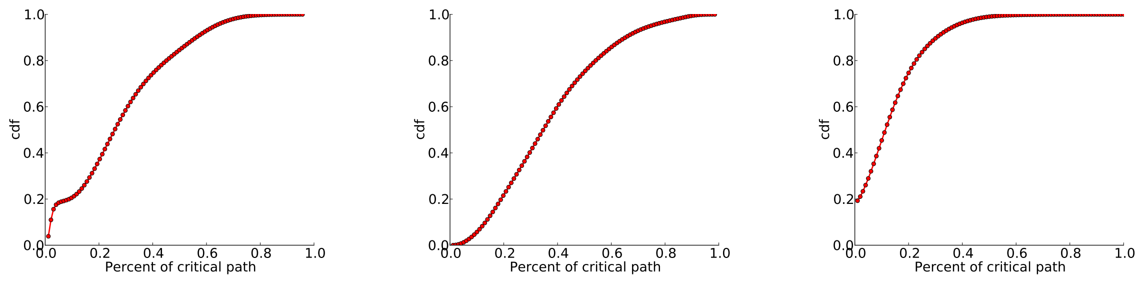


Figure 3.5: **Mean End-to-End Performance Breakdown.** Simply summing delay measured at each system component ("Summed Delay") ignores overlap and underestimates the importance of server latency relative to the actual mean critical path ("Critical Path").

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 3.6 shows



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

Figure 3.6: **Cumulative distribution of the fraction of the critical path attributable to server, network, and client portions**

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 3.9.

Stratification by connection type. We first consider stratifying by the type

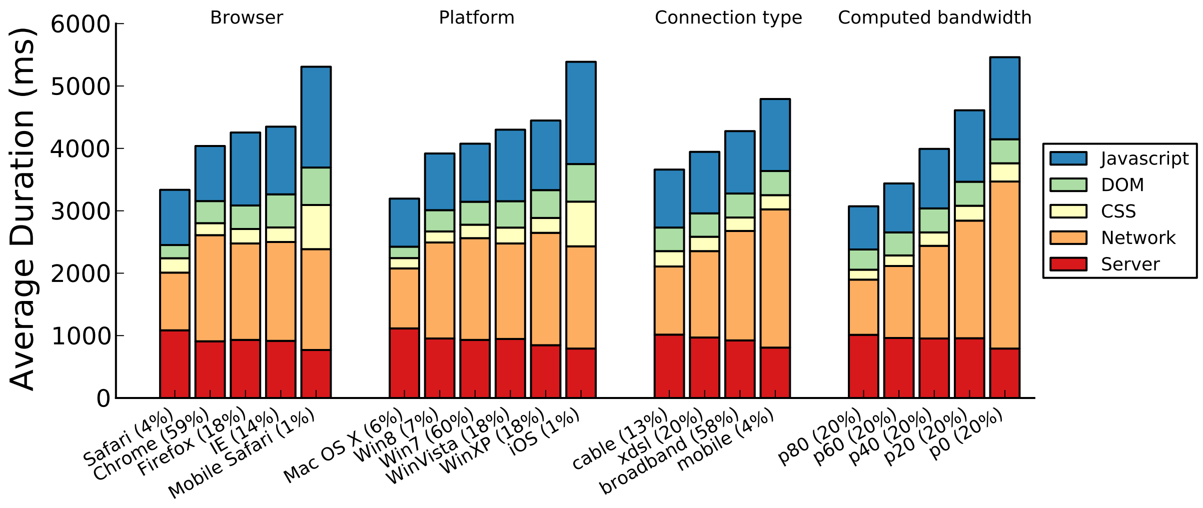


Figure 3.7: **Critical path breakdowns stratified by browser, platform, connection type, and computed bandwidth**

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 3.7 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 client operating system is a hint to the kind of client device, which in turn may suggest relative client performance. Figure 3.7 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 3.7, 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).

3.9 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.

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 3.8 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

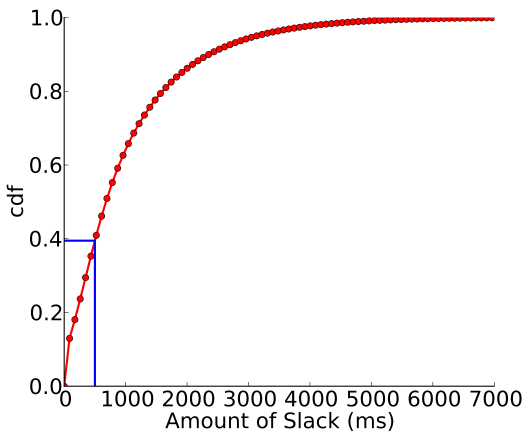


Figure 3.8: **Slack CDF for Last Data Item.** Nearly 20% of traces exhibit considerable slack—over 2 s—for the server segment that generates the last pagelet transmitted to the client. Conversely, nearly 20% of traces exhibit little (< 250 ms) slack.

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

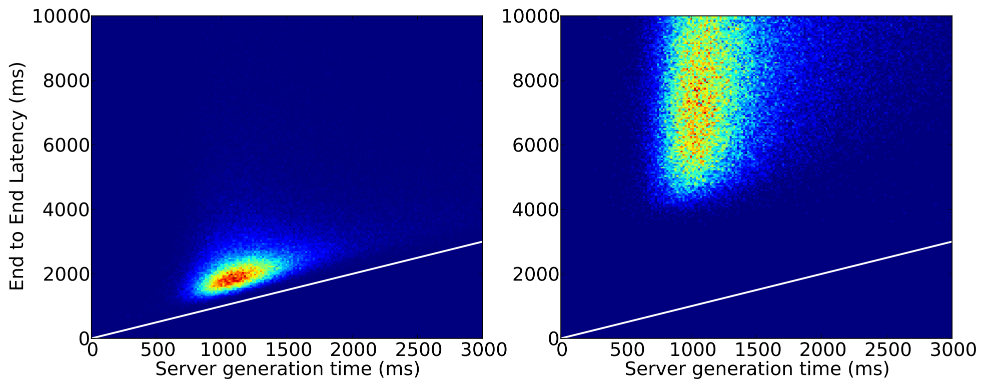


Figure 3.9: **Server vs. End-to-end Latency.** For the traces with slack below 25ms (left graph), there is strong correlation (clustering near $y = x$) between server and end-to-end latency. The correlation is much weaker (wide dispersal above $y = x$) for the traces with slack above 2.5s (right graph).

the 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 3.9 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). 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 3.10. Here,

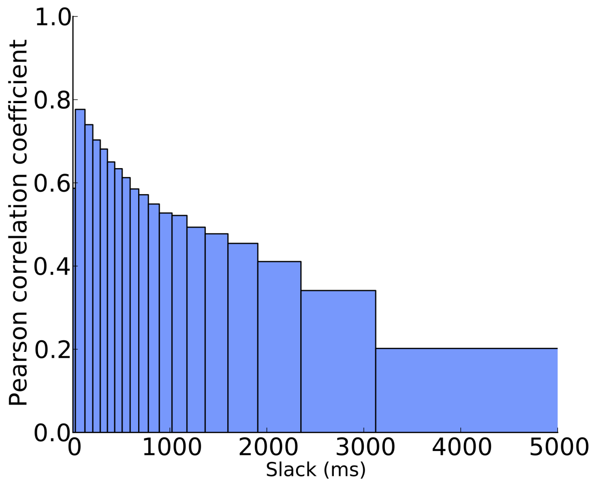


Figure 3.10: **Server–End-to-end Latency Correlation vs. Slack.** As reported slack increases, the correlation between total server processing time and end-to-end latency weakens, since a growing fraction of server segments are non-critical.

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

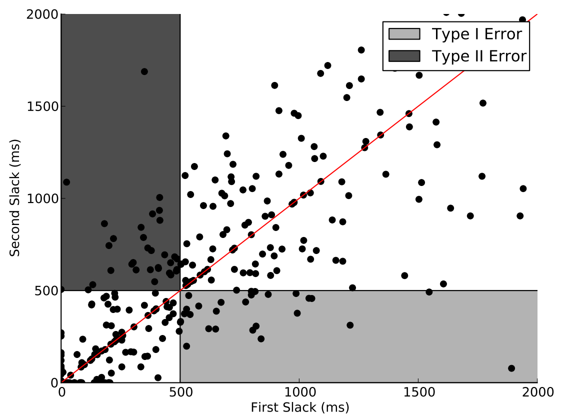


Figure 3.11: **Historical Slack as Classifier.** The clustering around the line $y = x$ shows that slack is relatively stable over time. The history-based classifier is correct 83% of the time. A type I error is a false positive, reporting slack as available when it is not. A type II error is a false negative.

with the user id via a persistent cookie or by storing the most recent slack estimate in Memcache [64].

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 3.11 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.

CHAPTER IV

DQBarge: Improving data quality tradeoffs in Internet services

4.1 Introduction

A *data-quality tradeoff* is an explicit decision by a software component to return lower-fidelity data in order to improve response time or minimize resource usage. Data-quality tradeoffs are often found in Internet services due to the need to balance the competing goals of minimizing the service response time perceived by the end user and maximizing the quality of the service provided. Tradeoffs in large-scale services are pervasive since hundreds or thousands of distinct software components may be invoked to service a single request, and each component may make individual data-quality tradeoffs.

Data-quality tradeoffs in low-level software components often arise from defensive programming. A programmer or team responsible for a specific component wishes to bound the response time of their component even when the resource usage or latency of a sub-service is unpredictable. For instance, a common practice is to time out when a sub-service is slow to respond and supply a lower-fidelity value in lieu of the requested data.

To quantify the prevalence of data-quality tradeoffs, we undertake a systematic study of software components at Facebook. We find that over 90% of components perform data-quality tradeoffs instead of failing. Some tradeoffs we observe are using default values, calculating aggregates from only a subset of input values, and retrieving alternate values from a stale or lower-quality data source. Further, we observe that the vast majority of data-quality tradeoffs are reactive rather than proactive, e.g., components typically set timeouts and make a data-quality tradeoff when a timer expires rather than attempt to predict and perform only those actions that can be performed within a desired time bound.

These existing data-quality tradeoffs are suboptimal for three reasons. First, they consider only local knowledge available to the low-level software component because of the difficulty in accessing higher-level knowledge such as the provenance of data, system load, and whether the component is on the critical request path. Second, the tradeoffs are usually reactive (e.g., happening only after a timeout) rather than proactive (e.g., issuing only the amount of sub-service requests that can be expected to complete within a time bound); reactive tradeoffs waste resources and exacerbate system overload. Finally, there is no mechanism to trace the set of data-quality tradeoffs made during a request, and this makes understanding the quality and performance impact of such tradeoffs on actual requests difficult.

DQBarge addresses these problems by propagating critical information related to data-quality tradeoffs along the causal path of request processing. The propagated data includes load metrics, as well as the expected critical path and slack for individual software components. It also includes provenance for request data such as the data sources queried and the software components that have transformed the data. Finally, it includes the specific data-quality tradeoffs that have been made for each request; e.g., data values left out of aggregations.

DQBarge uses this data to generate performance and quality models for low-level tradeoffs in the service pipeline. It consults the models to proactively determine which tradeoffs to make for future requests.

DQBarge generates performance and quality models by sampling a small percentage of the total requests processed by the service and redundantly executing them to compare the performance and quality when different tradeoffs are employed. Performance models capture how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and provenance. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of input data provenance.

These models enable better tradeoffs during subsequent request executions. DQBarge passes extra data along the causal path of request processing. It predicts the critical path for each request and the components along the requests path that will have substantial slack in processing time. It also measures current system load. This global and request-specific state is attached to the request at ingress. As the request propagates through software components, DQBarge annotates data objects with provenance (e.g., the sources from which data was retrieved and the algorithms used to generate the data). This information and the generated models are propagated to the low-level components, enabling them to make better tradeoffs.

We investigate three scenarios in which better data-quality tradeoffs can help. First, during unanticipated load spikes, making better data quality tradeoffs can maintain end-to-end latency goals while minimizing the loss in fidelity perceived by the end user. Second, when load levels permit, components with slack (because they are off the critical path of request processing) can improve the fidelity of the response without impacting the end-to-end latency. Finally, understanding the potential effects of low-level data-quality tradeoffs can inform dynamic capacity planning and maximize utility as a function of the resources required to produce output.

Thus, this work makes the following contributions. We provide the first comprehensive study of low-level data-quality tradeoffs in a large-scale Internet service.

Second, we observe that causal propagation of request statistics and provenance enables better and more proactive data-quality tradeoffs. Finally, we demonstrate the feasibility of this approach by designing, implementing, and evaluating DQBarge, an end-to-end approach for tracing, modeling, and actuating data-quality tradeoffs in Internet service pipelines.

We have added a complete, end-to-end implementation of DQBarge to Sirius [39], an open-source, personal digital assistant service. We have also implemented and evaluated the main components of the DQBarge architecture at Facebook and validated it with production data. Our results show that DQBarge can meet latency goals during load spikes, utilize spare resources without impacting end-to-end latency, and maximize utility by dynamically adding capacity for a service.

4.2 Study of data-quality tradeoffs

In this section, we quantify the prevalence and type of data-quality tradeoffs in production software at Facebook. We perform a comprehensive study of Facebook client services that use an internal key-value store called *Laser*. *Laser* enables online accessing of results of a batch offline computation such as a Hive [90] query.

We chose to study clients of *Laser* for several reasons. First, *Laser* had 463 client services, giving us a broad base of software to examine. Second, many details about timeouts and tradeoffs are specified in client-specific RPC configuration files for this store. We were able to process these files automatically, which reduced the amount of manual code inspection required for the study. Finally, we believe a key-value store is representative of the low-level components employed by most large-scale Internet companies.

Table 4.1 shows the results of our study for the 50 client services that invoke *Laser* most frequently, and Table 4.2 shows the results for all 463 client services. We categorize how clients make data-quality decisions along two dimensions: proactivity

	Failure	Data-quality tradeoff		
		Default	Omit	Low fidelity
Reactive	5	14	30	1
Proactive	0	0	2	1

Table 4.1: **Data-quality decisions of the top 50 *Laser* clients.** Each box shows the number of client services that make decisions according to the specified combination of reactive/proactive determination and resultant action. The total number of values shown is greater than 50 since a few clients use more than one strategy.

	Failure	Data-quality tradeoff		
		Default	Omit	Low fidelity
Reactive	40	250	174	4
Proactive	0	3	7	1

Table 4.2: **Data-quality decisions made by all *Laser* clients.** Each box shows the number of client services that make tradeoffs according to the specified combination of reactive/proactive determination and resultant action. The total number of values shown is greater than 463 since a few clients use more than one strategy.

and resultant action. Each table entry lists the number of client services that make at least one data quality decisions with a specific proactivity/action combination. We find that most clients employ a single strategy for all of their requests; only a few use different strategies for different requests. When a client uses multiple strategies, we include it in all relevant categories. Thus, the total number of values in each table is slightly more than the number of clients.

4.2.1 Proactivity

We consider a tradeoff to be *reactive* if the client service always initiates the request and then uses a timeout or return code to determine if the request is taking too long or consuming too many resources. For instance, we observed many latency-sensitive clients that set a strict timeout for how long to wait for a response. If *Laser* takes longer than the timeout, such clients make a data-quality tradeoff or return failure.

A *proactive* check predicts whether the expected latency or resource cost of processing the request will exceed a threshold. If so, a data-quality tradeoff is made immediately without issuing the request. For example, we observed a client that determines whether or not a query will require cross-data-center communication because such communication would cause it to exceed its latency bound. If there are no hosts that can service the query in its data center, it makes a data-quality tradeoff.

4.2.2 Resultant actions

We also examine the actions taken in response to latency or resource usage exceeding a threshold. *Failure* shows the number of clients that require a response from *Laser*. If the store responds with an error or timeout, the client service fails. Such instances mean a programmer has chosen to not make a data-quality tradeoff.

The remaining categories represent different types of data-quality tradeoffs. *Default* shows the number of client services that return a pre-defined default answer when a tradeoff is made. For instance, we observed a client service that ranks chat threads according to their activity level. The set of most active chat groups are retrieved from *Laser* and boosted to the top of a chat bar. If retrieving this set of active chat groups fails or times out, chat groups and contacts are listed alphabetically.

The *Omit* category is common in client services that aggregate hundreds of values from different sources; e.g., to generate a model. If an error or timeout occurs retrieving values from one of these sources, those values are left out and the aggregation is performed over the values that were retrieved successfully.

One specific example we observed is a recommendation engine that aggregates candidates and features from several data sources. It is resilient to missing candidates and features. Although missing candidates are excluded from the final recommendation and missing features negatively affect candidate scores in calculating the recommendation, the exclusion of a portion of these values allows a usable, but slightly lower-fidelity recommendation to be returned in a timely manner in the event of failure or unexpected system load.

The *Low fidelity* category denotes client services that make a tradeoff by retrieving an alternate, reduced quality value from a different data source. For example, we observed a client that requests a pre-computed list of top videos for a given user. If a timeout or failure occurs retrieving this list, the client retrieves a more generic set of videos for that user. As a further example, we observed a client that chooses among a pre-ranked list of optimal data sources. On error or timeout, the client retrieves the data from the next best data source. This process continues until a response is received.

Before performing our study, we hypothesized that client services might try to retrieve data of equal fidelity from an alternate data store in response to a failure. However, we did not observe any instance of this behavior in our study (all alternate sources had lower-fidelity data). Thus, we do not list this category in our results.

4.2.3 Discussion of results

Tables 4.1 and 4.2 show that data quality tradeoffs are pervasive in the client services we study. 90% of the top 50 *Laser* clients and 91% of all 463 clients perform a data-quality tradeoff in response to a failure or timeout; the remaining 9-10% of clients consider the failure to retrieve data in a timely manner to be a fatal error. Thus, in the Facebook environment, making data-quality tradeoffs is normal behavior, and failures are the exception.

For the top 50 clients, the most common action when faced with a failure or timeout is to omit the requested value from the calculation of an aggregate (60% of the top 50 clients do this). The next most common action (28% of the top 50 clients) is to use a default value in lieu of the requested data. These trends are reversed when considering all clients. Only 36% of all 463 clients omit the requested values from an aggregation, whereas 52% use a default value.

We were surprised that only a few clients react to failure or timeout by attempting to retrieve the requested data from an alternate source (4% of the top 50 clients and 1% of all clients). This may be due to tight time or resource constraints; e.g., if the original query takes too long, there may be no time left to initiate another query.

Only 6% of the top 50 clients and 2% of all clients are proactive. The lack of proactivity represents a significant lost opportunity for optimization because requests that timeout or fail consume resources but produce no benefit. This effect can be especially prominent when requests are failing due to excessive load; a proactive strategy would decrease the overall stress on the system. Failure of a proactive check always results in a data-quality tradeoff rather than a failure; this makes sense because it would be very pessimistic for a client to return a failure without at least attempting to fetch the needed data.

In our inspection of source code, we observed that low-level data-quality decisions are almost always encapsulated within clients and not reported to higher-level components or attached to the response data. Thus, there is no way for operators to check how the quality of the response being sent to the user has been impacted by low-level quality tradeoffs during request processing.

4.3 Design and implementation

Motivated by our study results, we designed DQBarge to help developers understand the impact of data-quality tradeoffs and make better, more proactive tradeoffs to improve quality and performance. Our hypothesis is that propagating additional information along the causal path of request processing will provide the additional context necessary to reach these goals.

Section 4.3.1 describes how DQBarge gathers and propagates data about request processing, including system load, critical path and slack predictions, data provenance, and a history of the tradeoffs made during request processing. Section 4.3.2 relates how DQBarge duplicates the execution of a small sample of requests to build models of performance and quality for potential data-quality tradeoffs. As described in Section 4.3.3, DQBarge uses these models to make better tradeoffs for subsequent requests: it makes proactive tradeoffs to reduce resource wastage, and it uses provenance to choose tradeoffs that lead to better quality at a reduced performance cost. Finally, Section 4.3.4 describes how DQBarge logs all tradeoffs made during request processing so that operators can review how system performance and request quality have been impacted.

4.3.1 Data gathering and propagation

DQBarge provides a library for developers to specify the information that should be propagated along the critical path. Developers use the library interface to annotate objects during request processing and query those annotations at later stages of the pipeline. The DQBarge library has a RPC-package-specific back-end that modifies and queries existing RPC objects to propagate the information.

DQBarge modifies RPC objects by adding additional fields that contain data to be propagated along the causal path. It supports three object scopes: request-level, component-level, and data-level.

Request-level objects are passed through all components involved in processing the request, following the causal path of request execution. For example, the systems in our case studies both have a global object containing a unique request identifier. DQBarge appends request-level information to this object. Request-level data includes system-wide load metrics, slack predictions, and a list of data-quality tradeoffs made during request execution. This technique for passing and propagating information is widely used in other tracing systems that follow the causal path of execution [36, 57].

Component-level objects persist from the beginning to end of processing for a specific software component within the request pipeline. Such objects are passed to all

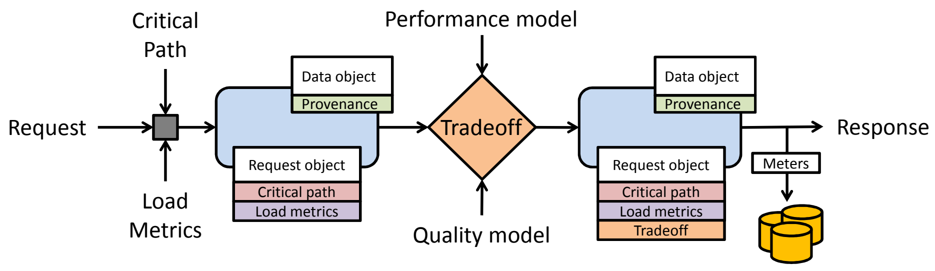


Figure 4.1: **DQBarge overview.**

sub-components that are called during the execution of the higher-level component. DQBarge appends component-specific data to these objects, so such data will be automatically deallocated when execution passes beyond the specified component. Component-specific load metrics are one example of data put in component-level objects.

Data-level objects are the specific data items being propagated as a result of request execution. DQBarge associates provenance with each data item, since the provenance is meaningful only as long as the data item exists.

Our library provide a useful interface for manipulating RPC objects, but developers must still make domain-specific decisions, e.g., what metrics and provenance values to add, what objects to associate with those values, and what rules should be used to model the propagation of provenance. Figure 4.1 shows an overview of how this data propagates through the system.

Load metrics may be relevant to the entire request or only to certain components. Each load metric is represented as a typed key-value pair (e.g., a floating point value associated with the key "requests/second"). Currently supported load metrics are throughput, CPU load, and memory usage.

Critical path and slack predictions are specified as directed acyclic graphs. Each software component in the graph has a weight that corresponds to its predicted slack (the amount of additional time it could take to process a request without affecting the end-to-end latency of the request). Components on the critical path of request execution have zero slack. Currently, slack predictions are made at request ingress based on historical log data; such predictions may cover the entire request or only specific components of the request.

DQBarge associates provenance with the data objects it describes. Provenance can be a data source or the algorithm employed to generate a particular object. Provenance is represented as an unordered collection of typed key-value pairs. DQBarge supports both discrete and continuous types. The types allow DQBarge to extract a schema from the data objects being passed to a component making a data-quality decision. Components are treated as black boxes, so developers must specify how provenance is propagated when a component modifies existing data objects or creates new ones.

Finally, DQBarge stores the tradeoffs that were made during request processing in a request-level object. As described in Section 4.3.4, this information may be logged and used for reporting the effect of tradeoffs on quality and performance.

4.3.2 Model generation

For each potential tradeoff, DQBarge creates a performance model and a quality model that capture how the tradeoff affects request execution. Performance models predict how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and the provenance of input data. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of provenance.

DQBarge uses *request duplication* to generate models from production traffic without adversely affecting the user experience. At the RPC layer, it randomly samples incoming requests from production traffic, and it routes a copy of the selected requests to one or more request duplication pipelines. Such pipelines execute isolated, redundant copies of the request for which DQBarge can make different data-quality tradeoffs. These pipelines do not return results to the end user and they are prevented from making modifications to persistent stores in the production environment; in all other respects, request execution is identical to production systems. Many production systems, including those at Facebook, already have similar functionality for testing purposes, so adding support for model generation required minimal code changes.

DQBarge controls the rate at which requests enter the duplication pipeline by changing the sampling frequency. At each potential tradeoff site, software components query DQBarge to determine which tradeoffs to make; DQBarge uses these hooks to systematically explore different tradeoff combinations and generate models. At the pipeline egress, DQBarge inserts meters that record both the request processing latency and the final response, which it uses to calculate a service-specific measure of quality.

To generate a performance model, DQBarge uses load testing [45]. Each data-quality tradeoff offers multiple fidelities. A default value may be used or not. Different types or percentages of values can be left out of an aggregation. Multiple alternate data stores may be used. For each fidelity, DQBarge starts with a low request rate and increases the request rate until the latency exceeds a threshold. Thus, the resulting model shows request processing latency as a function of request rate and tradeoffs made (i.e., the fidelity of the tradeoff selected). DQBarge also records the provenance of the input data for the component making the tradeoff; the distribution of provenance is representative of production traffic since the requests in the duplication pipeline are a random sampling of that traffic. DQBarge determines whether the resulting latency distribution varies as a result of the input provenance; if so, it generates separate models for each provenance category. However, in the systems we have examined, provenance has not had a statistically significant effect on performance (though it significantly affects quality).

Even though generating performance models is performed offline, there is still a non-trivial cost with producing these models. First, resources must be allocated in order to understand and reach the full limits of the service. Additionally, performing a parameter sweep of the request rate can take up to approximately an hour. Producing the performance curves for each tradeoff rate can be done in parallel but at the expense of additional resources. The request rate is held until the variance has settled so that accurate measurements of performance in production can be gathered. For example, for systems running on the JVM with a JIT, additional requests are needed to warm up the system to accurately model the distribution of request latencies.

Quality models capture how the fidelity of the final response is affected by data-quality tradeoffs during request processing. To generate a quality model, DQBarge sends each request to two duplication pipelines. The first pipeline makes no trade-offs, and so produces a full-fidelity response. The second pipeline makes a specified tradeoff, and so produces a potentially lower-fidelity response. DQBarge measures the quality impact of the tradeoff by comparing the two responses and applying a service-specific quality ranking. For example, if the output of the request is a ranked list of Web pages, then a service-specific quality metric might be the distance between where pages appear in the two rankings.

DQBarge next learns a model of how provenance affects request quality. As described in the previous section, input data objects to the component making the tradeoff are annotated with provenance in the form of typed key-value pairs. These pairs are the features in the quality model. DQBarge uses multidimensional linear regression to model the importance of each provenance feature in the quality of the request result. For example, if a data-quality tradeoff omits values from an aggregation, then omitting values from one data source may have less impact on response quality than omitting values from a different source.

Provenance can substantially reduce the number of observations needed to generate a quality model. Recall that all RPC data objects are annotated with provenance;

thus, the objects in the final request result have provenance data. In many cases, the provenance relationship is direct; an output object depends only on a specific input provenance. In such cases, we can infer that the effect of a data-quality tradeoff would be to omit the specified output object, replace it with a default value, etc. Thus, given a specific output annotated with provenance, we can infer what the quality would be if further data-quality tradeoffs were made (e.g., a specific set of provenance features were used to omit objects from an aggregation). In such cases, the processing of one request can generate many different data points for the quality model. If the provenance relationship is not direct, DQBarge generates these data points by sampling more requests and making different tradeoffs.

Because code changes will gradually render such models obsolete, we envision that DQBarge will continuously or periodically update its models by sampling a low percentage of production traffic.

4.3.3 Using the models

DQBarge uses its performance and quality models to make better, more proactive data-quality tradeoffs. System operators specify a high-level goal such as maximizing quality given a latency cap on request processing. Software components call DQBarge at each potential tradeoff point during request processing; the library returns a decision as to whether a tradeoff should be made and, if appropriate, what fidelity should be employed (e.g., which data source to use or which values to leave out of an aggregation). The software component then implements that decision proactively; i.e., it makes the tradeoff immediately.

DQBarge currently supports three high-level goals: maximizing quality subject to a component-level latency constraint, maximizing quality while using only slack execution time available during request processing, and maximizing utility as a function of quality and performance. These goals are useful for mitigating load spikes, efficiently using spare resources, and implementing dynamic capacity planning, respectively. We next describe these three goals in more detail.

4.3.3.1 Load Spikes

Services are provisioned to handle peak request loads. However, changes in usage or traffic are unpredictable; e.g., the launch of a new feature may introduce additional traffic. Thus, systems are designed to handle unexpected load spikes; the reactive data-quality tradeoffs we saw in Section 4.2 are one such mechanism. DQBarge improves on existing practice by letting an operator specify a maximum latency for a request or a component of request processing. It maximizes quality subject to this constraint by making data-quality tradeoffs.

At each tradeoff site, there may be many potential tradeoffs that can be made (e.g., sets of values with different provenance may be left out of an aggregation or distinct alternate data stores may be queried). DQBarge orders possible tradeoffs by “bang for the buck” and greedily selects tradeoffs until the latency goal is reached. It ranks each potential tradeoff by the ratio of the projected improvement in latency (given by the performance model) to the decrease in request fidelity (given by the quality model). The independent parameters of the model are the current system load and the provenance of the input data. DQBarge selects tradeoffs in descending order of this ratio until the performance model predicts that the latency limit will be met.

4.3.3.2 Utilizing spare resources

Because DQBarge has a prediction of request processing time for each software component, it can estimate which components are on the critical path and which components have slack available in processing time. If a component has slack, DQBarge can make tradeoffs that improve quality without negatively impacting the end-to-end request latency observed by the user. Similar to the previous scenario, DQBarge calculates the ratio of quality improvement to latency decrease for each potential tradeoff (the difference is that this goal involves improving quality rather than performance). It then greedily selects tradeoffs according to this order until the additional latency would exceed the projected slack time.

4.3.3.3 Dynamic capacity planning

DQBarge allows operators to specify the utility (e.g., the dollar value) of reducing latency and improving quality. It then selects the tradeoffs that improve utility until no more such tradeoffs are available. DQBarge also allows operators to specify the impact of adding or removing resources (e.g., compute nodes) as a utility function parameter. DQBarge compares the value of the maximum utility function with more and less resources and generates a callback if adding or removing resources would improve the current utility. Such callbacks allow dynamic re-provisioning.

4.3.3.4 Discussion

DQBarge does not guarantee an optimal solution since it employs greedy algorithms to search through potential tradeoffs. However, an optimal solution is likely unnecessary given the inevitable noise that arises from predicting traffic and from errors in modeling. For the last use case, DQBarge assumes that operators can quantify the impact of changes to service response times, quality, and the utilization of additional resources. DQBarge also assumes that tradeoffs are independent, since calculating models over joint distributions would be difficult.

4.3.4 Logging data-quality decisions

DQBarge logs all data-quality decisions and includes them in the provenance of the request data objects. The information logged includes the software component, the point in the execution where a tradeoff decision was made, and the specific decision that was made (e.g., which values were left out of an aggregation). To reduce the amount of data that is logged, only instances where a tradeoff was made are recorded. Timeouts and error return codes are also logged if they result in a reactive data-quality tradeoff. This information helps system administrators and developers understand how low-level data-quality tradeoffs are affecting the performance and quality of production request processing.

4.4 Case studies

We have implemented the main components of DQBarge in a portion of the Facebook request processing pipeline, and we have evaluated the results using Facebook production traffic. Our current Facebook implementation allows us to track provenance, generate performance and quality models and measure the efficacy of the data-quality tradeoffs available through these models. This implementation thus allows us to understand the feasibility and potential benefit of applying these ideas to current production code.

We have also implemented the complete DQBarge system in Sirius [39], an open-source personal digital assistant akin to Siri. Our Sirius implementation enables end-to-end evaluation of DQBarge, such as observing how data-quality tradeoffs can be used to react to traffic spikes and the availability of slack in the request pipeline.

4.4.1 Facebook

Our implementation of DQBarge at Facebook focuses on a page ranking service, which we will call *Ranker* in this paper. When a user loads the Facebook home page, *Ranker* uses various parameters of the request, such as the identity of the requester, to generate a ranked list of page recommendations. *Ranker* first generates candidate recommendations. It has a flexible architecture that allows the creation and use of multiple candidate generators; at the time of our study, there were over 30 candidate generators that collectively produced hundreds of possible recommendations for each request.

Ranker retrieves feature vectors for each candidate from *Laser*, the key-value store we studied in Section 4.2. *Ranker* is a service that makes reactive data-quality tradeoffs. If an error or timeout occurs when retrieving features, *Ranker* omits the candidate(s) associated with those features from the aggregation of candidates and features considered by the rest of the *Ranker* pipeline.

Ranker uses the features to calculate a score for each candidate. The algorithm for calculating the score was opaque to us (it is based on a machine learning model regenerated daily). It then orders candidate by score and returns the top N candidates.

DQBarge leverages existing tracing and monitoring infrastructure at Facebook. It uses a production version of the Mystery Machine tracing and performance analysis infrastructure [30]. This tool discovers and reports performance characteristics of the processing of Facebook requests, including which components are on the critical path. From this data, we can calculate the slack available for each component of request processing; prior results [30] showed that, given an observation of past requests by the same user, slack for future requests can be predicted with high accuracy. Existing Facebook systems monitor load at each component in the pipeline.

DQBarge annotates data passed along the pipeline with provenance. The data object for each candidate is annotated with the generator that produced the data. Similarly, features and other data retrieved for each candidate are associated with their data source.

We implemented meters at the end of the *Ranker* pipeline that measure the latency and quality of the final response. To measure quality, we compare the difference in ranking of the top N pages returned from the full-quality response (with no data-quality tradeoffs made) and the lower-fidelity response (that includes some tradeoffs). For example, if the highest-ranked page in the lower-fidelity response is the third-ranked page in the full-quality response, the *quality drop* is two.

4.4.2 Sirius

We also added DQBarge to Sirius [39], an open-source personal assistant similar to Apple's Siri or Google Now. Sirius answers fact-based questions based on a set of configurable data sources. The default source is an indexed Wikipedia database; an operator may add other sources such as online search engines.

Sirius generates several queries from the question; each query represents a unique method of parsing the question. For each query, it generates a list of documents that are relevant to answering the query. Each document is passed through a natural language processing pipeline to derive possible answers. Sirius assigns each answer a numerical score and returns the top-ranked answer.

Data-quality tradeoffs in Sirius occur when aggregating values from multiple sub-service queries. Our DQBarge implementation makes these tradeoffs proactively by using quality and performance models to decide which documents to leave out of the aggregation when the system is under load.

Initially, Sirius did not have request tracing or load monitoring infrastructure. We therefore added the ability to trace and measure the performance by integrating the Mystery Machine[30] software package. Each request has a unique identifier that is propagated through every component of the pipeline. The performance of each component is measured and the causal relationship among components is determined through empirical observation of large sets of traces. This allows the request critical path and slack for each component to be calculated offline. We also implemented a slack predictor using this data that estimates the available slack for each component. For load, we added counters at each pipeline stage to measure request rates. Additionally, we track the CPU load and memory usage of the entire service. The performance data, predicted slack, and load information are all propagated by DQBarge as each request flows through the Sirius pipeline.

In each stage of the Sirius pipeline, provenance is propagated along with data objects. For example, when queries are formed from the original question, the algorithm used to generate the query is associated with the query object. Sirius provenance also includes the data used to generate the list of candidate documents.

Since Sirius did not have a request duplication mechanism, we added the ability to sample requests and send the same request through multiple instances of the Sirius pipeline. User requests are read-only with respect to Sirius data stores, so we did not have to isolate any modifications to service state from duplicated requests.

4.5 Evaluation

Our evaluation answers the following questions:

- Do data-quality tradeoffs improve performance?
- How much does provenance improve tradeoffs?
- How much does proactivity improve tradeoffs?
- How well does DQBarge meet end-to-end performance and quality goals?

4.5.1 Experimental setup

For *Ranker*, we perform our evaluation on Facebook servers using live Facebook traffic by sampling and duplicating *Ranker* requests. Our entire implementation uses request duplication pipelines, so as to not affect the results returned to Facebook users. We change the system load by sampling a larger or smaller number of *Ranker* requests.

For Sirius, we evaluated our end-to-end implementation of DQBarge on 16-core

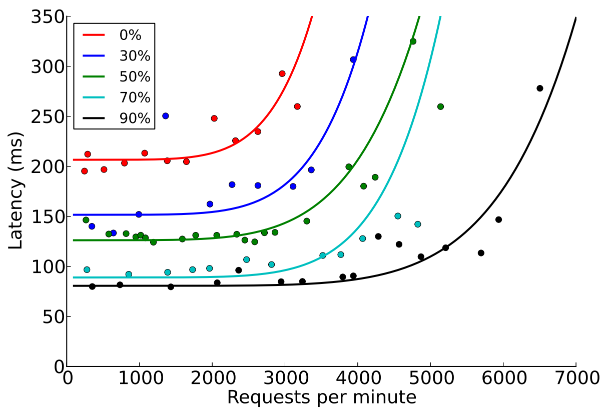


Figure 4.2: ***Ranker* performance model** This graph shows the effect of varying the frequency of data-quality tradeoffs on *Ranker* request latency. We varied the request rate by sampling different percentages of live production traffic at Facebook.

3.1GHz Xeon servers with 96GB of memory. We send Sirius questions sampled from an archive from previous TREC conferences [89].

4.5.2 Performance benefits

We first measure the effect of data-quality tradeoffs on throughput and latency by generating performance models for *Ranker* and Sirius. DQBarge performs a full parameter sweep through the dimensions of request rate, tradeoff frequency, and provenance of the data being considered for each tradeoff, sampling at regular intervals. For brevity, we report only a portion of these results.

4.5.2.1 *Ranker*

Figure 4.2 shows the latency-response curve for *Ranker* when DQBarge varies the incoming request rate. Each curve shows the best fit for samples taken at a different *tradeoff rate*, which we define to be the frequency at which data tradeoffs are made. At a tradeoff rate of 0%, no candidates are dropped from the *Ranker* aggregation.

These results show that data-quality tradeoffs substantially improve *Ranker* latency at low loads (less than 2500 requests/minute); e.g., at a 30% tradeoff rate, latency decreases by 28%. Prior work has shown that server slack at Facebook is

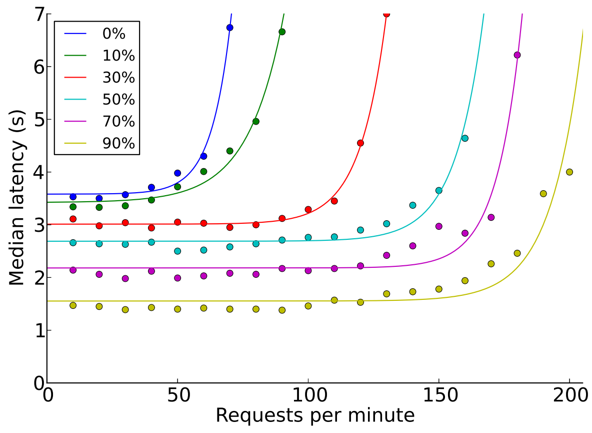
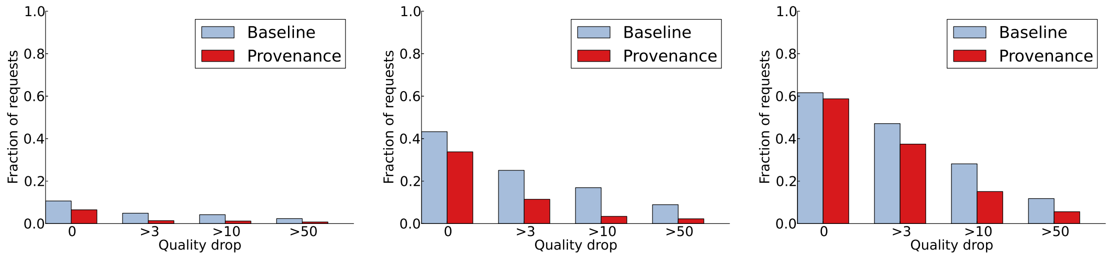


Figure 4.3: **Sirius performance model.** This graph shows the effect of varying the frequency of data-quality tradeoffs on Sirius request latency. Each curve shows a different tradeoff rate.

predictable on a per-request basis [30]. Thus, in this region, *Ranker* could make more tradeoffs to reduce end-to-end response time when *Ranker* is on the critical path of request processing, yet it could still provide full-fidelity responses when it has slack time for further processing.

Data-quality tradeoffs also improve scalability under load. Taking 250ms as a reasonable knee in the latency-response curve, *Ranker* can process approximately 2500 requests per minute without making tradeoffs, but it can handle 4300 requests per minute when the tradeoff rate is 50% (a 72% increase). This allows *Ranker* to run at a lower fidelity during a load spike.

DQBarge found that the provenance of the data values selected for tradeoffs does not significantly affect performance. In other words, while the number of tradeoffs made has the effect shown in Figure 4.2, the specific candidates that are proactively omitted from an aggregation do not matter. Thus, we only show the effect of the request rate and tradeoff rate.



(a) Tradeoff rate 10% (b) Tradeoff rate 50% (c) Tradeoff rate 80%

Figure 4.4: **Impact of provenance on *Ranker* quality.** We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the top ranked page, which is the difference between where it appears in the *Ranker* rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.

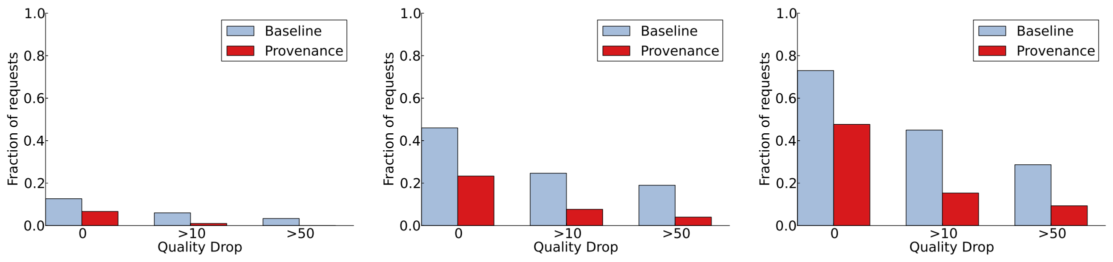
4.5.2.2 Sirius

Figure 4.3 shows results for Sirius. Like *Ranker*, the provenance of the data items selected for tradeoffs did not affect performance, so we show latency-response curves that vary both request rate and tradeoff rate.

The results for Sirius are similar to those for *Ranker*. A tradeoff rate of 50% reduces end-to-end request latency by 26%. Under load, a 50% tradeoff rate increases Sirius throughput by approximately 200%.

4.5.3 Effect of provenance

We next consider how much provenance improves the tradeoffs made by DQBarge. We consider a baseline quality model that does not take into account any provenance; e.g., given a target tradeoff rate, it randomly omits data values from an aggregation. This is essentially the policy in existing systems like *Ranker* and Sirius because there is no inherent order in requests from lower-level services and data stores and timeouts therefore affect a random sampling of the values returned. In contrast, DQBarge uses its quality model to select which values to omit, with the objective of choosing those that affect the final output the least.



(a) Tradeoff rate 10% (b) Tradeoff rate 50% (c) Tradeoff rate 80%

Figure 4.5: **Impact of provenance on Sirius quality.** We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the Sirius answer, which is the difference between where it appears in the Sirius rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.

4.5.3.1 *Ranker*

We first used DQBarge to sample production traffic at Facebook and construct a quality model for *Ranker*. DQBarge determined that, by far, the most important provenance parameter affecting quality is the generator used to produce a candidate. For example, one particular generator produces approximately 17% of the top-ranked pages but only 1% of the candidates. Another generator produces only 1% of the top-ranked pages but accounts for 3% of the candidates.

Figure 4.4 compares the quality of request results for DQBarge with a baseline that makes tradeoffs without using provenance. We sample live Facebook traffic, so the requests in this experiment are different from those used to generate the quality model. We vary the tradeoff rate and measure the *quality drop* of the top ranked page; this is the difference between where the page appears in the request that makes a data-quality tradeoff and where it would appear if no data-quality tradeoffs was made. The ideal quality drop is zero.

As shown in Figure 4.4a, at a low tradeoff rate of 10%, using provenance reduces the percentage of requests that experience any quality drop at all from 11% to 6%. With provenance, only 1% of requests experienced a quality drop of more than three,

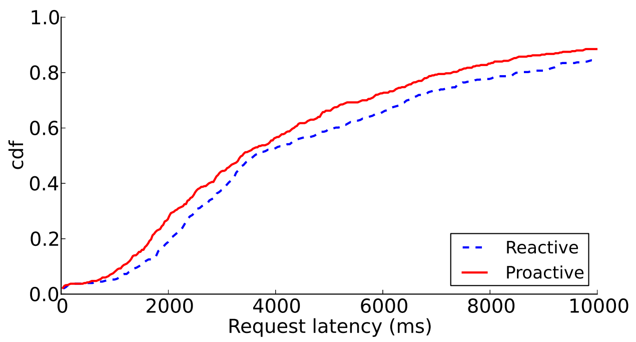


Figure 4.6: **Performance of reactive tradeoffs.** This graph compares the distribution of request latencies for Sirius when tradeoffs are made reactively via timeouts and when they are made proactively via DQBarge.

compared to 5% without provenance. Figure 4.4b shows a higher tradeoff rate of 50%. Using provenance decreases the percentage of requests with any quality drop at all from 43% to 33%. Only 3% of requests experienced a quality drop of 10 or more, compared to a baseline result of 17%. Figure 4.4c compares quality at a high tradeoff rate of 80%. Use of provenance still provides a modest benefit: 59% of requests experience a quality drop, compared to 62% for the baseline. Further, with provenance, the quality drop is 10 or more for only 15% of request compared with 28% for the baseline.

4.5.3.2 Sirius

For Sirius, we used k-fold cross validation to separate our benchmark set of questions into training and test data. The training data was used to generate a quality model based on provenance features, which included the language parsing algorithm used, the number of occurrences of key words derived from the question, the length of the data source document considered, and a weighted score relating the query words to the source document.

Figure 4.5 compares the quality of Sirius results with DQBarge using provenance with a baseline that does not use provenance. As shown in Figure 4.5a, at a tradeoff rate of 10%, provenance decreases the quality drop for the answer produced by

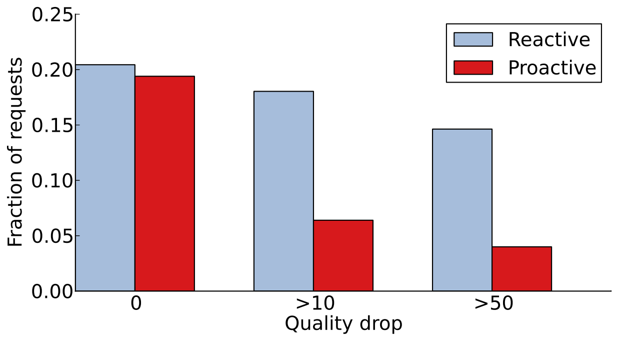


Figure 4.7: This graph shows that using proactive tradeoffs at a tradeoff rate of 40% can achieve higher quality tradeoffs than using reactive tradeoffs with a timeout of 1.5s in Sirius.

Sirius from 13% to 7%. Only 1% of requests see a quality drop of 10 or more using provenance, compared to 6% for the basline. Figure 4.5b shows that, for a higher tradeoff rate of 50%, provenance decreases the percentage of requests that see any quality drop from 46% to 23%. Further, only 8% of requests see a quality drop of 10 or more using provenance, compared to 25% for the baseline. Figure 4.5c shows a tradeoff rate of 80%; provenance decreases the percentage of requests that see any quality drop from 73% to 48%.

4.5.4 Effect of proactivity

We next examine how proactivity affects data-quality tradeoffs. In this experiment, we send requests to Sirius at a high rate of 120 requests per minute. Without DQBarge, this rate occasionally triggers a 1.5 second timeout for retrieving documents, causing some documents to be left out of the aggregation. These tradeoffs are reactive in that they occur only after a timeout expires. In contrast, with DQBarge, tradeoffs are made proactively at a rate of 40%.

Figure 4.6 shows request latency as a CDF for both the reactive and proactive methods of making data-quality tradeoffs. Comparing the two distributions shows that DQBarge improves performance across the board; e.g., the median request latency is 3.4 seconds for proactive tradeoffs and 3.6 seconds for reactive tradeoffs.

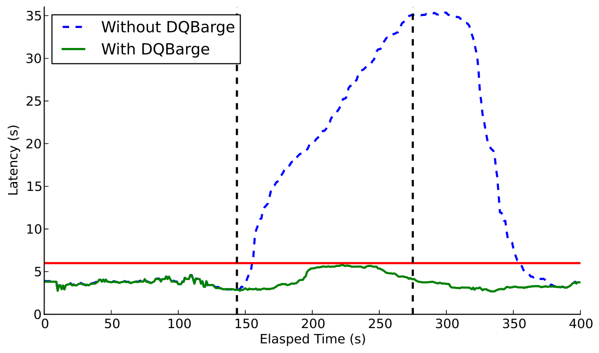


Figure 4.8: **Response to a load spike.** DQBarge makes data-quality tradeoffs to meet a median latency goal of 6s.

Figure 4.7 shows that proactive tradeoffs also improve quality. DQBarge slightly decreases the number of requests that have no quality drop from 20% to 19%. More significantly, it reduces the number of requests that have a quality drop of more than 10 from 18% to 6%.

Under high loads, reactive tradeoffs hurt performance because they waste resources (e.g., trying to retrieve documents that are not used in the aggregation). Further, their impact on quality is greater than with DQBarge because timeouts affect a random sampling of the values returned, whereas proactive tradeoffs omit retrieving those documents that are least likely to impact the reply.

4.5.5 End-to-end case studies

We next evaluate DQBarge with three end-to-end case studies on our Sirius testbed.

4.5.5.1 Load spikes

In this scenario, we introduce a load spike to see if DQBarge can maintain end-to-end latency and throughput goals by making data-quality tradeoffs. We set a target median response rate of 6 seconds. Normally, Sirius receives 50 requests/minute, but it experiences a two-minute load spike of 150 requests/minute in the middle of

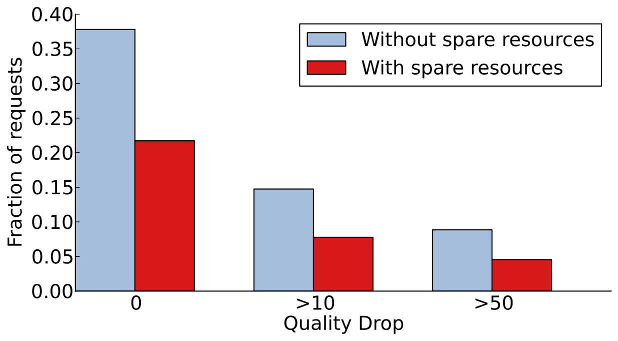


Figure 4.9: **Quality improvement using spare resources.** DQBarge uses slack in request pipeline stages to improve response quality.

the experiment. Figure 4.8 shows that without DQBarge, the end-to-end latency increases significantly due to the load spike. The median latency within the load spike region averages 25.2 seconds across 5 trials.

In comparison, DQBarge keeps median request latency below the 6 second goal throughout the experiment. Across 5 runs, the median end-to-end latency during the spike region is 5.4 seconds. In order to meet the desired latency goal, DQBarge generally selects a tradeoff rate of 50%, resulting in a mean quality drop of 6.7.

4.5.5.2 Utilizing spare resources

In the second scenario, we see if DQBarge can effectively use spare capacity and slack in the request processing pipeline to increase quality without affecting end-to-end latency. Sirius is configured to use both its default Wikipedia database and the Bing Search API [14] to answer queries. Each source has a separate pipeline that executes in parallel before results from all sources are compared at the end. The Bing pipeline tends to take longer than the default pipeline, so slack typically exists in the default pipeline stages.

As described in Section 4.4.2, DQBarge predicts the critical path for each request and the slack for pipeline stages not on the critical path. If DQBarge predicts there is slack available for a processing pipeline, it reduces the tradeoff frequency to increase

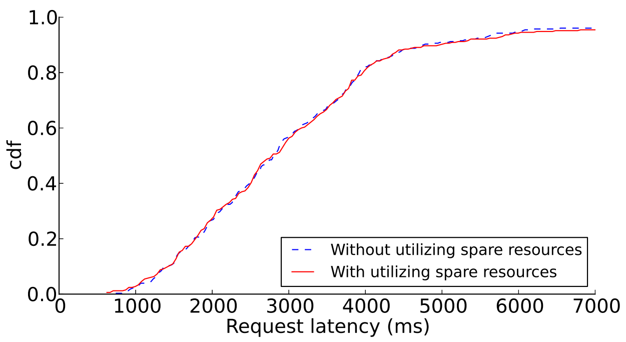


Figure 4.10: **Performance impact of using spare resources.** When DQBarge uses slack in request pipeline stages, it does not impact end-to-end latency.

quality until the predicted added latency would exceed the predicted slack. To give DQBarge room to increase quality, we set the default tradeoff rate to 50% for this experiment; note that this simply represents a specific choice between quality and latency made by the operator of the system.

Figure 4.9 shows that DQBarge increases quality for this experiment by using spare resources; the percentage of requests that exprience any quality drop decreases from 38% to 22% (as compared to a full-fidelity response with no data-quality tradeoffs). Figure 4.10 shows a CDF of request response times; because the extra processing occurs off the critical path, the end-to-end request latency is unchanged when DQBarge attempts to employ only spare resources to increase quality.

4.5.5.3 Dynamic capacity planning

Finally, we show how DQBarge can be used in dynamic capacity planning. We specify a utility function that provides a dollar value for reducing latency, improving quality, and provisioning additional servers. The utility of latency and quality are shown in Figure 4.11. DQBarge makes data-quality tradeoffs that maximize the utility function at the incoming request rate.

In this scenario, we examine the benefit of using DQBarge to decide when to provision addtional resrouces. We compare DQBarge with dynamic capacity planning

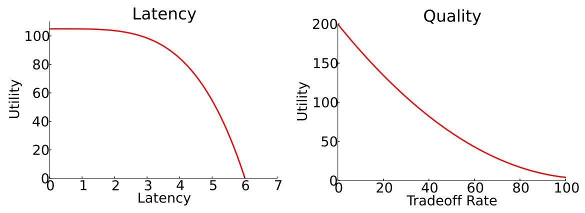


Figure 4.11: **Utility parameters for dynamic capacity planning.** These values are added together to calculate final utility.

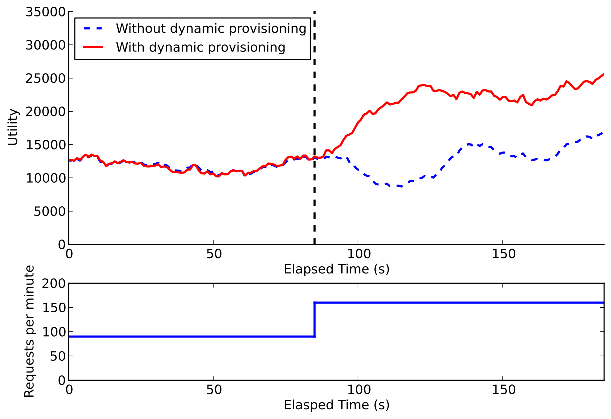


Figure 4.12: **Benefit of dynamic capacity planning.** With dynamic capacity planning, DQBarge improves utility by provisioning an additional server.

against DQBarge without dynamic capacity planning. Figure 4.12 shows the total utility of the system over time. When the request rate increases to 160 requests per minute, DQBarge reports that provisioning another server would provide a net positive utility. Using this server increases utility by an average of 58% compared to a system without dynamic capacity planning.

CHAPTER V

Related Work

The area of causality analysis is a well-studied field. The work presented in this thesis builds on many foundational ideas.

Taint tracking initially was used in the security domain to detect software attacks [62] online while a program is executing. Since its introduction, systems have used taint tracking for understanding fine-grained causality outside of the security domain. For example it has been applied to diagnose misconfigurations [8, 7], forensics [49], finding privacy leaks [34, 69, 55], and data provenance [32].

Online taint tracking systems require low overheads to run in production. This requires constraining the types of queries that can be run online. For example, a typical online query tracks whether sensitive data is leaked during the execution of a program [34]. The requirement of having low online overheads constrains the type of supportable queries to be relative simple; these queries require knowing which inputs are of interest beforehand. This constraint has limited the number of possible taint sources that can be tracked. It has also led to specialized hardware support to accelerate taint tracking [22, 93, 42]. Although these systems speed up online propagation of taint, the number of supported taint sources are limited by hardware constraints.

Our taint tracking framework is focused on after-the-fact analysis that is suitable for tasks like forensics [49], debugging misconfigurations [8, 7], and data provenance [32]. Our taint tracking framework differentiates itself from previous systems in its ability to track millions of input sources, orders of magnitude greater than the state of the art. The use of deterministic replay in our system enables low online overheads. Replaying executions offline enables complex queries that are too heavyweight to perform online. These complex queries allow for answering what inputs influenced a particular byte of output. Our taint tracking framework is able to support tens of millions of taint labels while also being able to support a wide variety of complex, multi-threaded programs. Additionally, our taint tracking framework works on application binaries, without the need of source code.

Besides taint tracking, there are other methods that have been used to understand fine-grained causality with various tradeoffs in speed, complexity, and overhead. Static taint analysis [6, 94] has been used to track causality from sources to sinks by examining all possible static code paths from source to sink. It also has the benefit of performing the analysis offline. However, these static techniques suffer from imprecision and over-tainting. Additionally, the number of sources are limited in order to have an efficient analysis. Reverse execution [68, 4] is a technique that allows for stepping back through an execution and examining the effects of a particular instruction. Recap [68] achieves the appearance of this effect by periodically taking checkpoints and logging system calls and shared memory accesses in order to reproduce intervals of program execution. This is similar to a deterministic replay system with checkpointing. Akgul et al. [4] statically produce a reverse execution binary of the program and uses dynamic slicing in order to answer backward queries. However, the static analysis required to produce a reverse execution version of the binary does not scale beyond simple programs.

Symbolic execution is another technique for deriving causality in a program [19, 27]. Symbolic execution replaces inputs with symbolic values in order to reason about the program logically. It enables analyzing all possible paths of program execution. Compared to taint tracking, the complexity in scaling symbolic execution is not in dealing with a large number of inputs, since inputs are represented symbolically, but rather the complexity of the program. Symbolic execution is prone to path explosion, making it difficult to reason about complex multi-threaded programs.

On the other side of the spectrum, large-scale Internet systems have posed difficulties in understanding causality between the software components that make up these systems. The scale, heterogeneity, and fast-changing nature of these systems makes understanding causality in these systems difficult. Previous systems have derived a model of causal dependencies can be derived from comprehensively instrumenting all middleware for communication, scheduling, and/or synchronization to record component interactions [3, 7, 21, 36, 50, 58, 76, 78, 87]. 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. Like these previous systems, *The Mystery Machine* uses the causal model to perform well-known performance analyses such as critical path, slack, and what-if analysis.

Finally, data-quality tradeoffs are a specific type of quality-of-service tradeoff [15, 65, 85], akin to recent work in approximate computing [10, 20, 44, 43, 84, 86]. The distinguishing feature of data-quality tradeoffs is that they are embedded in low-level software components within complex Internet pipelines. This leads to a lack of global knowledge and makes it difficult for individual components to determine how making specific tradeoffs will impact overall service latency and quality. DQBarge addresses this issue by incorporating principles from the literature on causal tracing [13, 21, 24, 36, 57, 76, 77, 87] to propagate needed knowledge along the path of request processing, enabling better tradeoffs by providing the ability to assess the impact of tradeoffs.

The remainder of this chapter discusses specific prior work that has influenced the work discussed in this thesis for scaling intraprocess taint tracking, scaling causality to large-scale systems, and using causality to make better data quality tradeoffs.

5.1 Scaling intraprocess taint tracking

The earliest taint tracking frameworks, such as TaintCheck [62], only support a single bit indicating whether data is tainted or untainted. TaintCheck focused on tracking a limited set of input from unknown network sources. Since then, LIFT [74] and libdft [48] are forward taint tracking frameworks with several optimizations that are suited for speeding up online dynamic taint tracking. Their taint representations allow for fast taint merge operations through the use of bit vectors. However, this fundamentally limits the number of taint labels that can be efficiently tracked. DyTan [31] is similar to our framework in being flexible in scoping inputs and outputs and supports several taint propagation functions. Like libdft and LIFT, DyTan uses a bit vector to represent taint sets so the number of unique sources it supports is fundamentally limited by the length of the representation. TaintPipe [60] uses a symbolic taint analysis to convert segments of machine code to the symbolic taint operation. The symbolic representation allows them to support a larger number of inputs compared with the previous systems' bit-level representation. It is unclear whether symbolic tracking can scale efficiently to millions of labels and dependencies. Other prior work has sped up taint tracking using parallelization [63, 80, 47], but they only support a limited number of taint sources.

The merge log is derived from the log optimization proposed by Ruwase et al. [80].

Ruwase et al. and the merge log both offer a compact way of representing taint sets. Ruwase et al. use a symbolic representation of taint operations, which is only resolved when the taint of a location is needed. The merge log keeps a concrete representation of all taint sets as the program executes and resolves the taint set of a location by traversing the merge log. Our enhancements to the merge log allows us to support the tracking of millions of input sources. The merge log structure is a flat data structure of a DAG, which has been used in other applications such as the binary decision diagram [17].

Deterministic replay is a well-studied technique for faithfully reproducing the execution of programs. It has been used for debugging data races [5, 70, 92] and intrusion detection [33]. Our deterministic replay implementation has the unique ability to cheaply record an uninstrumented execution and later replay the execution using Pin. We leverage deterministic replay in order to reduce online overhead. Some previous taint tracking systems have used the idea decoupling of execution from the DIFT analysis in order to accelerate taint tracking for the same purpose [47, 60].

5.2 The Mystery Machine

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 [82], distributed systems [12], and Internet and mobile applications [76, 95].

Sherlock [11] 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 [99] 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 [81] and diagnosing software errors [35, 73]. *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 [54] 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 [95] 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 [37] 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. [26] 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 [25] 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 [88] 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. [18] 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 [5, 97, 96] and debugging performance issues [61, 83].

5.3 Data Quality Tradeoffs

Although there is an extremely rich history of quality-of-service tradeoffs [15, 65, 85] and approximate computing [10, 20, 44, 43, 84, 86] in software systems, our work focuses specifically on using the causal propagation of request information and data provenance to make better data-quality tradeoffs in low-level software components. Our study revealed the need for such an approach: existing Facebook services make mostly reactive tradeoffs that are suboptimal due to limited information. Our evaluation of DQBarge showed that causal propagation can substantially improve both request performance and response quality.

Many systems have used causal propagation of information through distributed systems to trace related events [13, 21, 24, 36, 57, 76, 77, 87]. For example, Pivot Tracing [57] propagates generic key-value metadata, called baggage, along the causal path of request processing. DQBarge uses a similar approach to propagate specific data such as provenance, critical path predictions, and load metrics.

DQBarge focuses on data quality tradeoffs in Internet service pipelines. Approximate Query Processing systems trade accuracy for performance during analytic queries over large data sets [1, 2, 9, 41, 51]. These systems use different methods to sample data and return a representative answer within a time bound. BlinkDB [2] uses an error-latency profile to make tradeoffs during query processing. This is similar to the performance and quality models that DQBarge applies to Internet pipelines.

Some Internet services have been adapted to provide partial responses after a latency deadline [40, 46, 51]. They rely on timeouts to make tradeoffs, whereas the tradeoffs DQBarge makes are proactive. PowerDial [44] adds knobs to server applications to trade performance for energy. These systems do not employ provenance to make better tradeoffs.

CHAPTER VI

Conclusion

This chapter describes directions for future work and summarizes the contributions of this thesis.

6.1 Future work

In this thesis we focused on scaling causality analysis at both a fine-grain instruction level for a single process and for a large-scale distributed system with hundreds of software components. We next discuss areas of future research for continuing scaling DIFT and leveraging causality for debugging and troubleshooting.

6.1.1 Parallelizing DIFT

In chapter II, we describe a method for scaling taint tracking to millions of input sources. However, our method only utilizes a single CPU core; parallelizing DIFT is difficult since each step could have a large set of prior dependencies. There has been previous work done in parallelizing DIFT [80, 63]. However, these systems only scale to one computer and still do not provide the level of interactivity required to run DIFT queries for long running, complicated programs. Scaling DIFT to a cluster of machines would make queries interactive. Researchers are actively working on parallelizing DIFT [75].

6.1.2 Personalized performance

Our results from chapter III and chapter IV show that causal information yield optimizations that lead to better performance. We examined how causality can improve data quality tradeoffs. As future research, we propose using causality to introduce client-specific optimizations that improve performance. We hypothesize that using client-specific information can lead to client-specific performance optimizations.

As our studies have shown, users of Internet services are increasingly diverse in the networks and platforms they employ. These networks and platforms have varying characteristics. For example, mobile networks are poorer than broadband connections. Different browsers have various Javascript interpreters. Increasingly web applications have been reliant on Javascript to provide an interactive and content-rich experience, but browsers have different Javascript interpreters with different performance variances. We propose exploring different client-side optimizations. For example, Internet services can produce Javascript content that is specific to a user's platform and/or mobile network. If a user is currently on a low-bandwidth network, the server can choose to send low-bandwidth updates in order to provide a better experience. If a user is low on energy, the server can choose to send updates that require less energy to display. We could extend DQBarge by supporting these models and their possible optimizations.

6.1.3 Identifying data quality tradeoffs

Our study in chapter IV relies on a combination of manual and automated analysis in order to identify and understand data-quality tradeoffs. As future research, we propose automating this method for two purposes. First, an automated means of studying data quality tradeoffs gives operators of all software services and understanding of how their services are used. Second, automation allows for these studies to be continually done over time to understand how data quality tradeoffs change as systems evolve.

Using the categorizations from our study, we can train classifiers that statically analyze code in order to identify existing data-quality tradeoffs in the code. This would involve deriving a symbolic representation of the categorizations and match the code to these patterns. For example, data quality tradeoffs that return a default value should be match the same category even if they return different default values. However, there are challenges in using static analysis as it can be prone to a high false positive rate. Therefore, we plan on exploring automated means of verification, such as sampling dynamic executions and evaluating the symbolic representation to verify the correct classification.

6.2 Contributions

This thesis shows methods for scaling causality analysis for debugging and optimization in programs and large-scale Internet services. First, we showed the ability to scale fine-grain causality analysis to millions of input bytes. We demonstrated this by scaling intraprocess taint tracking to millions of input bytes. This allows for the ability to track the provenance of data from a particular output byte to any particular set of input bytes. Next, we demonstrated how to scale causality analysis to large-scale Internet services with hundreds of software components. We hypothesized causal relationships between software components and rule out relationships by leveraging the scale of these services and observing counter-examples. Using this technique, we showed how we can use these causal relationships to understand performance optimizations. Finally, we explored a new dimension that combines causality with data quality tradeoffs. We conducted a study of data quality tradeoffs of a production system at Facebook. We found that most data quality tradeoffs are reactive and suboptimal. To address these issues, we developed DQBarge, a system that uses causal propagation of information in order to make better data quality tradeoffs.

We showed how we can make better data quality tradeoffs in situations such as load spikes, utilization of spare resources, and dynamic capacity planning.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[3] 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.

[4] Tankut Akgul, Vincent J. Mooney III, and Santosh Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, 2004.

[5] 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, October 2009.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, 2014.

[7] 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.

[8] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[9] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[10] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[11] 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.

[12] 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.

[13] 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.

[14] `https://datamarket.azure.com/dataset/bing/search`.

[15] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for internet services. Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation, 2005.

[16] 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. San Jose, CA, June 2013.

[17] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computing*, 35(8), August 1986.

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

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 209–224, December 2008.

[20] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptablility properties of relaxed nondeterministic approximate programs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation.*

[21] 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.

[22] Haibo Chen, Xi Wu, Liwei Yuan, Binyu Zang, Pen chung Yew, and Federic T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[23] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[24] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the 32nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, June 2002.

[25] 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.

[26] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. Hong Kong, China, August 2013.

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2011.

[28] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 1–14, June 2008.

[29] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, pages 22–22, 2004.

[30] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.

[31] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, July 2007.

[32] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[33] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[34] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[35] 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.

[36] 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.

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

[38] Jim Gray. Why do computers stop and what can be done about it? In *RELDSDB*, 1986.

[39] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[40] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: Scheduling interactive services with partial execution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC '12)*, 2012.

[41] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, 1997.

[42] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-based Protection using Demand Emulation. In *Proceedings of the 1st ACM European Conference on Computer Systems*, 2006.

[43] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[44] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, California, March 2011.

[45] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1991.

[46] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '13)*, 2013.

[47] Kangkook Jee, Vasileios P. Kermerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[48] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[49] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[50] 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.

[51] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold 'em or fold 'em? aggregation queries under performance variations. In *Proceedings of the 11th ACM European Conference on Computer Systems*, 2016.

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

[53] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[54] 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.

[55] Benjamin Livshits. Dynamic taint tracking in managed runtimes. Technical report, Microsoft Research, 2012.

[56] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[57] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[58] 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.

[59] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 141–150, 1988.

[60] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th Usenix Security Symposium*, Washington, D.C., August 2015.

[61] 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.

[62] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[63] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[64] 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.

[65] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.

[66] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles.*

[67] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4rd USENIX Symposium on Internet Technologies and Systems*, March 2003.

[68] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, 1988.

[69] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. Php aspis: Using partial taint tracking to protect against injection attacks. In *WebApps'11*, 2011.

[70] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 177–191, October 2009.

[71] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for determinisrtic replay and reproducible analysis of parallel programs. In *Proceedings of the 2010 IEEE/ACM International Symposium on Code Generation and Optimization*, March 2010.

[72] Perl: Taint mode. `http://perldoc.perl.org/perlsec.html#Taint-mode`.

[73] 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.

[74] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, Orlando, FL, 2006.

[75] Andrew Quinn, David Devecsery, Jason Flinn, and Peter M. Chen. JetStream: Cluster-scale parallelization of information flow queries. In *In submission*.

[76] 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.

[77] 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.

[78] 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.

[79] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*.

[80] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[81] 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.

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

[83] 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.

[84] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power consumption. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[85] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002.

[86] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Ricard. Managing performance vs accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[87] Benjamin H. Sigelman, Luiz Andr 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, Inc., 2010.

[88] Mukarram Bin Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with wise. Seattle, WA, August 2008.

[89] `http://trec.nist.gov/`.

[90] 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.

[91] 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.

[92] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2011.

[93] Guru Venkatarami, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *14th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[94] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Annual Computer Security Applications Conference (ACSAC) 2008*, 2008.

[95] 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.

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

[97] 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.

[98] 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.

[99] 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.