PERFDB + PERFML: ENABLING BIG DATA-DRIVEN RESEARCH ON FINE-GRAINED PERFORMANCE PHENOMENA

A Dissertation
Presented to
The Academic Faculty

by

Joshua M. Kimball

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2021

PERFDB + PERFML: ENABLING BIG DATA-DRIVEN RESEARCH ON FINE-GRAINED PERFORMANCE PHENOMENA

To my wife, Dena, and daughters, Nadia and Nessa

ACKNOWLEDGEMENTS

I am extremely grateful to my advisor, Dr. Calton Pu, who guided me tirelessly throughout my PhD. I am fortunate to have worked with him. He taught me how to be a better researcher. In doing so, he helped me diagnose my own "bugs" to become a more productive and efficient person. He helped me become a better Josh.

I would also like to give special thanks to the members of my dissertation committee—Dr. Ling Liu, Dr. Shamkant Navathe, Dr. Joy Arulraj, and Dr. Qingyang Wang—for serving on my dissertation committee. I very much appreciate the time and effort they took to supervise my dissertation work.

The work in this dissertation is the result of collaboration with my previous and current colleagues in the Elba project: Rodrigo Alves Lima, Chien-An Lai, Tao Zhu, Deepal Jayasinghe, and Aibek Musaev. They have supported me all along the way. It has been a privilege to work with such amazing colleagues.

Lastly, I want to thank my wife, Dena, and daughters, Nadia and Nessa. They gave me the energy and inspiration to endure and always helped me to remember the bigger picture, to prioritize wisely and laugh along the way. Finally, I would like to thank my parents, Phillip and Patricia, my sisters, Shana and Stephanie for their encouragement and examples of bravery, and I would like to thank my in-laws, Diana and Arthur, for their support and enthusiasm along this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

CTQO	Cross-Tier Queue Overflow
DVFS	Dynamic Voltage Frequency Scaling
JVM	Java Virtual Machine
LLR	Less Long Requests
PiT	Point-in-Time Response Time
VLRT	Very Long Response Time Requests

SUMMARY

The long-tail latency problem is a well-known problem in large-scale system topologies like cloud platforms. Long-tail latency can lead to less predictable system performance, degraded quality of experience and potential economic loss. Previous research has focused on coarse-grained, symptomatic treatments like redundant request executions to mitigate tail latency and its effects. Instead, we propose studying these performance bugs systematically and addressing their underlying root cause.

The millibottleneck theory of performance bugs provides a testable hypothesis for explaining at least some requests comprising the latency long tail. The theory posits that transient performance anomalies cause a non-negligible number of requests to complete in seconds, called Very Long Response Time Requests (VLRT), instead of tens of milliseconds like the vast majority of other requests.

In this dissertation, we enable the systematic evaluation of the millibottleneck theory across a big data-scale experimental data collection. First, we present *perftables*, a performance log parser, that extracts resource monitoring data across a wide variety of hardware and software configurations. Secondly, we use our data management system, *PerfDB*, to load and integrate fine-grained system performance data from approximately 400 experiments. We conduct the first-generation population study of VLRT, and our data support millibottlenecks inducing VLRT through CTQO (Cross-Tier Queue Overflow). We also enable the study of a second latency class called Less Long Requests (LLRs). Finally, we present our ensemble-based, supervised machine learning system, *PerfML*, that handles data characterized by heterogenous feature space and hierarchical, imbalanced classes—

characteristics inherent to the data needed to study millibottlenecks and latency performance bugs. The analytics results from PerfML demonstrate its ability to isolate different kinds of millibottlenecks across a range of systems and configurations with high recall and acceptable precision.

CHAPTER 1. INTRODUCTION

Internet-scale, cloud-based platforms like ecommerce or social networking platforms, which experience "bursty" workloads, confront a well-known performance pathology termed the "long-tail latency problem" [1]. Dean et al. shows servers with 99th-percentile latencies of 1s used in tandem, (i.e., 100 servers), to service user requests result in 63% of user requests taking longer than 1s to complete. They also report "the slowest 5% of requests across Google's services accounted for *half* of the total 99th-percentile latency," i.e. an additional 70ms of latency is added yielding 140ms in total [1].

Long-tail latency has also been reported to be bad for business. For example, Amazon has reported [2] that a marginal increase of 100ms in page load time correlates with approximately a 1% reduction in sales. Google found that 500ms of additional delay returning search results could reduce revenue by up to 20% [3]. Given the size of their respective customer bases, these platforms need to reduce their 95th, 99th and 99.9th percentile latencies as close to zero as possible [1] [4].

The millibottleneck theory of performance bugs [5] provides a testable hypothesis to explain some long-tail latency requests. The theory posits that transient performance anomalies cause a non-negligible number of requests to complete in seconds, called Very Long Response Time Requests (VLRT), instead of tens of milliseconds like most other requests. This occurs despite low average resource utilization (e.g. less than 50%)—adding to the puzzle of long-tail latency requests. Millisecond-scale delays due to transitory resource bottlenecks, called millibottlenecks, can propagate through a system via intra-system, synchronous communication mechanisms like Remote Procedure Calls (RPCs), ultimately leading to performance bugs like VLRTs [5]. We have isolated root causes of millibottlenecks crossing the architectural stack, including CPU dynamic voltage frequency scaling (DVFS) control [6], Java garbage collection (GC), “noisy neighbors” due to virtual machine consolidation [7], dirty page flushing [8], memory thrashing due to rapid succession of page faults [9] and writing log data to disks [10] [11].

To identify and study this diverse set of millibottlenecks, we have used our experimental computer science infrastructure to execute hundreds of experiments at fine-grained time scales across a variety of system configurations [12, 13, 14, 15, 16, 17]. Our infrastructure with its fine-grained, precise event tracing and expansive resource monitoring provides the monitoring data that is necessary for detecting millibottlenecks. To isolate millibottleneck periods, this event and resource data needs to be integrated across space and time.

Generally, web-facing applications running in a cloud have (potentially infinite) scalability and heterogeneity requirements. Consequently, our monitoring data is very complex to integrate due to this variety of n-tier system configurations. We categorize these data heterogeneity challenges as follows:

- *Performance Log Diversity*. As we show later, performance monitoring logs can vary across executions despite controlling for system topology due to orthogonal considerations such as OS and other machine hardware-level characteristics. Said differently, resource monitors’ performance logs can have different resource schemas.

- *Experimental Schema Diversity.* Secondly, we demonstrate later that our infrastructure covers an enormous experimental configuration space resulting in many experiments with different topologies, components and experimental parameter ranges. In fact, we have found hundreds of different experimental schemas across just a few hundred experiments. Consequently, to study millibottlenecks *across experiments*, we must integrate these constituent experiments diverse event and resource schemas—a non-trivial undertaking.

- *Millibottleneck Diversity*. Finally, we have identified a range of different millibottlenecks, evidenced by the earlier list. Each type of millibottleneck has its own set of data or signals that are used for diagnosis. Our prior studies unit of analysis was at the individual period level, so they only included the data needed to analyze each type of millibottleneck—a significant data analysis simplification. Analyzing periods *across an entire experimental collection* requires evaluating all relevant signals, since we do not know *a priori* which of these signals might apply to a particular period. An effective analysis technique needs to be extensible to support a large (and growing) set of signals (features). It also needs to be sensitive enough to identify the same type of millibottleneck over a range of systems, including those with different topologies, components, resources, workloads and configurations.

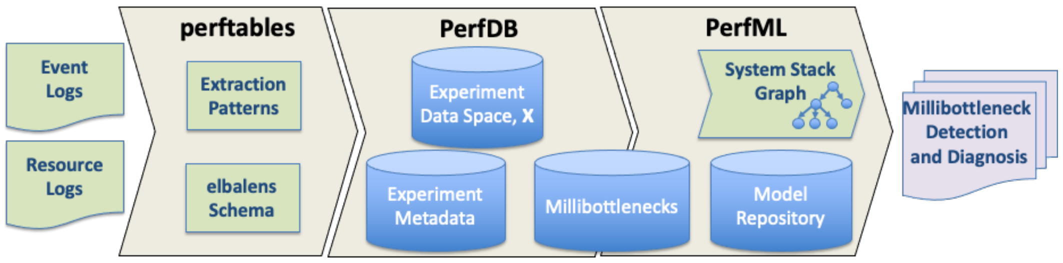


Figure 1 – PerfML system data flow. Event and Performance Monitoring Logs are ingested and interpreted by perftables, which automatically induces relations. This relational data is persisted in PerfDB, which provides just-in-time data integration and materializes the Millibottleneck schema. PerfML leverages PerfDB relations and system model domain knowledge to train ensemble-based machine learning models to (i) isolate millibottleneck periods (ii) diagnose the type of millibottleneck respectively.

The focus of this dissertation is on such an analytical system. As shown in Figure 1, PerfML features a pipeline-based design. We employ a lazy, flexible approach to reliably extract data from a variety of performance monitor logs into experiment-specific relations. We materialize a data store, including important experiment-specific metadata, and provide a query façade over these experiment tables. Lastly, we use a *one-versus-all (OvA), ensemble-based* machine learning approach to detect and diagnose known millibottlenecks and their latency-related pathologies like VLRT.

1.1 Thesis Statement

Experimental computer science big data spanning hundreds of diverse systems' experimental schemas present significant challenges and opportunities to apply ensemble-based supervised machine learning and lazy data extraction and integration techniques for analyzing a variety of millibottlenecks and associated long-tail latency phenomena.

1.2 Contributions

To support this thesis statement, we make the following three contributions:

- Our first and most technical contribution and our first paper is the beginning of the pipeline—perftables. While the contribution is narrow, it provides an important building block for the rest of the pipeline. Our measurement toolkit's measurement data outputs are the PerfML pipeline's inputs, specifically inputs to perftables. perftables can parse, extract, transform, and load a collection of fine-grain (sampling at 50ms intervals) resource monitoring data (Figure 20) on a wide variety of hardware and software configurations (Table 4).

- Our second and broadest contribution is PerfDB. PerfDB interprets and loads fine-grained event (timestamped arrival and departure of every message at every server) data in addition to the resource data extracted with perftables. We demonstrate that our approach was able to clean, integrate and analyze approximately 2TB of performance monitoring data spanning ~400 experiments into aggregated data sets, enabling the confirmation of the association of VLRT and millibottlenecks causing CTQO and the identification of a latency class called Less Long Latency Requests (LLRs). We confirm the previously known association between transient VLRT requests and millibottlenecks with a large-scale study. Furthermore, our data support the explanation of CTQO (Cross-Tier Queue Overflow) [18] as cause of transient VLRT requests in our experiments. We also employ a detailed statistical analysis to identify LLRs that have duration between VLRT request and the average latency. LLRs are typically associated with millibottlenecks of shorter duration than those causing CTQO.

- Our final and most impactful, methodological contribution is PerfML. It consists of an ensemble-based supervised machine learning approach designed to handle data characterized by heterogenous feature space and hierarchical, imbalance classes—data characteristics inherent to detecting and diagnosing millibottlenecks and related performance anomalies like VLRTs and LLRs. We evaluate our ensemble-based supervised machine learning approach using a diverse set of approximately 30K millibottlenecks found across ~400 experiments.

CHAPTER 2. ELBA-WISE MEASUREMENT TOOLKIT

2.1 Motivation

In this chapter, we briefly outline the constituent parts of our toolkit. The diversity of our measurement toolkit's outputs serves as the principal motivation for our pipeline. We begin by explaining the factors that explain the data heterogeneity our pipeline must contemplate.

First, our measurement toolkit covers an enormous experimental configuration space. Using a declarative approach, it can execute many different types of experiments consisting of different topologies, components and experimental parameter ranges. In fact, we have found hundreds of different experimental schemas spanning just a few hundred experiments as we later show in sections 4.2 and 4.3.4.

Secondly, we have isolated millibottlenecks across a range of resources and for each type of component software. Per the first point, our measurement toolkit needs to cover an enormous experimental space to study each of these effectively. Different kinds of millibottlenecks means that different signals (or variables) are required to isolate each one. Consequently, we need different data to diagnose the various types of millibottlenecks.

Finally, the resource logs output by our toolkits' resource monitors, which provide a subset of the variables, have complicated formats or layouts. Orthogonal concerns can affect the layout such as: system topology, types of components, monitoring program (the type, version, and bootstrap parameters), hardware or computer architecture, operating system and kernel version of the host, and experimental benchmark application-specific parameters. For example, running the same monitor holding all else constant except on different machines can result in different layouts due to hardware-specific or architectural characteristics like the number of block devices, CPU cores or number of caches. Examples and a more detailed explanation of these orthogonal concerns appear later in sections 2.1 and 2.2.

2.2 Elba-WISE Measurement Toolkit Description

Our measurement toolkit provides the special monitoring instrumentation that has enabled the fine-grained measurement of millibottlenecks in our experiments. Specifically, its constituent parts fulfill three important requirements for studying millibottlenecks and latency pathologies: low overhead instrumentation and monitoring, fine-grained (resource) samples and capturing and important system events reconstruction. By meeting these design goals, our measurement toolkit provides the necessary data for studying millibottlenecks and the interesting events they induce like queue overflows and VLRTs.

Fine-grained and expansive resource monitoring. Studying millibottlenecks pose a resource monitoring challenge due to their short life spans and variety. According to the Sampling Theorem, multi-second sampling periods are insufficient to study transient phenomena effectively. (For example, we can reliably detect millibottlenecks of 100ms or longer with 50ms sampling intervals.) We also need to monitor as many performance counters (resources) as possible using a diverse set of resource monitors.

Precise event tracing. Previous research has shown that millibottlenecks cannot be detected using only hardware resource utilization [19]. To identify and confirm the root cause of millibottlenecks, we need to study the significant system events that occur during them. We decorate the boundaries of systems' components using specialized component software to record interesting events during experimental workload execution, (i.e. request arrivals and departures via remote procedure calls across nodes), in these components' native event logs. This instrumentation also enables us to capture the queueing-related propagation mechanisms associated with millibottlenecks and reconstruct the execution paths of pathological requests like VLRTs across a distributed system.

Low overhead monitoring and logging. A measurement toolkit needs to limit the monitoring overhead from obscuring the subject under study (i.e. the system processing a given workload). Our careful implementation of open-source resource monitors enables us to bypass any conflict among the previously described fine-grained monitoring requirements and this low overhead requirement.

Benchmarks. In experimental computer science research, researchers typically use (application) benchmarks to evaluate the performance of said benchmark under different system configurations and workloads. Using benchmarks enables the systematic of millibottlenecks by enabling us to formulate and validate hypotheses through the collection of compelling evidence. Besides leveraging benchmarks, our measurement toolkit uses a declarative framework like a workflow modeling language to facilitate the benchmark's deployment and ensuring its dependencies are fulfilled. Ensuring complex benchmarks can be constructed, executed and reproduced is important to studying phenomena like millibottlenecks where some kinds can be due to non-deterministic systems execution like Java garbage collection. A declarative approach also enables us to efficiently extend the benchmark's configuration space to support: different public cloud infrastructures, multiple operating systems and kernels, various component software and additional monitors. This declarative framework through an integrated workload generator and benchmark-specific configuration files controls the “bursty workloads” our application benchmark confronts. Bursty workloads, which are typical of web-facing applications, have been shown to cause short resource saturations [18]. As such, an application benchmark should have an integrated workload generator that offers fine-grained controls over the workload size, its characteristics (e.g. request type and frequency) and the burstiness (e.g. variation in the number of requests per second) to re-create these kinds of millibottlenecks.

2.3 Studies of Individual Experiments

Our measurement toolkit has enabled us to study individual millibottleneck periods effectively, evidenced by the previous studies cited in Chapter 1. To study these individual instances, we have employed experiment-specific parsers to extract and integrate the necessary event and resource data. Then, we used graphical and foundational statistical techniques like correlation to “zoom into” these individual, anomalous periods.

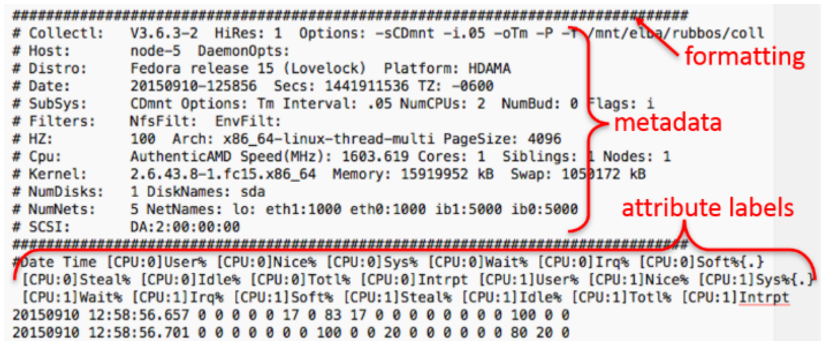
However, the inherent complexities of the data have limited our ability to conduct population studies. To study millibottlenecks as a group, we need techniques to process and integrate the heterogenous performance data output by the measurement toolkit. As such, the data outputs of the measrurement toolkit are the inputs to our PerfML pipeline. Specifically, these outputs are the inputs to perftables—our flexible performance log data extraction, parsing component. Perftables is the beginning of the performance data pipeline. It induces relations for resource data. PerfDB ingests the induced resource data relations and loads the event data using a defined event measurement schema. PerfML performs the analytics on data loaded in PerfDB. Once PerfML's ensemble-based machine learning models have been trained, we use them to identify millibottlenecks and millibottleneck-induced latency pathologies in newly loaded datasets.

CHAPTER 3. PERFTABLES

3.1 Introduction

Our experimental computer science infrastructure, *elba*, generates huge volumes of data from large numbers of diverse experiments and systems topologies, which support our empirical-based method for understanding computer systems' more fundamental behavior. As we show later, we have run over 20,000 experiments on *elba* over the last three years generating over 100TB of data spread across 400K various log files. To isolate and diagnose nuanced, fine-grained performance anomalies, we need to support a broad array of experimental configurations, since these bugs can materialize under a range of conditions. For example, experimental artifacts like logs can vary in number and layout *per experiment* making data extraction and subsequent analysis challenging to perform at scale. Recent approaches like DeepLog operate over arbitrary text and attempt to isolate "macro-level" system events like crashes [20]. Our automated relation induction approach, *perftables*, operates over the diverse performance monitoring outputs with the objective of isolating much more precise (shorter and transient) events.

The layout diversity observed across these performance logs stems from our infrastructure's enormous experimental parameter space and its diversity of instrumentation. Resource monitoring is one particularly good illustration. Elba infrastructure currently features five resource monitors: *iostat*, *systat* (*sar*), *collectl*, *oprofile* and *lockmon*. Each execution (of a given monitor) can have very different output even though each of these programs accepts a fixed number of parameters. For example, toggling a runtime parameter to change the resources being monitored alters the layout of the monitor's log file. Assuming each resource monitoring decision is binary, there can be as many as 2^n possible layouts for a performance monitor capable of measuring up to n resources. (From this point forward, layout and format are used interchangeably.) Given this, the number of possible layouts is exponential in the number of resources being monitored. This makes a naïve approach of writing a parser for each unique format simply intractable. In our data set, we have found the number of distinct layouts to number in the hundreds (under the most conservative accounting). Data variety and volume at our scale impedes automated data extraction and subsequent data analysis, creating an enterprise data-lake-scale data management challenge for our infrastructure [21]. The longer data remains unprocessed, the more unwieldly its management becomes [22].



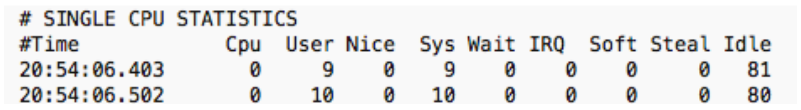
```
####################################################################################
# Collectl:   V3.6.3-2  HiRes: 1  Options: -sCDmnt -i.05 -oTm -P -T /mnt/elba/rubbos/coll
# Host:       node-5  DaemonOpts:
# Distro:     Fedora release 15 (Lovelock)  Platform: HDAMA
# Date:       20150910-125856  Secs: 1441911536 TZ: -0600
# SubSys:     CDmnt Options: Tm Interval: .05 NumCPUs: 2  NumBud: 0 Flags: i
# Filters:    NfsFilt:  EnvFilt:
# HZ:         100  Arch: x86_64-linux-thread-multi PageSize: 4096
# Cpu:        AuthenticAMD Speed(MHz): 1603.619 Cores: 1  Siblings: 1 Nodes: 1
# Kernel:     2.6.43.8-1.fc15.x86_64  Memory: 15919952 kB  Swap: 1050172 kB
# NumDisks:   1 DiskNames: sda
# NumNets:    5 NetNames: lo: eth1:1000 eth0:1000 ib1:5000 ib0:5000
# SCSI:       DA:2:00:00:00
####################################################################################
#Date Time [CPU:0]User% [CPU:0]Nice% [CPU:0]Sys% [CPU:0]Wait% [CPU:0]Irq% [CPU:0]Soft%{.}
 [CPU:0]Steal% [CPU:0]Idle% [CPU:0]Totl% [CPU:0]Intrpt [CPU:1]User% [CPU:1]Nice% [CPU:1]Sys%{.}
 [CPU:1]Wait% [CPU:1]Irq% [CPU:1]Soft% [CPU:1]Steal% [CPU:1]Idle% [CPU:1]Totl% [CPU:1]Intrpt
20150910 12:58:56.657 0 0 0 0 0 17 0 83 17 0 0 0 0 0 0 0 0 100 0 0
20150910 12:58:56.701 0 0 0 0 0 0 0 100 0 0 20 0 0 0 0 0 0 80 20 0
```

Figure 2 – Example of collectl output for multi-core CPU machine

Example 1.1. Most previous work assumes record boundaries have been established beforehand or can be easily established using repeated patterns found in explicit structures such as the HTML DOM tree. As Gao et al. explain, log files have no natural record boundaries or explicit mechanisms like HTML tags for determining them [23]. In addition, log files can have nested structures and variable length records, i.e. records which span a variable number of rows. Log files also include noise such as formatting concerns and various metadata as shown in Figure 2.

Performance logs present some specific and unique challenges. First, performance logs output formats are impacted by two implicit factors: the computer architecture of the system components being monitored and the actual behavior of the system under study. This latter characteristic suggests layout is at least partially runtime dependent, thus the layout of a given performance log for a given execution is not known a priori.

Example 1.2. Figure 2 and Figure 3 shows the performance output from the same performance monitor bootstrapped with the same monitoring parameters but running on different systems. Figure 2 displays the output for the multicore system while Figure 3 depicts the single core system. Clearly, the output is significantly different holding all else constant. The impact of these implicit factors on the layout of the output underscores the need for an unsupervised approach.



```
# SINGLE CPU STATISTICS
#Time           Cpu  User Nice  Sys Wait IRQ  Soft Steal Idle
20:54:06.403      0     9    0    9    0    0    0     0   81
20:54:06.502      0    10    0   10    0    0    0     0   80
```

Figure 3 – Example of collect output for single-core CPU machine

Secondly, performance logs often contain multiple, related record types. In addition, record types can have degenerative sub-structures such as variable length attributes. This characteristic only adds to the complexity of comparing records once they are found. Prior work has assumed records are independent, so this work contains no mechanism for evaluating the relationships among records. This step is critical to realizing an end-to-end unsupervised approach. Information must be able to be extracted and directly represented in relations.

Example 2. Figure 5 shows a snippet of a log file containing process and context switch data from two sampling periods. First, notice that each active process appears on a separate line. Since each sampling period has a different number of active processes, each sampling period spans a variable number of lines. Also, the sampling period is the record in this case. Under a record type independence assumption, each region of data, i.e. the regions containing data about context switches and processes respectively, would be treated as their own record types. In this case, the two sampling periods not the data regions constitute the two record structures, which also happen to span a variable number of rows, i.e. variable length records.

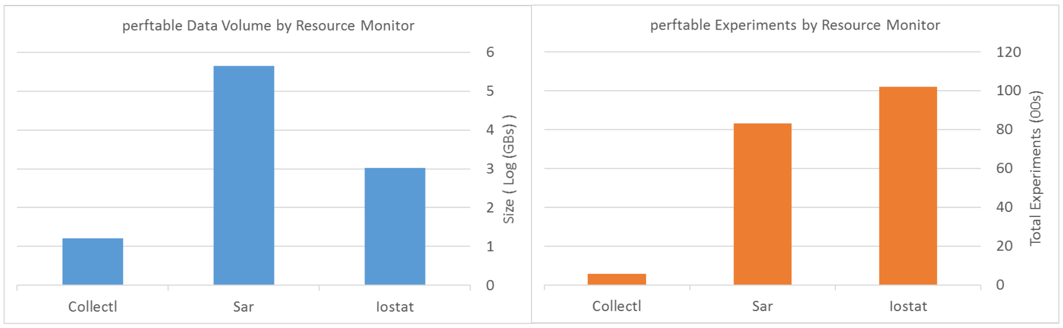


Figure 4 – perftables Usage. Left graph shows the amount of experiment data extracted by perftables for each monitoring program. Graph on the right shows the number of experiments by monitoring program where perftables was used.

While assuming independent record types is suitable for simple extraction, it is impractical at our scale. In our case, once the data has been extracted, it would still require significant transformation to get it into the correct relational form. This last example also demonstrates the need for an approach to identify record boundaries over a (potentially large) variable number of rows due to the impact of runtime factors can have on the layout.

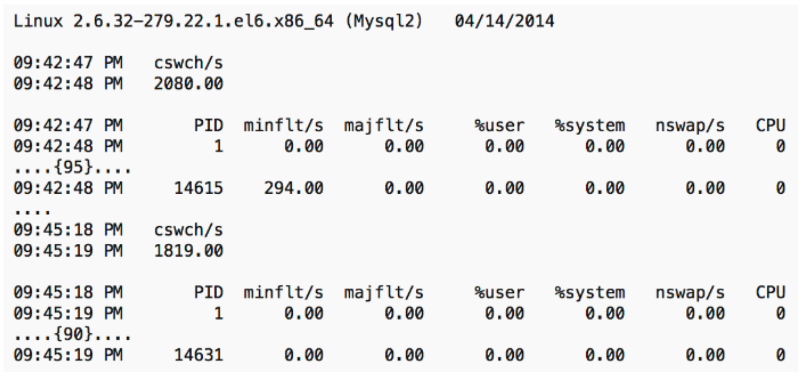
Approach Overview. In this paper, we present *perftables*—our unsupervised algorithm for automatically inducing relations directly from performance monitoring log files. Our method goes beyond extraction as our unsupervised approach constructs tables directly from the observed data.

To accomplish this objective, we have defined a small set of pattern-based templates. We use a set of delimiters to first convert text into tokens. Then we transform each token sequence into a sequence of data type labels by applying a series of data type functions to each token. Next, we lazily match these sequences of data labels to one of our templates based on similarity. Once data has been matched to a template, the template can be used to extract the data and separate semantically meaningful metadata from "noise." To detect record boundaries, we induce a graph over the matching template instances. Finally, we construct relations from the template-matched data according to the record structure detected in the graph.

Generally, our method differs from previous work in its ability to handle logs with runtime dependent layouts. Due to the impact of runtime behavior on log output, multiple record types and variable length records are particularly prevalent in performance logs. For example, we have observed over 100 distinct layouts generated from several distinct runtime configurations across 3 monitoring programs. Specifically, *perftables* does not depend on pre-defining record boundaries. Moreover, it does not assume record boundaries appear over some constant, fixed number of lines. Its lazy approach obviates the need for such a hyperparameter.

Secondly, our method goes beyond extraction and induces relations directly from the log text data. Previous work has relegated schema definition and data transformation to manual post-extraction tasks—a significant burden at our infrastructure's scale. To analyze experimental data at our scale, we require an approach that can extract and transform unstructured log data into structured (relational) data with as little human supervision as possible. As Figure 4 shows, we have used *perftables* to successfully extract with more than 98% accuracy over 250 GB of data from over 1 TB of log data.

In this paper, we demonstrate how our approach efficiently, accurately and automatically identifies and extracts relations from performance log files. Specifically, we have developed a small set of layout pattern-based templates, which support data extraction and attribute identification. Secondly, we have developed a set of algorithms to automatically identify record boundaries even in the presence of irregular and variable length records. We also show how our templates support automatically defining relations from matching data. Finally, we demonstrate the effectiveness (accuracy and efficiency) of our templates inside our environment and provide coverage for performance log data beyond our domain.



```
Linux 2.6.32-279.22.1.el6.x86_64 (Mysql2)   04/14/2014

09:42:47 PM   cswch/s
09:42:48 PM   2080.00

09:42:47 PM       PID  minflt/s  majflt/s     %user   %system   nswap/s   CPU
09:42:48 PM         1      0.00      0.00      0.00      0.00      0.00     0
....{95}....
09:42:48 PM     14615    294.00      0.00      0.00      0.00      0.00     0
....
09:45:18 PM   cswch/s
09:45:19 PM   1819.00

09:45:18 PM       PID  minflt/s  majflt/s     %user   %system   nswap/s   CPU
09:45:19 PM         1      0.00      0.00      0.00      0.00      0.00     0
....{90}....
09:45:19 PM     14631      0.00      0.00      0.00      0.00      0.00     0
```

Figure 5 – Long variable length records. Intermediate rows, indicated by braces, were removed for space considerations.

3.2 Terminology and Problem Statement

In this section, we will formally define our problem of unsupervised table extraction from performance log files.

3.2.1 Definition – Data Type Sequence

By applying one or more delimiters to a string, it can be transformed into a sequence of tokens. This process is typically referred to as tokenization.

A best-fitting data type description can be estimated for each token by applying a data type function to each one. For example, if a token consists of the characters "123" then a data type description function might return "INT" to indicate integer as the best-fitting data type for this character sequence. By applying a data type function to every token in a sequence, a sequence of data type labels can be constructed. We refer to this sequence of labels as a data type sequence for short.

3.2.2 Definition – Layout Template

A Layout Template is a regular expression for data type sequences. We say the data type sequence matches a layout template *iff* the regular expression of a layout template matches the string form of a data type sequence.

3.2.3 Definition – Layout

A Layout is a specific arrangement of data. Formats or layouts like those depicted in the previous Figures use formatting characters like whitespace and other special character delimiters like "#" or ":" and the order of metadata and data and their orientation to accomplish two objectives: partition data from metadata and metadata from "noise" and express relationships among the data. For example, metadata which immediately precedes data can be assumed to describe the data that follows it. In short, a layout is a sequence such that order can be used to partition the sequence into data and metadata constituent parts.

Formally, a layout, L, consists of text that can be divided into rows separated by newline characters, i.e. "\n." A layout consisting of n rows is $<r_1, r_2, ..., r_n>$. Applying some tokenization function, f, to the i^{th} row r_i results in m tokens $<t_{i1}, t_{i2}, ... , t_{im}>$, and applying some function g to one or more successive t_{ij} determines its membership in M or D, the sets of metadata and data respectively.

Example 3. In the performance monitoring domain, layout explicitly encodes or aligns the measurements to corresponding resources. It expresses relationships among data visually. In Figure 5, each line expresses the relationship between time and a magnitude for each of the resources being measured. Specifically, at 20:54:06.403, the CPU utilization is 19%, i.e. 100% - Idle%. It also shows the components of this utilization: User and (Sys)tem. Since the values appear on the same line, the layout is expressing a co-occurrence between these components of utilization at time, 20:54:06.403. In the multi-core case in Figure 3, we see each CPU core (and corresponding components of utilization) are represented as separate columns. Once again, the layout expresses a co-occurrence among these cores' measurements at time 12:58:56.657. In both Figures, the preceding labels describe the data, and more specifically, that a label at a specific position corresponds to data at the same position in a subsequent row. The presence of labels provides an additional important signal. Specifically, knowing labels exist and their location in a file provides information about the location of the data they describe. Moreover, labels immediately preceding values in a tabular-like orientation suggests order can be used to match values to labels—an important signal that could be used during processing. In this respect, these files exhibit some self-describing characteristics.

3.2.4 Definition – Log Data

Consider a file F with *m* layouts <l_1, l_2, ... , l_m>. Given our layout definition, interpreting each of the file's layouts can help us separate data from metadata and segregate useful labels from other metadata. Our goal is to find a layout that most closely matches the observed data, so it can be used to extract a Table T from this data. This is a subjective goal as solutions will have a different number of tables, columns and records. We obviously want to maximize the amount of information that can be reliably extracted. Instead, we need to formulate the problem as an optimization task.

Problem. The task is to find the best fitting Layout Template or Templates given the text. Once we have matched a template to an observed layout, we can use the template to construct a table T containing some number of columns and a maximal number of rows from the matching data. So, our refined problem is to extract a table T from the given log data using the best fitting layout L so that the number of extracted tuples is maximized.

3.3 Model

In this section, we identify the layout patterns our model covers and its assumptions. Our model reduces layout patterns into sequences of coarse-grained data type labels. By casting each token to a best-fitting data type, we can begin to "see" the format patterns more explicitly. Our model also includes a collection of Layout Templates that are expressed as regular expressions over the same alphabet as the one for data type labels. Layout templates not only express data composition but also a specific ordering. These model components combined with a few other reasonable assumptions enable us to automatically extract relations from performance log data.

3.3.1 Visual Structural Cues and Layouts

Performance log files are often formatted to support human readability and comprehension. As such, humans can use visual cues provided by a file layout to easily separate data from metadata. Unfortunately, performance log text does not explicitly and consistently identify the regular structures that are visually obvious.

We can view the problem of automatically extracting data from performance log files as one of interpreting the file layout. Our task is to find a mechanism to convert the visual cues provided by the layout into something more explicit to support automated detection and extraction. As we will show, the arrangement or sequence of data types seems to sufficiently approximate the layout's visual cues.

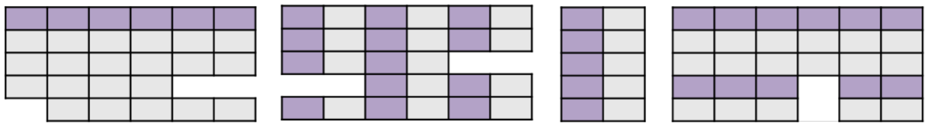


Figure 6 – Common Layouts

We previously defined layout as a sequence of metadata and data in which order can be used to differentiate data from metadata. Figure 5 depicts some of the most common layout structures appearing in performance logs. Each "row" in the figure represents a line, and each "cell's" shading indicates whether it represents a data (grey) or metadata (lavender) element. Gaps among the cells indicate breaks or irregularities, i.e. NULLs.

Each example in the figure can be described by their orientations of data to metadata: tabular, horizontal, vertical, or series of independent tabular structures. We use these graphical models as a basis for defining our collection of Layout Templates, which relate sequences of data types to data and metadata distinctions.

3.3.2 Layout Templates

To support our broader identification and segmentation tasks, we have defined a set of *data layout templates*, or *layout templates* for short, to codify each of the layouts pictured in Figure 6. Specifically, our templates are defined using regular expressions over the same alphabet used for representing data type sequences (S, D, N). We name these patterns after the basic data type sequences they describe.

Table 1 – Template Definitions

Type	Pattern
Uniform (U)	U = {S+ | (D | N)+}
Alternating (A)	A = {D?(SN)+ | D?(NS)+}
Tag (G)	G = {S(D | N)+}
Tabular (T)	U_{S_i} , $U_{N}+_{j}$ where $0 <= i < j < n$
Horizontal (H)	$A+_{i}$, $A+_{i+1}$ where $0 <= i < n$
Vertical (V)	$G+_{i}$, $G+_{i+1}$ where $0 <= i < n$
Series	$(T_i \mid H_i \mid V_i)+$ where $0 <= i < n$

For example, Figure 3 could be expressed as a sequence of four matching patterns: U_S, U_S, U_N, U_N. (A note on notation: an alphabetic subscript on a basic pattern refers to the specific branch taken in the pattern definition.)

Our templates express varying degrees of restrictiveness. This follows the intuition that the more structure or regularity the data exhibits, the more specific the matching rules can be. Accordingly, our templates can be applied using the best fit principle.

3.3.3 Assumptions

Our method makes several assumptions about files and their layouts, which enable the application of our templates. While these assumptions might appear to be restrictive, we demonstrate in our evaluation that all log files in our sample, including those collected from the wild, respect these assumptions.

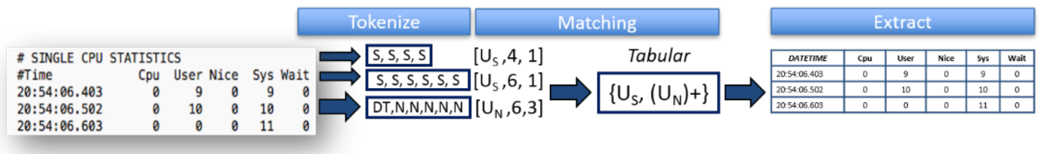
Layout Templates Coverage Assumption. This assumption makes explicit the set of files our approach covers. Our approach begins by assuming files observe a left-to-right, top-to-bottom orientation. Specifically, our method covers files that match our Series layout template. Stated differently, our method can process log files that can be expressed as an ordering of our Layout Templates.

For our method to achieve its ultimate objective to automatically recover relations, the order of data in files of interest needs to matter; order must have semantic meaning. This assumption originates from our definition of a Layout. Our coverage assumption not only restricts the potential layouts our method covers, but it also bounds the search space and limits the set of files from which we can automatically materialize relations.

Token Creation Assumption. This assumption concerns the process of applying some regularly occurring delimiter to split a text of interest into tokens. Specifically, we assume each character in a text is either used for formatting or as part of a data value. Under this assumption, a character used as a delimiter cannot also be part of a data value.

Said differently, characters used as delimiters should not split semantically meaningful data. For example, using a colon “:” as a delimiter on text containing datetime would split semantically meaningful tokens, since the corresponding date entity is now represented as a series of independent tokens. Under this assumption, a colon character needs to either be a delimiter or part of a value for a given text.

While this assumption seems to be restrictive, we add flexibility by limiting the context under which the assumption must be true. Previous work has assumed delimiter characters need to be pre-identified or apply uniformly to a file. This has typically been referred to as tokenization or chunking. In our case, we assume the context for evaluation is the text between two consecutive newline characters, i.e. a line of a file.



```
# SINGLE CPU STATISTICS
#Time            Cpu  User Nice  Sys Wait
20:54:06.403       0     9    0    9    0
20:54:06.502       0    10    0   10    0
20:54:06.603       0     0    0   11    0
```

DATETIME	Cpu	User	Nice	Sys	Wait
20:54:06.403	0	9	0	9	0
20:54:06.502	0	10	0	10	0
20:54:06.603	0	0	0	11	0

Figure 7 – perftables Approach

In our domain, whitespace is frequently used for formatting and layout purposes. From this experience, we have found using whitespace characters as delimiters usually respects this assumption.

3.4 perftables Approach

Our approach consists of four steps: tokenizing the file, matching data type sequences to layout templates, identifying candidate relations and records and finally extracting relations. Our layout templates are projected onto a file after a file has been transformed into sequences of data type labels. We create this tokenized representation by applying user-provided (or a default set) of delimiters to the file. Next, we create sequences of data type labels by inferring the best fitting coarse-grained data type for each token. We match our Layout Templates to these data type label sequences using a backtracking approach. Once data type sequences have been matched to templates, we use information from the matched data to identify candidate relations and their constituent records. Finally, we this information and the matchings to form relations.

3.4.1 Token Creation

First, a file is broken up into tokens using either a set of user-provided or default delimiters. The default set consists of white space characters, pipe (|), comma and quotation marks.

Specifically, each line can be converted into a "row of tokens" by applying one or more of these delimiters to it. After the file has been tokenized, we now consider each line of the file to be a row. Specifically, a row r with m tokens is expressed: $r = <t_0, t_1, t_2, \ldots, t_{m-1}>$.

The default delimiters are used to bootstrap or initialize our method. Users can supply supplemental delimiters; however, we have found our default set to be reliable for performance log data.

Sequences of Data Type Labels. Each token in the row can be evaluated for fit among three coarse-grained data types: DATETIME, NUMBER and STRING. We represent each token in the row with a label corresponding to the best fitting coarse-grained data type: S for STRING, D for DATETIME and N for NUMBER. At the end of this encoding step, each row is represented by a sequence of S, D and N characters. We call these sequences of data type labels sequences for short.

The next step in our method involves analyzing these patterns for the implicit semantic clues expressed in the layout. For example, a row with the sequence S, N, S, N, S, N describes a sequence of alternating STRING and NUMBER data. This layout suggests data is located at the positions corresponding to the "N" labels, and its metadata is located at the "S" label positions. (Note: the preceding sequence can be expressed by the regular expression (SN)+ which also corresponds with our Alternating pattern definition.) The next step in our approach involves evaluating these sequences by matching these data type labels to our Layout Templates' regular expressions.

3.4.2 Model

During this step, we interpret the sequences and match them to our templates to identify candidate tables. The objective of this step is to identify those rows that "belong together."

Matching Sequences to Layout Templates. After tokenizing the file, we try to match sequences of data type labels to the best fitting layout template. Not knowing a file's layout *a priori* motivates the need for a lazy, adaptive approach to matching. Accordingly, these sequences are lazily evaluated according to their topological order.

Backtracking. We match sequences to templates using a backtracking algorithm. This approach optimizes the best-fitting template through a process of elimination. We evaluate sequences according to their topological order. Each data type label sequence's string form is matched to each of the regular expressions accompanying each template definition. Only matching templates are preserved until only one remains. The process restarts once a sequence invalidates the remaining template, but not before the remaining template and its span of matching rows is added to an array of template, row span tuples. Once all rows' sequences have been matched to templates, a *table candidate* can be induced from the constituent rows corresponding to each matching template, row span tuple.

3.4.3 Extracting Relations

Besides helping to isolate common patterns, our Layout Templates provide another important function. They provide some of a matching file's missing semantic information. Specifically, they use the location and position of matching data to impart relational model semantics. For example, based on data type, composition and position, a piece of matching text might be used as attribute labels. These mapping rules also support data alignment, i.e. determining which labels (if they exist) correspond to which data. In this respect, our templates provide a convenient abstraction for aligning matching data to the constructs of the relational model.

Each template specifies how matching data can be separated into attributes and attribute labels. For example, some of our definitions use sequence or a common token index to align labels and corresponding attribute data.

Candidate attribute labels. Each template includes rules for identifying the location of *label candidates*. Each label candidate must be a string, but each string is not necessarily a label candidate. The mapping rules accompanying each template make this noise, label or data distinction. For example, in Figure 6, the rows with similar size, {Us, 6, 1} and {Un, 6, 3}, were paired, and the row matching a uniform string pattern can be conveniently used as semantically appropriate attribute labels in defining a relation for the data matching {Un, 6, 3}.

Given this mapping between our templates and relational model constructs, we can automatically infer a schema (one or more relations) directly from this log data. For now, we assume each instance of a template matching data is independent. We show next how our approach accommodates situations when this is not true.

Boundary Identification. We induce a graph, termed Record Boundary Graph (RBG), from the instances where log text matches a Layout Template. A vertex in the RBG corresponds to a template matching some text, termed an "instance." A directed edge is induced for vertices $j \rightarrow k$ if the instance corresponding to vertex j appears in the file (from top to bottom) before the instance corresponding to vertex k. After edges are induced, we find all simple cycles (or elementary circuits) in the graph. We bound the time complexity of this operation by limiting the number of visits a vertex can be visited to two. Next, we sort the identified simple cycles in descending order according to their length. We use the longest simple cycle identified in the RBG as the principal record boundary for defining candidate relations.

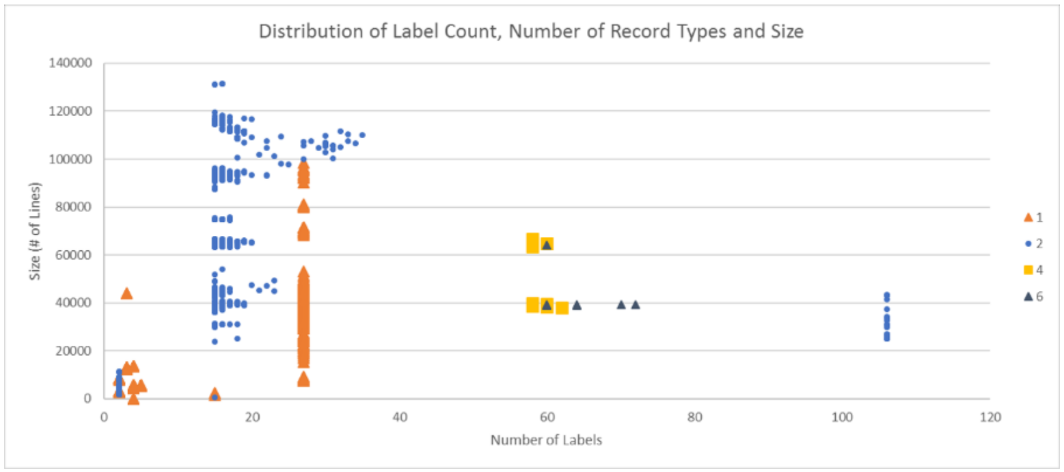


Figure 8 – Data volume and variety for three Elba performance monitors

3.5 Evaluation

We assess perftables along two broad dimensions. First, we assess its coverage of data extraction tasks of performance logs. Later, we present three illustrative scenarios to demonstrate perftables ability to provide the necessary data to diagnose millibottlenecks.

We explore our method's extractive coverage and performance along two dimensions: accuracy and processing time. We assess its performance across two different datasets. The first data set originates from our substantial experimental systems infrastructure. We use this infrastructure to conduct a broad array of systems experiments to diagnose experimental systems performance. To support this work, we need to collect an enormous amount of performance data.

3.5.1 perftables performance

3.5.1.1 <u>Elba Dataset Characteristics</u>

As Figure 4 shows, we have used *perftables* to process tens of thousands of experimental systems' performance data on the order of hundreds of GBs. Our experimental systems infrastructure primarily relies on three monitoring software programs: *collectl*, *sar* (*systat*) and *iostat*. As we briefly discussed earlier, these monitors can generate a large variety of layouts. Figure 8 shows the log diversity generated by these three monitors in our environment.

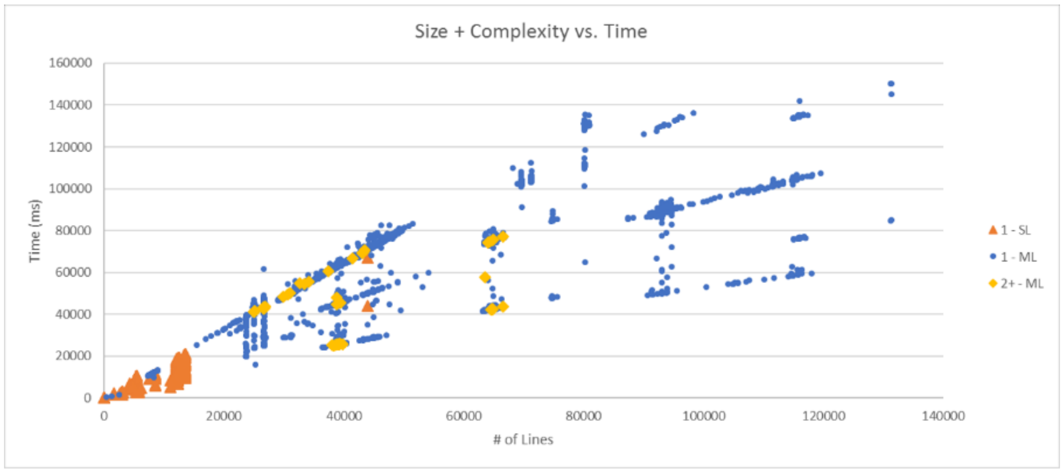


Figure 9 – Variety and Size on Performance (Elba)

Gao et al. developed a convenient categorization for describing layout variety. We adapt their categorization as follows: we differentiate interleaved and non-interleaved records precisely. In the following graphs, non-interleaved record structures are represented by "1," and in the interleaved case, we enumerate the number of record structures present in files to illustrate the variety of layouts more explicitly. We don't explicitly separate files with single lines from those with multiple lines in the interleaved case.

Figure 9 shows perftables performance by varying size and variety. The sub-linear trend highlights the effectiveness of our lazy approach. We consider variety in terms of the number of repeated record structures that appear in a file. Even files with multiple record structures, perftables performs in sub-linear time.

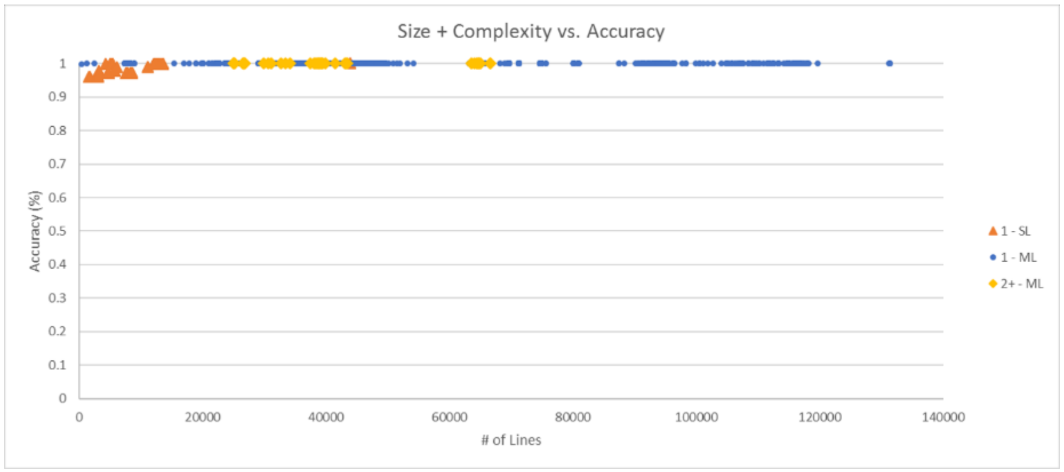


Figure 10 – Variety and Size on Accuracy (Elba)

3.5.1.2 <u>GitHub Data Characteristics</u>

The second dataset comes from the "wild" via a popular public source code repository, Github.

Github Sample. We evaluate perftables coverage of performance monitor data by collecting a randomized dataset from the wild via a popular public source code repository, Github. We retrieved this dataset by querying Github using keywords such as "log," "nagios," and "top." The latter two terms refer to two popular open-source resource monitoring tools. Given their widespread use, we thought they should be included in our sample. Table 2 details our Github sample's characteristics.

We adopted Gao et al.'s record type categorization for describing log format or layout variety [23] with one modification. For files with interleaved record structures, we do not distinguish between those with single line and multiple line records. Instead, we use the number of interleaved record structures to explicitly illustrate their variety. Our sample covers at least 7 unique monitors not currently deployed in our infrastructure, including: *top* (*profiling* and *processes*), *vmstat* (*vmware*), *oprofile*, *nagios*, *logstat* (*kvm*) and a *bespoke CPU/network monitoring tool*.

Table 2 – Sample Characteristics for Random Github Sample

Record Types	# of Samples	Avg. # of Lines
One (Single Line)	336	350
One (Multiple Lines)	434	120
Two or More	497	1559

Figure 11 show *perftables* ability of our method to extend beyond the monitors used in our experimental computer science infrastructure. On average, we were able to correctly extract over 70% of the data obtained from Gitub into relations. Instances where most of the data could not be extracted were primarily due to our approach treating network message labels as attribute labels instead of elements of an enumeration. Despite this result, repairing this error can be accomplished with some simple post-processing.

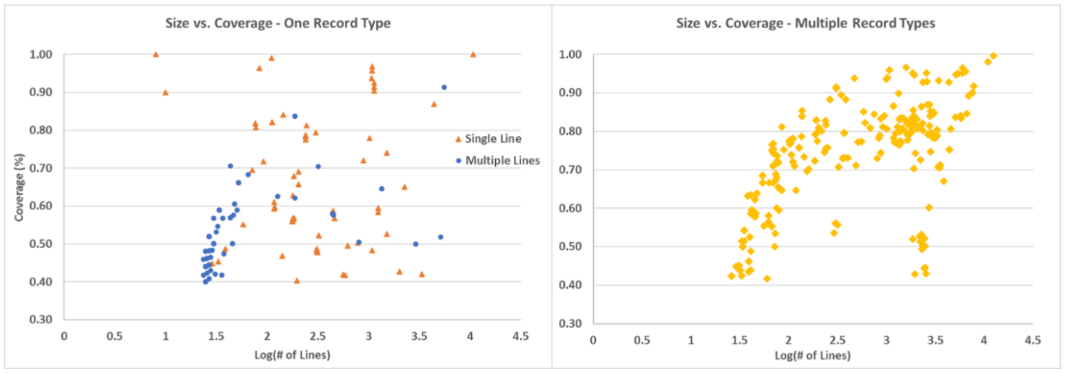


Figure 11 – Performance logs randomly sampled from Github. Each graph compares the coverage and size for files containing a Single Record Type (left) and Multiple Record Types (right).

Small Curated Set. Next, we eliminated those files from the sample that did not originate from performance monitoring programs. Then, we selected a highly regular sample from the original random data set for each of the previously listed 7 monitors. Table 3 details this small, curated subset from the original Github sample using the same categorization as Table 2.

Table 3 – Sample Characteristics for Small, Curated Subset of Github Sample

Record Types	# of Samples	# of Lines
One (Single Line)	5	3500
One (Multiple Lines)	2	3000
Two or More	1	4000

As shown in Figure 12, for this smaller, more regular subset, we were able to on average correctly extract over 90% of the data into relations. We were only able to extract approximately 75% of the *nagios* log data. Fortunately, when perftables treats some data as labels, we can employ some trivial post-processing to repair the error.

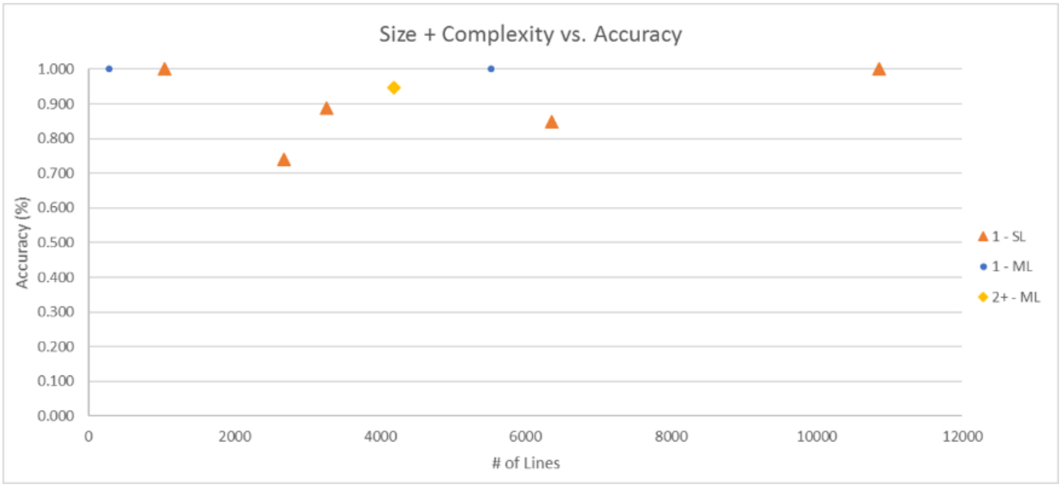


Figure 12 – Variety and Size on Accuracy for small, curated subset from Github sample

3.5.2 *perftables Illustrative Scenarios*

Next, we demonstrate perftables ability to extract the necessary data for detecting millibottlenecks in experimental data generated by our experimental infrastructure.

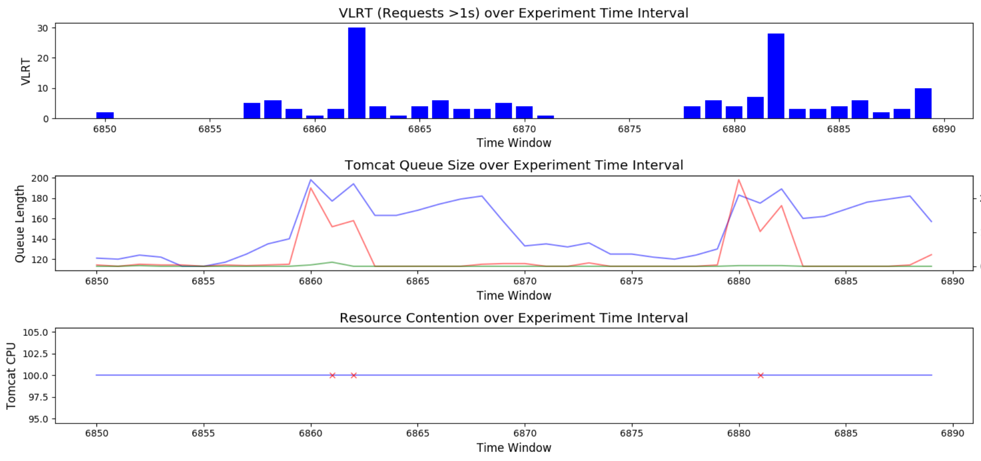


Figure 13 – Tomcat JVM Millibottleneck.

3.5.2.1 <u>Illustrative Scenario 1 – Tomcat JVM as Millibottleneck</u>

Figure 13 is a representative set of graphs necessary for effectively isolating and diagnosing millibottlenecks. In this case, these graphs correspond to a millibottleneck induced by JVM garbage collection. Each graph corresponds to a specific diagnostic step. The top graph shows the number of requests associated with very long response time, defined as requests exceeding 1s to be processed, for each 50 ms interval. The middle graph depicts the queue size of each component for each 50 ms window. We determine the size of a queue using the number of requests waiting to be processed by the given component during each interval. The bottom graph depicts the resources that are temporarily saturating over the same interval. In this case, the Tomcat node's CPU is saturated due to Java Garbage Collection. This period of saturation coincides with the appearance of VLRT and the growth in queue size among dependent components. Given the correlation among these three variables—number of VLRT, component queue size and resource utilization—we can conclude the Tomcat CPU millibottleneck is induced by the Tomcat node's JVM Garbage Collection process. We have detailed this millibottleneck and its diagnostic procedure, briefly explained here, in our prior work [24]. *perftables* automates the data extraction process to support this graphically oriented, correlation-based diagnostic method.

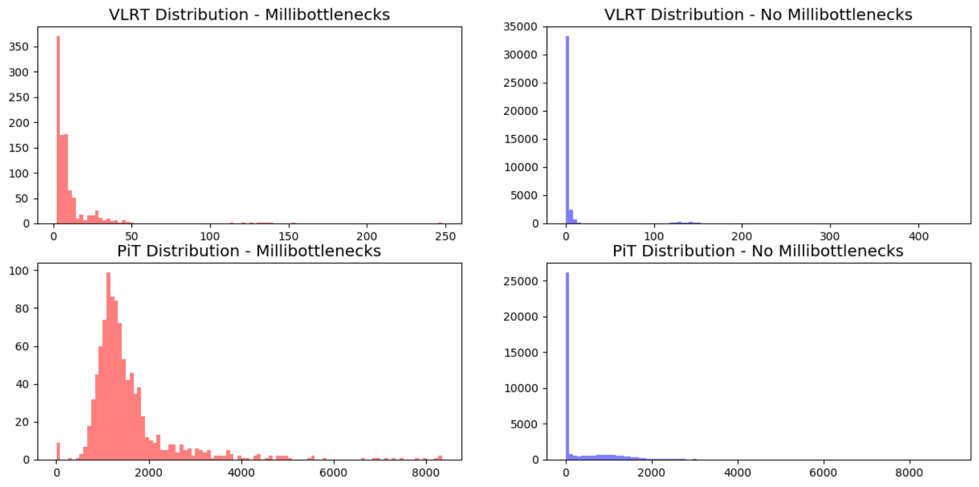


Figure 14 – VLRT and Point-in-Time Distribution comparisons for periods with Millibottlenecks and No Millibottlenecks detected.

Reconstructing Distributions. The Tomcat JVM garbage collection illustrative scenario and the ones that follow depend on two metrics that are important to diagnosing tail latency: Point-in-Time Response Time and the number of VLRTs, i.e. requests needing more than 1s to complete. To evaluate the association between these metrics and the presence of millibottlenecks more broadly, we use perftables to extract the resource data collected from thousands of system benchmark experiments to label periods with and without millibottlenecks. Then, we compare the distributions for VLRT and Point-in-Time Response Time stratified by the presence of millibottlenecks.

Figure 14 compares the data distributions for Point-in-Time response time and the number of VLRT requests split between millibottleneck and non-millibottleneck periods. These graphs suggest some interesting results. First, the number of VLRT requests provides better separability between the classes. For periods with no millibottlenecks, there is less than 0.01% weight in the tail suggesting that when there are no VLRT, millibottlenecks are not likely present. The Point-in-Time graphs suggest this metric does not separate the classes as well. Approximately 70% of the weight of the distribution occurs between 0 and 2000ms for the Millibottleneck case. In the No Millibottleneck case, over 90% of the weight of the distribution occurs between these same thresholds. This overlap in the distributions suggests that using Point-in-Time alone could lead to millibottleneck period misclassification.

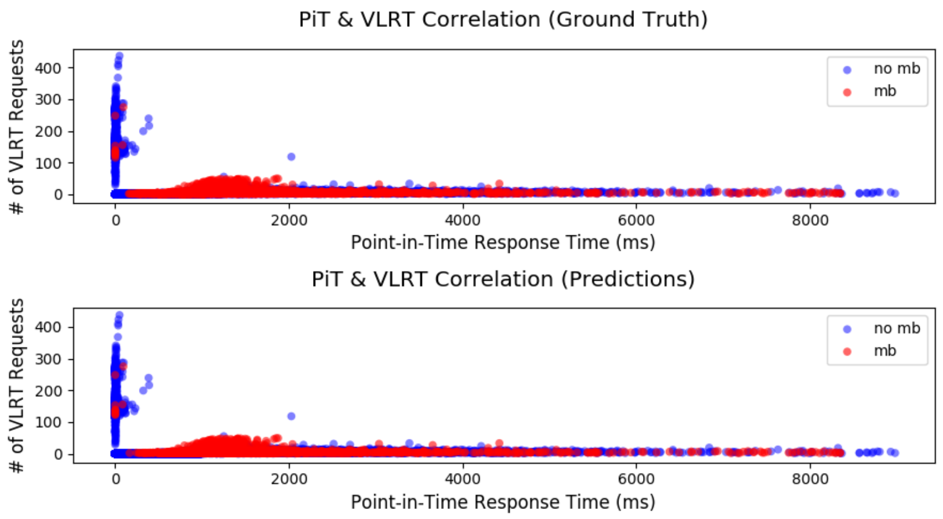


Figure 15 – VLRT and Point-in-Time comparison among predicted and ground truth Millibottleneck and No Millibottleneck periods.

Figure 15 compares the Point-in-Time Response Time and VLRT distributions for actual and predicted millibottleneck periods. The figure generally affirms the relationship among longer Point-in-Time Response Times and non-zero VLRT during periods when millibottlenecks are present. This figure highlights that using these signals together leads to accurate millibottleneck predictions.

3.5.2.2 Illustrative Scenario 2 – Database IO as Millibottleneck

In Figure 16, we illustrate perftables ability to isolate a millibottleneck due to Mysql's temporary disk saturation. Specifically, we review the period where the number of VLRT requests begins to grow and remains above 5. We can see the number of VLRT requests begins to decline quickly eventually returning to 0 in less than 250ms.

To understand what occurs during this interval, *perftables* begins by extracting request traces generated by our specialized event tracing framework, milliScope, found in component logs. This data captures execution flow dependencies. As the middle figure shows, we observe obvious Cross-Tier Queue Overflow evidenced by the components' queue lengths elongating over the period of interest. Perftables uses the extracted request trace data to calculate a few metrics every 50ms: point-in-time response time, the number of VLRT requests, i.e. those exceeding 1s, and component queue length.

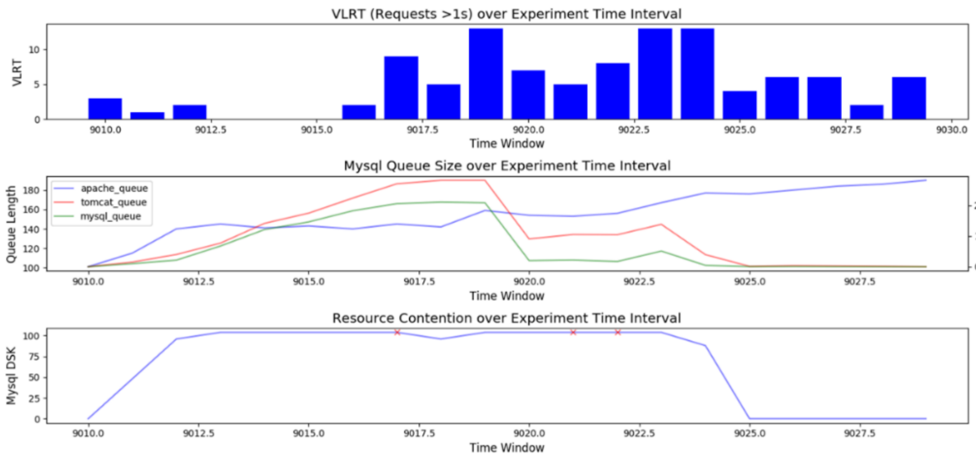


Figure 16 – Database IO as Millibottleneck. These provide visual evidence of a millibottleneck indicated by the appearance of VLRT, queue extension and temporary resource saturation.

Perftables also extracts resource data from performance logs output by resource monitors. This data provides a representation of system state. In our scenario, there was one resource monitor, *collectl¸* measuring CPU, Disk Memory and Network at 50ms intervals. We see Mysql's disk is temporarily saturated, i.e. utilization reaches 100%, but returns to 0% after approximately 300ms from the first moment it saturates. Perftables represents the extracted data for each resource category as a multivariate timeseries.

As mentioned earlier, perftables uses a data-driven approach to detect and analyze millibottlenecks. In short, it learns state and event-specific patterns consistent with the presence of millibottlenecks by relying on machine learning to systematically identify such patterns. In our case, diagnosing this database IO millibottleneck requires finding patterns where events such as the number of VLRT, the number of queued requests and the average Point-in-Time response time are maximal at the same time as state indicators such as Mysql disk resources are temporarily saturated.

We transform the event-based metrics and system state data into salient numerical features for detecting millibottlenecks. Specifically, we apply fixed-width windows to the event-based metrics mentioned earlier like point-in-time response time, the number of VLRT requests, i.e. those exceeding 1s, and component queue length to create feature column vectors. These feature column vectors are concatenated into a matrix of row vectors such that each row represents a sample. In this scenario, we would construct an event feature column vector as follows: {PIT, VLRT, Apache_Queue, Tomcat_Queue, Mysql_Queue}.

We construct system state column feature vectors from the extracted resource data in a similar fashion. Each resource measurement every 50ms is a component of a fixed width vector. In this case, each components' CPU, Disk, Memory and Network utilization are vector components. In our scenario, we would construct a state column vector as follows: {Apache_CPU, Apache_Disk, Apache_Mem_Used, Apache_Net_Bandwth, …}. Like the event vectors, these vectors are concatenated into a matrix of row vectors where each row of the matrix is a sample.

We model the problem of determining the existence of a millibottleneck over some interval of time as a multi-class classification problem. To identify millibottlenecks, we use models trained over previously labeled data. Our labels indicate whether a millibottleneck is present, and if one exists what kind it is. These models are used to predict labels for each event and state matrix sample. In this case, the model indicates the presence of a Mysql (database) IO millibottleneck indicated by the red "X's" in the bottom figure.

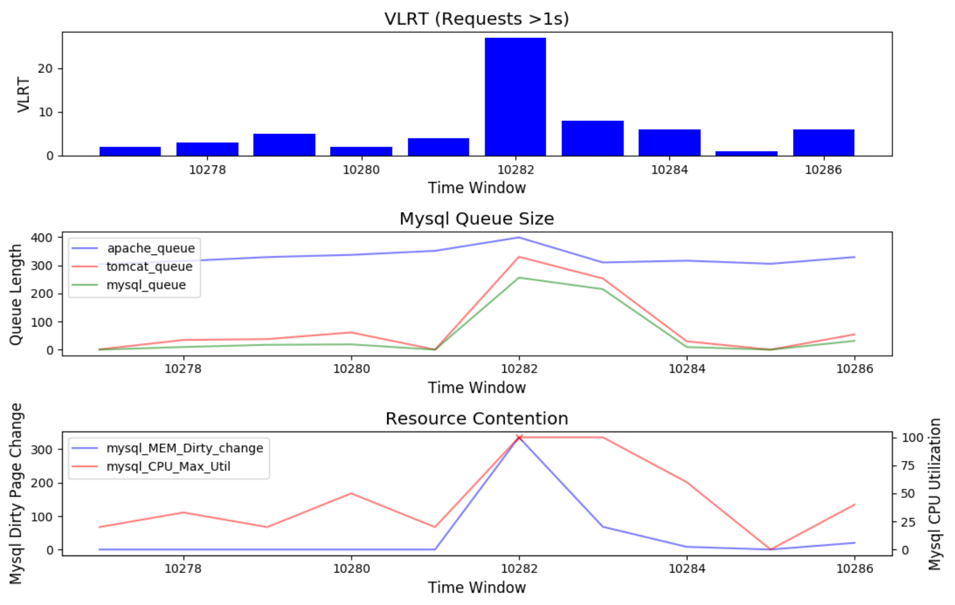


Figure 17 – Dirty Page as Millibottleneck Example. These provide visual evidence of a millibottleneck indicated by the appearance of VLRT, queue extension and temporary resource saturation.

3.5.2.3 Illustrative Scenario 3 – Memory Dirty Page as Millibottleneck

In Figure 17, we illustrate perftables ability to isolate a millibottleneck due to memory dirty page being flushed to disk. Specifically, we review the period where the number of VLRT requests begins to grow. We can see the number of VLRT requests begins to decline quickly eventually returning to 0 in less than 250ms.

As in the prior situation, *perftables* begins by extracting request traces and calculates the associated event metrics. As the middle figure shows, we observe obvious Cross-Tier Queue Overflow evidenced by the components' queue lengths elongating over the period of interest. As before, perftables also extracts all resource data from the pertinent performance logs, in this case, *collectl*. We see Mysql's CPU temporarily saturates during the period of interest.

This situation highlights the need to create features to represent magnitudes like counters, percentages or rations and derivatives like velocity and acceleration. In the prior situation, we created features directly from data extracted from performance logs. In this case, Mysql flushing dirty pages to disk is a phase change. Diagnosing this type of millibottleneck requires features to account for a magnitude such as CPU utilization and a velocity measure like the change in dirty pages. As such, derivative measures are also components of the event and state feature matrices. In this illustration, the sudden change in the number of dirty pages is the primary signal. During this period of interest, we see this change corresponds to the other conditions present during this period: Mysql CPU suddenly and temporarily saturating, queue lengths elongating and the number of VLRT requests increasing.

This scenario also highlights the need to look for performance patterns across multiple resources across multiple components simultaneously. To account for this multiplicity, we employ a team-based classification approach to learning. Specifically, we train millibottleneck-specific models meaning we train over data containing negative and positive examples where the positive examples are of the same type. During prediction, we feed the feature vectors corresponding to a period of interest into each model. The model with the highest confidence is the final prediction for the given input. In this case, the model trained on data consisting of memory dirty page examples returns with the highest probability indicated by the red “X’s” in the bottom figure.

3.6 Conclusion

In this paper, we introduced our approach, perftables, for automatically inducing relations from the log data generated by commonly used performance monitoring tools. The reasons for extracting this data into a relational form are many: facilitates integrating event and resource data across space (which node / component) and time (when did the event happen or when was the measurement made), supports automated analysis techniques like machine learning and ultimately enables researchers to glean patterns across a vast volume of experiments occurring over many years.

We demonstrated that we can successfully extract over 98% of our experimental infrastructure's performance data into relations despite the presence of variable length records and multiple record types. Finally, we have shown that our approach extends beyond the array of performance monitors present in our infrastructure by collecting a sample of other performance monitoring logs from the wild. We also presented three case studies showcasing perftables provides the data necessary for isolating millibottlenecks.

CHAPTER 4. PERFDB

4.1 Introduction

The long tail latency problem is a well-known problem, particular in web-facing applications. For example, Amazon has reported [2] that a marginal increase of 100ms in page load time correlates with approximately a 1% reduction in sales. Google has reported a similar revenue-latency relationship. It has found that 500ms of additional delay returning search results could reduce revenue by up to 20% [3]. Given the size of their respective customer bases, these platform companies need to reduce their 95th, 99th and 99.9th percentile latencies as close to zero as possible [1] [4].

Dean et al identified long tail latency as a problem common to large scale system topologies, and they prescribed several approaches for bypassing it, specifically fault tolerance and resource redundancy techniques [1] [25]. While these strategies are effective at reducing the impact of long tail latency problems, they come with significant cost. One implication of their recommendations is lower overall datacenter utilization. According to an NRDC report, from 2008 – 2012, average data center server utilization ranged from 12-18% [26].

Our theory—*the millibottleneck theory of performance bugs* [5]—attempts to explain performance anomalies that cause requests, which normally return in tens of milliseconds, to complete on the order of hundreds or thousands of milliseconds. We call these Very Long Response Time (VLRT) Requests, and they comprise part of systems' response time long tail. We have isolated root causes of millibottlenecks crossing the architectural stack, including CPU dynamic voltage frequency scaling (DVFS) control [6], Java garbage collection (GC), “noisy neighbors” due to virtual machine consolidation [7], dirty page flushing [8], memory thrashing due to rapid succession of page faults [9] and writing log data to disks [10] [11].

Approaching latency issues by addressing root cause has three important benefits. First, isolating and eliminating performance bugs (in the first place) reduces the system redundancy costs and improves the overall datacenter return on investment.

Secondly, we believe systematically studying performance bugs can help us to potentially develop new theories or models about system design and implementation. For example, practitioners often treat each performance problem they encounter as a specific, one-off instance. According to a recent report, NVIDIA estimates automating away only 5 minutes of the performance debugging process could yield $5M in engineering time savings [27]. Developing a more robust understanding of performance anomalies and their root causes could improve performance diagnosis and debugging efficiency.

Thirdly, long tail latency contributes to performance unpredictability, which according to the most recent Berkeley View on Serverless Computing is an enormous challenge to cloud-native architectures like serverless and microservices [28]. Specifically, they report performance unpredictability as one of the three most cited reasons practitioners do not migrate application workloads to serverless computing topologies. Understanding latency and its root cause is critical to increasing the adoption of newer architectures like serverless.

Our previous millibottlenecks studies' unit of analysis have been the individual period or experiment level. Studying transient phenomena even at this level is challenging due to their fleeting nature and the intricate system component interactions necessary for reproducing them. Studying transient phenomena as a population or group introduce an additional set of challenges, including controlling for variation across experiments such as resource capacity, data management concerns like data quality, and developing new visualizations to support hypotheses and conclusions.

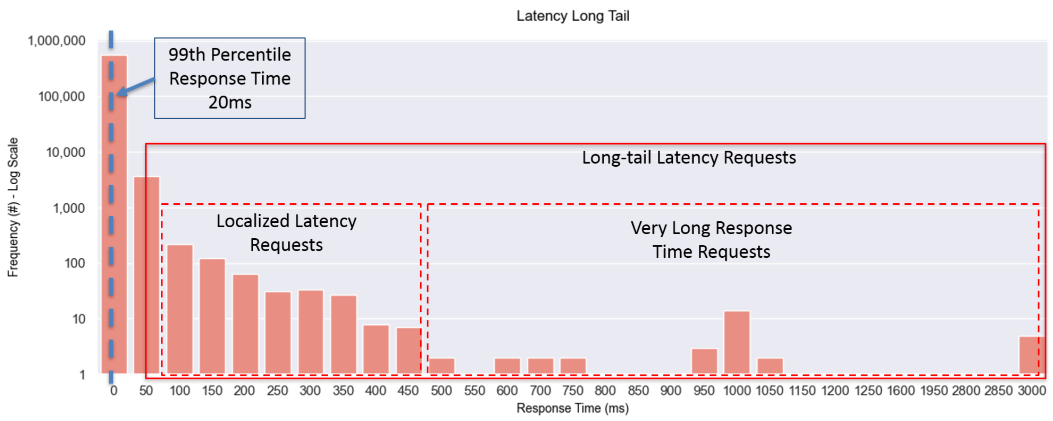
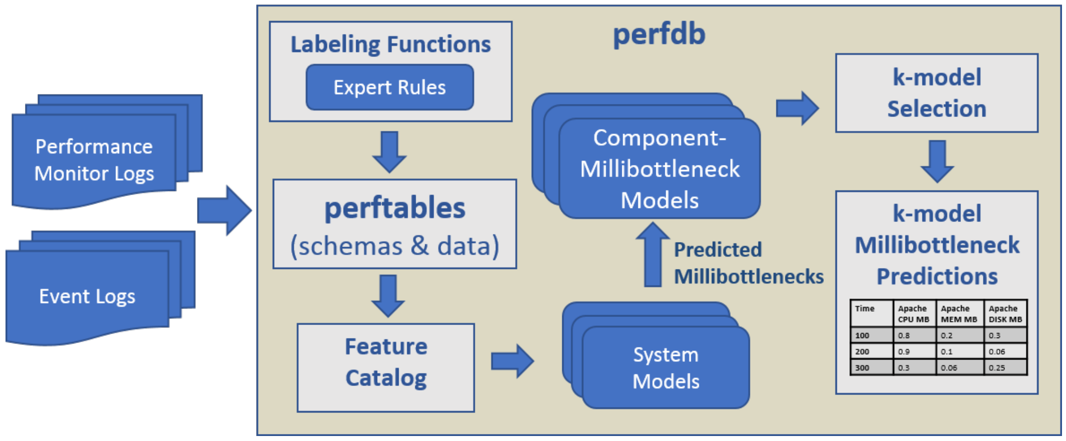


Figure 18 – Latency long tail of experimental periods in our data catalogue. We see the appearance of VLRT and LLR with 99% periods completing 20 ms or less.

This paper describes two contributions: the first one is phenomenological and the second one methodological. The main phenomenological contribution is a large-scale aggregate analysis of a diverse set of millibottlenecks found from ~500 experiments. By studying the similarities and differences among the approximately 30K millibottlenecks, we confirmed the previously known association between transient VLRT requests and millibottlenecks. Furthermore, our data support the explanation of CTQO (Cross-Tier Queue Overflow) [18] as cause of transient VLRT requests in our experiments. More interestingly, we identified the class of Localized Latency Requests, (LLRs) that have duration between VLRT request and the average latency. LLRs are associated typically with millibottlenecks of shorter duration than those causing CTQO.



Time	Apache CPU MB	Apache MEM MB	Apache DISK MB
100	0.8	0.2	0.3
200	0.9	0.1	0.06
300	0.3	0.06	0.25

Figure 19 – PerfDB system data flow. Event and Performance Monitoring Logs are ingested and interpreted by perftables, which automatically induces relations. This relational data is transformed using features from the catalogue that are necessary for detecting millibottlenecks. System and component-millibottleneck models are used to (i) isolate millibottleneck periods (ii) diagnose the type of millibottleneck respectively.

The main methodological contribution of the paper consists of the PerfDB data management system that produced the aggregated data sets, enabling the first identification of LLRs and confirmation of the association of VLRT with millibottlenecks causing CTQO. Using a combination of rule-based and machine learning techniques, PerfDB is able to parse, extract, transform, and load a collection of fine-grain (sampling at 50ms intervals) resource monitoring data (Figure 20) on a wide variety of hardware and software configurations (Table 4). In addition, PerfDB also interprets and loads fine-grain event monitoring data (timestamped arrival and departure of every message at every server). From the ~500 experiments, PerfDB was able to clean and integrate approximately 2TB of performance monitoring data into a data warehouse for detailed statistical analysis that enabled the first identification of LLRs and confirmation of strong association of VLRTs with millibottlenecks and CTQO.

4.2 Problem Definition and Related Work

4.2.1 Long Tail Latency Requests

The response time long tail consists of requests that take more time to return than the 99th (or 95th) percentile of system latency. Some of these requests take this long due to request-specific characteristics, for example, a search query composed of rare terms. VLRT requests, which arise due to queueing effects, are the other set of requests comprising the latency long tail that complete in hundreds or thousands of milliseconds but complete in tens of milliseconds when they are executed separately.

Our group's previous work focused on the study of individual millibottlenecks, extended to the millibottlenecks of the same class, (the same contended resource), that caused VLRT requests [5] [6] [7] [29] [8] [9] [10] [11] [30]. In this work, we conduct a series of population studies, and explore a phenomenon called Relatively Long Tail Requests (LLR). Unlike VLRT, LLRs are not associated with queue growth, but as depicted by the red dashed region in Figure 18 are still apart of the latency long tail. We define LLR more formally later.

Long tail latency. To estimate system latency for each sampling period, we use an aggregate measure called *Point-in-Time Response Time (PIT).* This measure of system latency is defined by the wall clock time requests take to complete a system round trip. To calculate this metric, we average the round-trip time of all requests beginning within a specified interval. In our case, we use fixed width 50ms intervals.

4.2.2 Millibottlenecks

Millibottlenecks are short resource saturations, i.e. order of tens of milliseconds. Millibottlenecks arise due to competition for a shared system resource irrespective of the type of resource. Moreover, their effects are propagated and amplified through system dependencies.

4.3 PerfDB Approach

4.3.1 Overview

PerfDB pictured in Figure 19 has three parts: millibottleneck schema, *perftables* and a data management platform we call *perfstore*. We begin by describing the schema necessary for experimentally verifying millibottlenecks and their relationship to VLRTs and LLRs.

4.3.2 Millibottleneck Data Model

We define a Millibottleneck Instance as a tuple: {Start Time, Duration, Contended Resource, and Location} where each attribute is defined as follows:

- *Contended Resource* is a temporarily, highly utilized resource, e.g. at least 60% utilization.
- *Location* marks the component with a Contented Resource.

- Saturation period is the interval bounding the period of high resource utilization for a specific component, i.e. the period framing the Contended Resource for a specific Location. *Start Time* marks the beginning of a saturation period, and *Duration* denotes its length.

Given this schema, (collection of attributes), we need to track precisely when resources become contended, i.e. highly utilized, and saturated. As such, we use a collection of standard, open source linux performance monitors such as top, perf, sar and collectl to capture resource metrics at fine-grained timescales, i.e. in our case, 50ms. Monitors like these cover the collection of performance metrics such as CPU Utilization, Disk Utilization, Network Throughput, Cache Usage and Memory counters such as Dirty Page and Page Faults. Metrics such as these constitute the attributes comprising our resource measurement schema.

The second part of the *millibottleneck theory* concerns diagnosing millibottlenecks that induce long-tail latency requests. This means we need to know the location of requests at fine-grained timescales, so we can correlate requests' latencies to the temporary (or, fleeting) resource saturations. Our event schema provides the necessary data for doing so, and it is as follows: {*RequestID, Start Timestamp Upstream, Start Time Downstream, End Timestamp Downstream, End Timestamp Upstream*}. PerfDB ingests this data, so it can calculate system-level metrics such as Point-in-Time Response Time and component-level measurements like queue length to analyze potential queueing effects [11].

Labels. Some types of millibottlenecks, for example those caused by Java Garbage Collection, can be labeled with deterministic rules assuming one knows beforehand that they exist in an experiment. We implemented a set of labeling functions to bootstrap our learning models, and we applied them to an initial set of experiments where we knew *a priori* millibottlenecks that can be described by these functions existed. Not all millibottlenecks can be labeled in this manner. Some occur due to statistical circumstances in the sense that very short resource contentions materialize due to the "right" sequence of system events occurring, i.e. nondeterministic execution. This noisy process of labeling millibottlenecks in part has motivated us to begin exploring machine learning techniques for this purpose, an area of future work.

Features. PerfDB also includes a pre-defined set of features that are important to isolating and detecting millibottlenecks. Features are data transformation functions, which are applied to schema attributes. These data transformations include encoding and standardizing data as well as aggregate functions, such as min, max and mean. We bootstrapped PerfDB's labeling using a combination of hand labeled data and labeling functions that were applied to feature-transformed data.

4.3.3 perftables

We use *perftables* relation induction approach to automatically transform the performance log data output by our infrastructure's event and resource monitors into relational structures [31]. PerfDB ingests these relations for each experiment. They provide a convenient data abstraction for mapping to PerfDB's data management platform.

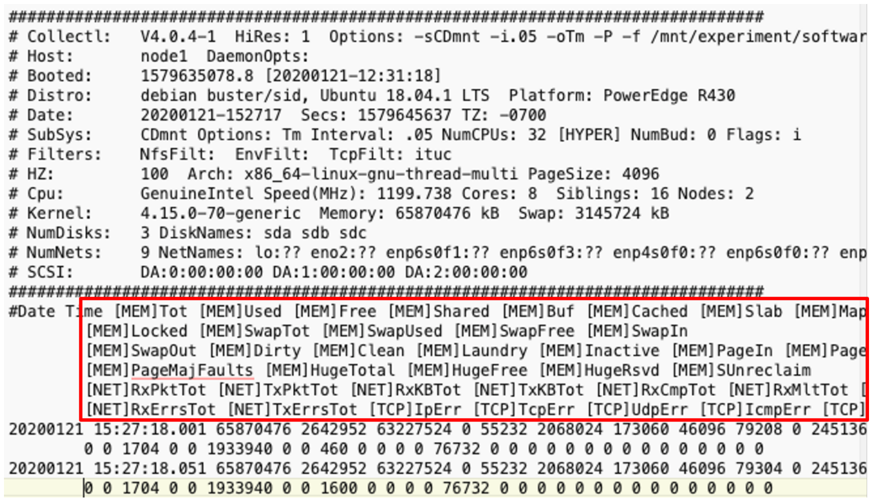
Table 4 – PerfDB Experiment Catalog

Measure	Value
# of Experiment Periods	1.2M+
# of Millibottleneck Periods	30,000+
Performance Data Size (GB)	2000
Benchmark	Rubbos
Benchmark Workload Range	100-20,000
Benchmark Runtime (Avg.)	3 mins
Hardware	p3000, d430, d710
Topologies	111, 1111, 1112, 121, 441
Components	Apache, Tomcat, CJDBC, Mysql

4.3.4 perfstore

PerfDB encompasses a large experimental systems space, as indicated by Table 4. Our platform contains experiments executed on *CloubLab* using three primary machine types. The p3000 machines are single core machines with 2GB of memory while the d710 and d430 machines contain 8-cores and 32-cores respectively with over 64 GB of memory in both. To date, we have relied on the Rubbos monolithic, n-tier application benchmark. The benchmark is a Reddit-style bulletin board system. We deployed this benchmark using Apache web server, Tomcat JEE application server, CJDBC middleware and Mysql database in various topology configurations. The numbers related to topology correspond to the number of deployed Apache, Tomcat, CJDBC and Mysql components, in that order. All experiments in our collection lasted for 3 minutes, but the workload intensity varied from 100 to 20,000 concurrent client sessions or connections.

Spanning this experimental configuration space, PerfDB's schema supports over 500 unique system experiments comprising 2TB of performance data and over 600 different performance attributes across these systems. Within this data set, we have identified over 30,000 millibottleneck periods—occupying about 2% of all experimental time periods.



```
################################################################################
# Collectl:   V4.0.4-1  HiRes: 1  Options: -sCDmnt -i.05 -oTm -P -f /mnt/experiment/softwar
# Host:       node1  DaemonOpts:
# Booted:     1579635078.8 [20200121-12:31:18]
# Distro:     debian buster/sid, Ubuntu 18.04.1 LTS  Platform: PowerEdge R430
# Date:       20200121-152717  Secs: 1579645637 TZ: -0700
# SubSys:     CDmnt Options: Tm Interval: .05 NumCPUs: 32 [HYPER] NumBud: 0 Flags: i
# Filters:    NfsFilt:  EnvFilt:  TcpFilt: ituc
# HZ:         100  Arch: x86_64-linux-gnu-thread-multi PageSize: 4096
# Cpu:        GenuineIntel Speed(MHz): 1199.738 Cores: 8  Siblings: 16 Nodes: 2
# Kernel:     4.15.0-70-generic  Memory: 65870476 kB  Swap: 3145724 kB
# NumDisks:   3 DiskNames: sda sdb sdc
# NumNets:    9 NetNames: lo:?? eno2:?? enp6s0f1:?? enp6s0f3:?? enp4s0f0:?? enp6s0f0:?? enp
# SCSI:       DA:0:00:00:00 DA:1:00:00:00 DA:2:00:00:00
################################################################################
#Date Time [MEM]Tot [MEM]Used [MEM]Free [MEM]Shared [MEM]Buf [MEM]Cached [MEM]Slab [MEM]Map
        [MEM]Locked [MEM]SwapTot [MEM]SwapUsed [MEM]SwapFree [MEM]SwapIn
        [MEM]SwapOut [MEM]Dirty [MEM]Clean [MEM]Laundry [MEM]Inactive [MEM]PageIn [MEM]Page
        [MEM]PageMajFaults [MEM]HugeTotal [MEM]HugeFree [MEM]HugeRsvd [MEM]SUnreclaim
        [NET]RxPktTot [NET]TxPktTot [NET]RxKBTot [NET]TxKBTot [NET]RxCmpTot [NET]RxMltTot [
        [NET]RxErrsTot [NET]TxErrsTot [TCP]IpErr [TCP]TcpErr [TCP]UdpErr [TCP]IcmpErr [TCP]
20200121 15:27:18.001 65870476 2642952 63227524 0 55232 2068024 173060 46096 79208 0 245136
        0 0 1704 0 0 1933940 0 0 460 0 0 0 0 76732 0 0 0 0 0 0 0 0 0 0 0 0 0 0
20200121 15:27:18.051 65870476 2642952 63227524 0 55232 2068024 173060 46096 79304 0 245136
        0 0 1704 0 0 1933940 0 0 1600 0 0 0 0 76732 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 20 – collect 4.0.4 performance log Memory and Network attributes

We materialize four relations to realize the complex schemas outlined in the first sub-section: one containing *latency* information, another containing components' *queue lengths*, another containing *resource metrics* and the last persisting *Millibottleneck instances*.

Start Time, Duration, Location and Resource are attributes used to define a Millibottleneck. We observe that each of the first three relations either aligns explicitly to one of these attributes or provides the necessary information for relating millibottlenecks to long-tail latency requests. For example, the latency and queue size tables are necessary for identifying periods where VLRT requests and millibottlenecks overlap and assessing whether queueing effects coincide with these periods.

Each of the first three relations have the same *primary key*: the *observation timestamp* and a *globally unique identifier* (guid) to represent the experimental execution. Each row in each of these tables corresponds to 50ms intervals—the finest granularity our resource and event monitors provide. To ensure millibottlenecks are unique instances, we add duration to the Millibottleneck relation's primary key. Consequently, each row represents one instance of a millibottleneck.

Our *latency* relation in addition to Point-in-Time Response Time also contains counts for the number of requests that complete in: 5ms, 10ms, 25ms, 50ms, 100ms, 250ms, 500ms, 1000ms, 3000ms, 5000ms. These thresholds are right-side bounded intervals, i.e. requests less than or equal to the threshold are counted in the corresponding "bin." These counts are important to understanding the distribution of system latency.

Our *queue length* relation contains attributes for each system component apart of our collection. In this case, we have an attribute representing the size of the Apache, Tomcat, CJDBC and Mysql queues.

Our *resources* relation contains many resource metric-related attributes. Figure 20 shows a log snippet, which highlights the large number of resource attributes just for Memory and Networking as indicated by the red box. Besides being large in number, resource attributes are not necessarily consistent across experiments.

To illustrate this point, we present Table 5, which captures the number of resource attributes for each machine type holding constant the monitor version, e.g. *collect 4.0.4* shown in Figure 20, and monitoring parameters. Despite holding the resource monitor and parameterization constant, the number of attributes vary due to architectural and kernel disparities across the machine types. To compound matters, we see ranges for some machine types, especially for the d430 machine type, which appears in the last column of the table. For example, we see 31 – 97 CPU attributes reported for this type of machine. This range can be due to differences in how hardware and kernels report unused or underused resources—suggesting the number of attributes can change *per execution*, i.e. an experiment-specific resource schema. In fact, we found 347 different resource schemas represented in PerfDB across our single largest experimental subset consisting of approximately 380 experiments. Consequently, the resource attributes are not consistent across the *perfstore* experimental data table space.

Despite these trans-experiment schema complexities, our millibottleneck data model provides a convenient data abstraction for isolating, diagnosing, and studying millibottlenecks and their association with the latency long-tail as we detail in the following section.

Table 5 – Experimental Schema Characteristics

	collectl 4.0.4, Fixed monitoring parameters		
PerfDB Attributes	***p3000 (2 core)***	***d710 (8 core)***	***d430 (32 core)***
CPU Attributes	4	27-62	31-97
Memory Attributes	--	8	7-22
Disk Attributes	2	6-8	6-32
Network Attributes	--	4	4-12

4.3.5 Millibottleneck Analytics

To isolate millibottlenecks and associated effects like VLRT requests and LLRs, we need to integrate the latency, queue length and resource relations implemented in *perfstore*. (We refer to these collectively as the "measurement" relations). Integration begins the tables associated with an individual experiment or execution. We join the "measurement relations" using each relation's timestamp key.

We employ a similar strategy to integrate data across experiments. We define projections containing all attributes related to a specific component or topology found in PerfDB. In our case, we have projections for Apache, Tomcat, CJDBC, Mysql components in addition to three- and four-tier topologies. Specifically, we build *component* and *topology-specific* materialized views, i.e. relations assembled in memory.

Example. A CPU millibottleneck in one experiment might correspond to the 1st CPU core being saturated while another millibottleneck in the same experiment might refer to the 8th CPU core, i.e. attributes like CPU_1_Utilization and CPU_8_Utilization. The contended resource for each of these millibottlenecks would be the name of each of these attributes. If left unchanged, the domain of the Contended Resource attribute (defined in 4.3.2) would be the *union* of all resource metric attributes across the *perfstore* table space—an enormous space given Table 5. As such, we need a mechanism for relating similar contended resources across individual millibottlenecks. In our example, the CPU utilizations for the 1st and 8th cores respectively should both refer to the same contended resource, e.g. *CPU Utilization*.

Resource Variables. We use a data abstraction, Resource Variables, to help us relate similar or a category of attributes. A Resource Variable is defined with a label and regular expression. Using our CPU Utilizaton example, we define a Resource Variable for it with the following label and regular expression:

CPU Utilization: ***.*CPU.*Util.****

Resource Variables regular expressions are matched against all resource attributes' labels (across *perfstore)*. The matching attributes are then used to assemble the component-specific materialized views. In effect, this abstraction provides a dynamic *UNION* of resource attributes. This abstraction and approach help us to manage the schema complexity described in Table 5.

Millibottleneck Identification. Features are special attributes defined using Resource Variables and a data transformation function(s) like aggregations or standardization. They transform data to help isolate the signals important to isolating millibottlenecks. Features are implemented in *perfstore* materialized views.

Our CPU Feature specifies the **CPU Utilization** Resource Variable and a *maximum* aggregation function. Returning to our example, the CPU Resource Variable would return the core-specific attributes, e.g. CPU_1_Utilization, CPU_2_Utilization, CPU_3_Utilization, …, CPU_8_Utilization. This feature's maximum function applied to a row across these core-specific attributes would results in a maximal CPU utilization value for the row. Applying the feature to an entire relation (across all rows) yields a CPU Utilization attribute.

To determine millibottleneck periods, we employ a simple heuristic:

- Input: Topology or component-specific materialized view
- In Step 1, we select the rows from the view where millibottleneck-specific features show saturation like 100% utilization or some other suitable threshold, i.e. 95th or

99th percentile value. For example, in our CPU example, we would select rows where the CPU feature produces values greater than 90%.

- In step 2, if two consecutive data points are saturated, we group them together into a "saturation period". We repeat step 2 and continue the grouping of saturated data points until all the saturation periods have a beginning (a transition from non-saturated point to a saturated point) and ending (a transition from saturated to non-saturated).

We observe that at the end of Step 2, we have the requisite data needed to materialize a Millibottleneck Instance as defined in 4.3.2. Specifically, the materialized view used as input to the heuristic determines the Location. The saturation period provides the Start Timestamp and Duration information. The Resource Variable's label available through the feature used in Step 1 provides the Contended Resource.

Joining Millibottlenecks to Measurements. We observe millibottlenecks span multiple rows as described by Step 1 in our heuristic, i.e. they span multiple time periods or observations. To join measurements to millibottleneck instances, we can apply a left outer join with perfstore's millibottleneck relation (assuming the measurement relations are on the left-hand side). We have implemented millibottleneck-specific materialized views using this approach to provide convenient access to the experimental data used to identify millibottlenecks in the first place.

Table 6 – Millibottlenecks and VLRT Requests

	Presence of VLRT Requests	
Millibottleneck Type	***VLRT Present***	***VLRT Not Present***

Apache CPU	23	5600
Tomcat CPU	35	8015
CJDBC CPU	17	6892
MySQL CPU	129	9664

4.4 Experimental Verification of VLRT and LLR

The data schema described in Section 4.3 enabled an aggregated study of ~30K millibottlenecks across ~500 experiments. Instead of analyzing each category of millibottleneck separately, for the first time we are able to compare many millibottlenecks both within a category and across categories, in terms of their similarities and differences. In subsection 4.4.1, we describe the frequency of each millibottleneck category occurring in the aggregate data set. In subsection 4.4.2, we describe the experimental confirmation that VLRT requests in this data set are associated with millibottlenecks that cause CTQO. This finding provides support for the observation that CTQO was an explanation for VLRT requests. In subsection 4.4.3, we introduce a quantitative characterization and experimental verification of Localized Latent Requests (LLRs), which have durations between system average (low milliseconds) and VLRTs.

4.4.1 Frequency of Millibottleneck Occurrence

Our first step is a population study of all millibottlenecks in our experimental collection. We start from a subset of particularly interesting (and less prevalent) millibottlenecks that coincide with VLRT requests. Given Table 6, we see millibottlenecks overlapping periods with VLRT requests represent less than 1% of all millibottleneck instances, which is consistent with VLRTs being a latency long-tail phenomenon.

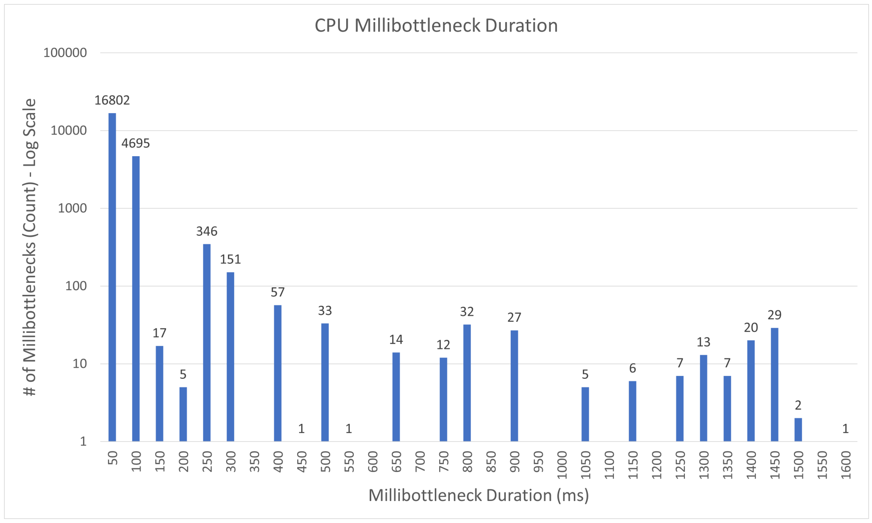


Figure 21 – Distribution of CPU Millibottleneck Duration over machine capacities and workload across experiment catalog

Figure 23 shows that the experimental data set contains a rich collection of these kinds millibottlenecks. They include all node resources being monitored (CPU, memory, disk), with CPU millibottlenecks appearing in every server (Apache, Tomcat, CJDBC, and MySQL). The diversity and coverage of millibottlenecks suggest that the experiments are well designed and tuned for the configurations that have been tested (see Table 4 with representative performance and millibottlenecks found.

In a second step, we subdivide the CPU millibottlenecks (which comprise more than 80% of the total) according to their duration in Figure 21. The log scale shows a relatively complex long-tail distribution, with more than 95% of the millibottlenecks lasting less than 100ms (only one observation point). The complex distribution of millibottleneck duration (aggregated over all servers) suggests that CPU millibottlenecks may have a variety of causes with statistical queuing effects.

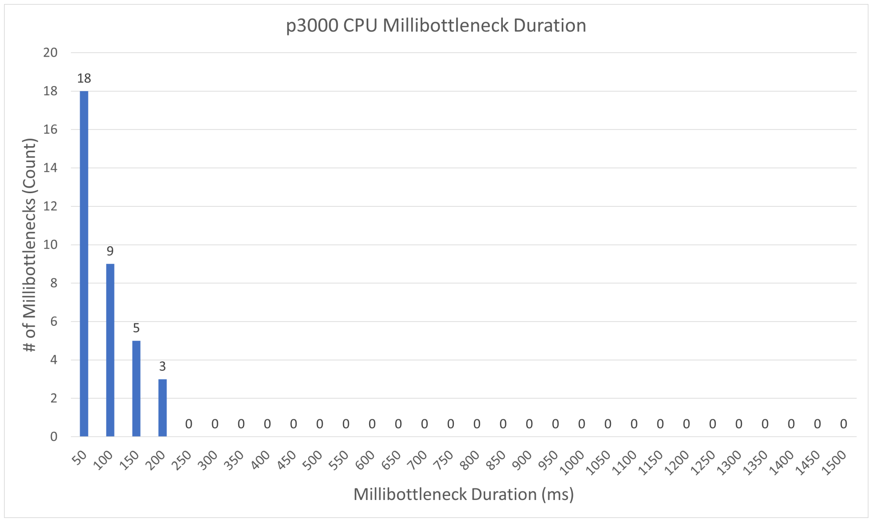


Figure 22 – Distribution of CPU Millibottleneck Duration for low capacity machines, e.g. p3000

As an example of analytical refinement to understand the complex aggregated distribution, we separated out the CPU millibottlenecks from experiments on p3000 machines, a specific (older) type of servers. The subplot of p3000 CPU bottlenecks is shown in Figure 22, with short millibottlenecks (no longer than 200ms) only. This is a surprising result, since the longer millibottlenecks (right side of Figure 21 occurred on faster, more capable servers, contrary to our expectations. Early appearance of millibottlenecks (second row of Table 6) also suggests potentially interesting causes. Further analyses of millibottleneck characteristics according to hardware and software configurations is a topic of ongoing research.

Table 7 – Minimum Millibottleneck Workloads

Presence of VLRT Requests		
Machine Type	***Min. Workload***	***Millibottlenecks @ Workload***
2 GB, 1-core	1,000	1
64 GB, 8-core	100	9
16 GB, 4-core	15,000	47

4.4.2 Millibottleneck with CTQO as Cause of VLRT

In our previous work [5] [6] [7] [29] [8] [9] [10] [11] [30], we have shown that a variety of millibottlenecks can cause VLRT requests in the presence of Cross-Tier Queue Overflow (CTQO [18]), which happens when a millibottleneck appears in an upstream server (e.g., Tomcat), stopping services temporarily and causing a downstream server (Apache in this case) to suffer queue overflow, dropped packets, and VLRT requests.

In Figure 23, we show occurrences of millibottlenecks that coincide with the VLRT requests, called "VLRT Millibottlenecks." Each bar represents the frequency of occurrence for each type of millibottleneck marked on the X-axis. While CPU millibottlenecks are more prevalent, all previously reported types of millibottlenecks are represented in our catalogue.

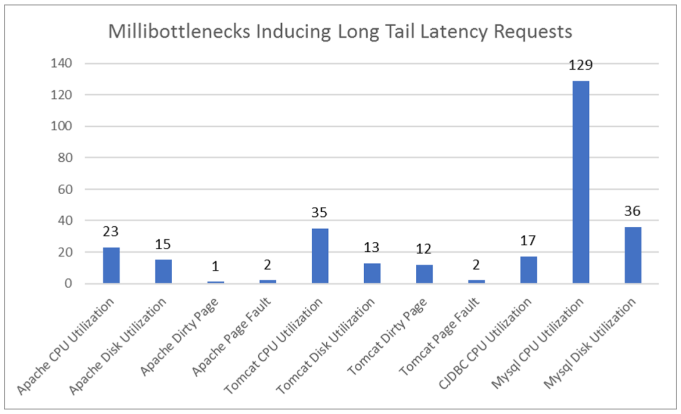


Figure 23 – Millibottlenecks found in our experiment collection stratified by type that coincide with periods where long-tail latency requests are observed.

In the third step of our analysis, we verify the presence of CTQO in each instance of VLRT Millibottlenecks (Figure 23). The presence of CTQO is confirmed by correlated queues in connected servers.

In the fourth step of our analysis, we look at the LLR requests shown in Figure 24, with duration from 100ms to about 500ms. The LLRs have not been studied previously, since they are shorter than the threshold considered to be VLRT requests. Due to their duration being shorter than TCP timeout period (currently default setting of 1sec), LLRs are unrelated to CTQO.

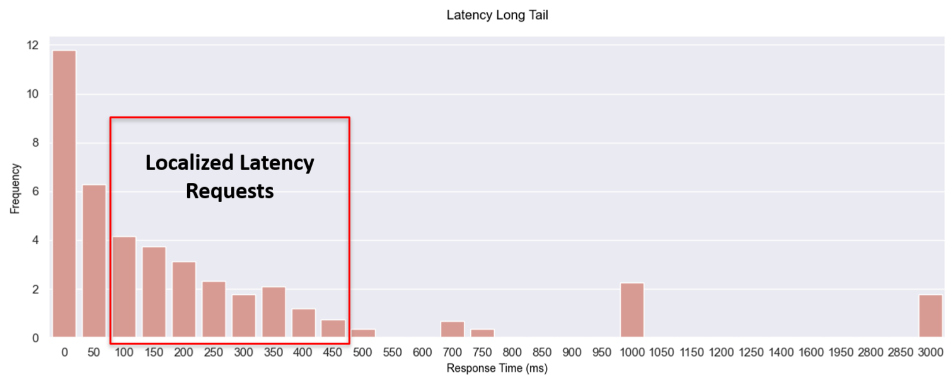


Figure 24 – Latency long tail of CPU Millibottlenecks. We see the appearance of LLRs. Most of the periods coincide with system response times < 50 ms.

LLR Requests. Figure 24 shows a representative example of Mysql CPU millibottleneck in which LLR requests appear. The LLRs are considerably shorter than VLRT, but they also appear more often.

Next, we explore the overlap between the Mysql CPU millibottleneck instances and queue growth. Figure 25 shows the queue growth and Mysql CPU Utilization levels corresponding to a subset of Mysql CPU millibottlenecks. Each instance appears along the X-axis ordered by the total queue length (of Apache and Tomcat respectively) at the time of the millibottleneck. All instances have high CPU utilization, a key component of the millibottleneck theory. The queue growth for columns 6 – 14 are consistent with the CTQO explanation. These instances are highlighted with the green circle. Figure 25 also shows several instances, which do not coincide with any queue growth, as highlighted by the red circle and causing only LLRs.

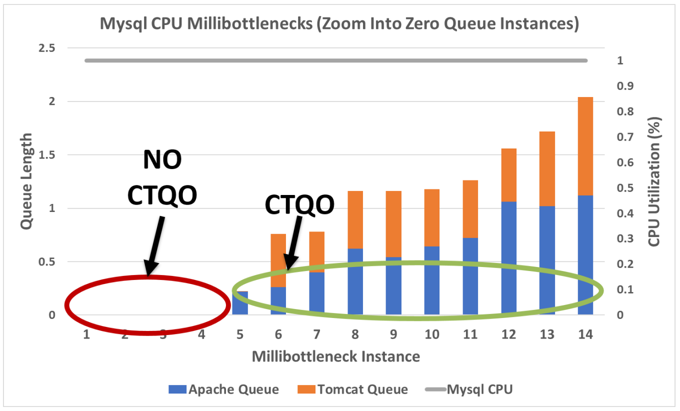


Figure 25 – MySQL CPU Millibottlenecks, without CTQO as highlighted by the red circle, overlap with LLRs depicted in the previous figure.

To assess the effects of LLRs on long-tail latency identified in Figure 24, we aggregate these Mysql CPU millibottleneck instances into two groups: those that coincide with queue growth and those that do not. We compare the distribution of request latencies for each group in Figure 26. The plot shows clear differences between these two groups. For the “queue-growth” group, we see more requests, with higher latencies present. The “no-queue-growth” group has fewer long-tail latency requests, and the ones that are present do not exceed 250ms. This distributional difference partly explains why “no-queue-growth”-related requests have not been isolated until now. Requests with higher latencies obfuscate the effects of the “no-queue-growth” requests.

In the fifth step of our analysis, we define LLR requests (Figure 24) as requests that have latencies between 100ms and 500ms. We now increase the magnification to examine a LLR request and associated millibottleneck. Beginning with Figure 27, the graphs show a detailed event analysis [24] of one millibottleneck, one of the MySQL CPU millibottlenecks. This analysis is similar to our past studies of VLRT-millibottlenecks [5] [6] [7] [29] [8] [9] [10] [11] [30].

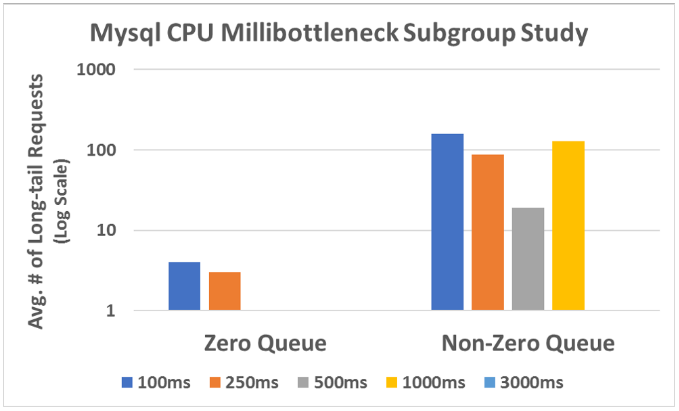


Figure 26 – Mysql CPU Millibottlenecks instances grouped by their associated queue length growth. We observe periods without queue growth have fewer requests and with lower latencies, i.e. 100ms and 250ms, than the periods coinciding with queue growth, i.e. 500ms and 1000ms.

Figure 27 – Figure 29 show a short interval (approximately 0.5s long) from one experiment in our catalogue. The X-axis is consistent across all figures, and it is a timeline delineated by 50ms time periods. The Mysql CPU millibottleneck period of interest, which is 50ms in duration, is indicated by the red-shaded region on each graph, i.e. period 7.

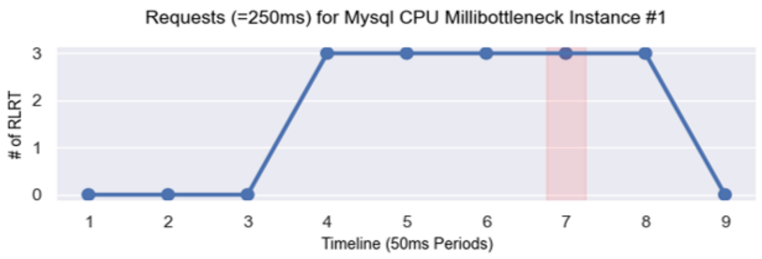


Figure 27 – Number of LLR requests counted at every 50ms time window. These requests overlap with the increase in system response time.

We begin by assessing the number of long-tail latency requests, specifically LLRs, depicted in Figure 27. While not significant in number, e.g. 3 such requests, their emergence corresponds to the period of increased latency. The cluster of LLR requests disappears once CPU utilization approaches 0%. We note that the response time graph, (omitted to conserve space), looks the same as the graph in Figure 27, and the average response time is approximately 5ms.

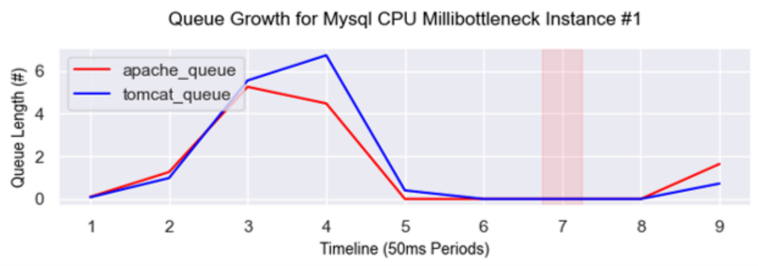


Figure 28 – Queue peaks do not appear during the period of temporary resource saturation. Any queueing that is present is minimal and precedes the short, temporary saturation by at least 100ms.

Figure 28 contains the queue lengths for the upstream Apache and Tomcat components, indicated by the "red" and "blue" lines, respectively. The queue graph over this experiment interval presents an interesting picture. We see some minimal queue growth prior to the period of interest, but it disappears almost 150ms before the temporary resource saturation begins. We also note Mysql CPU utilization never exceeds 40% during that period. We also see during the millibottleneck no observable queue growth in either upstream component despite Mysql being temporarily saturated as shown in Figure 29.

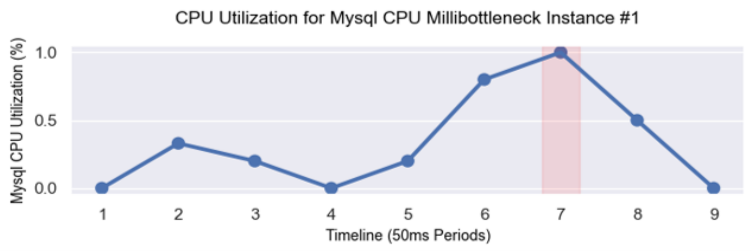


Figure 29 – Transient CPU saturation co-occurs with the appearance of LLR requests and increases in system latency without any queue peak overlap.

We summarize our discussion of LLRs with the following observations. First, we have shown empirical evidence documenting the existence of LLRs with a micro-event analysis. Secondly, we demonstrated their membership in the latency long-tail, i.e. by inducing requests with latencies of 100ms or greater. Finally, we provided a population study to explain how the effects of higher threshold tail-latency requests can eclipse the LLRs.

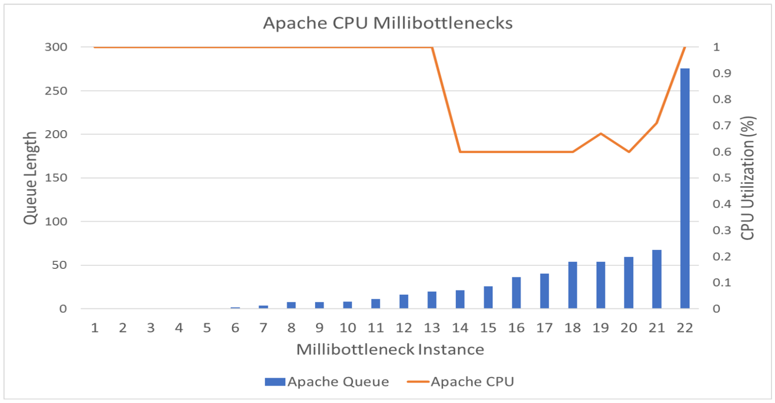


Figure 30 – Apache CPU Millibottlenecks coinciding with VLRT requests and Apache queue growth. No preceptible queue increase occurs during the first four millibottlenecks.

In the sixth step of our analysis, we look at the CPU millibottlenecks associated with other servers. We begin by looking at Apache CPU millibottlenecks pictured in Figure 30. This graph is like Figure 25 in its construction. Each bar along the X-axis represents each instance's corresponding queue growth whereas the line graph indicates the Apache CPU utilization during each millibottleneck. Once again, we see several instances do not coincide with any observable queue increases. To conserve space, we do not zoom into these instances. This result suggests LLR requests are not component-specific phenomena.

We use graphs similar to those employed for previous components' CPU millibottlenecks to examine the Tomcat and CJDBC CPU millibottlenecks in Figure 31 and Figure 32, respectively. In Figure 31, we again see a few millibottlenecks that are not associated with an observable increase in the Apache queue length notwithstanding higher Tomcat CPU utilization.

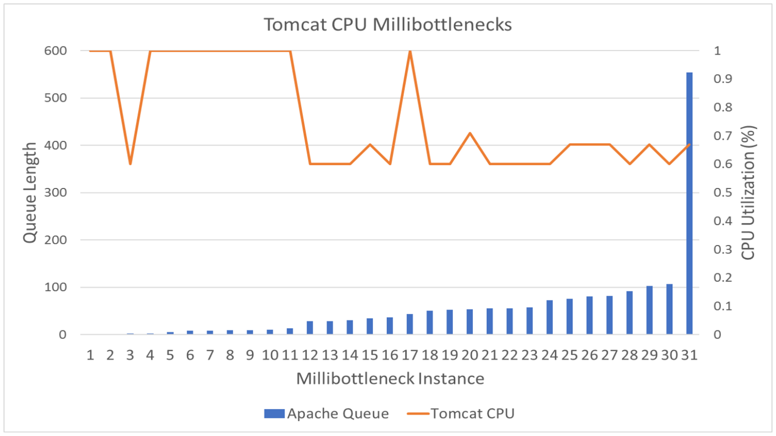


Figure 31 – Tomcat CPU millibottlenecks coinciding with VLRT requests and upstream Apache queue growth. Once again, we see a few instances where no queue increase occurs.

CJDBC CPU millibottlenecks in Figure 32 all coincide with queue growth in upstream Apache and Tomcat queues. This finding suggests LLRs are either being masked by longer latency requests or not materializing—a potential for future work.

4.4.3 Disk and Memory Millibottleneck Subgroups

In the seventh step of our analysis, we perform similar subgroup analyses for Disk and Memory millibottlenecks. We begin by examining the Mysql Disk millibottleneck that induced VLRT requests. Like our previous CPU millibottleneck subgroup plots, we graph the queue lengths for Apache and Tomcat components corresponding to each millibottleneck instance along the X-axis in Figure 33. The line in the figure denotes the Mysql Disk utilization during each millibottleneck. We observe each instance overlaps with queue length growth and high disk utilization. For these millibottlenecks, CTQO explains the appearance of VLRT requests during these millibottleneck periods. Like the CJDBC CPU figure, Figure 33 does not provide visual evidence of any LLR requests. Again, this could be due to higher latency requests obscuring the LLRs.

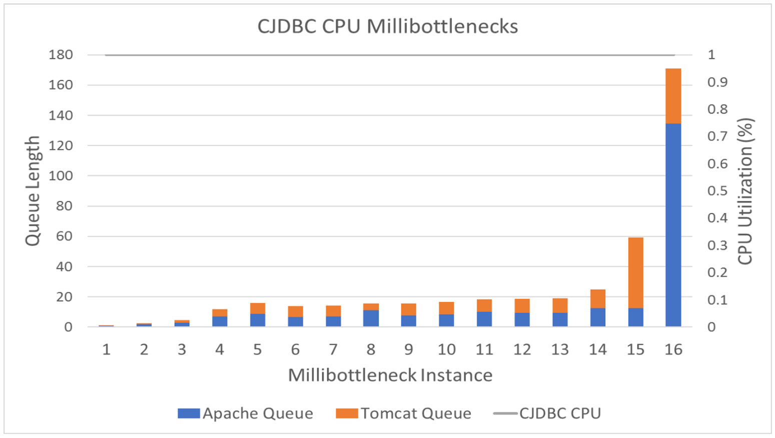


Figure 32 – CJDBC CPU Millibottlenecks coinciding with VLRT requests and Apache and Tomcat queue increases. All millibottlenecks co-occur with queue increases.

We conclude our subgroup studies by examining the case of millibottlenecks due to Dirty Page flushes on Tomcat illustrated in Figure 34. This graph is a different than the previous. The Apache queue corresponding to each millibottleneck still appears on the X-axis. However, it contains two lines representing the Tomcat CPU and the number of Dirty Page, respectively. We use these pair of resources to show the temporary resource saturation that happens at the time of the millibottleneck. As with the Mysql Disk case, queue increases correlate with these instances. As such, CTQO seems to explain the VLRTs associated with each of the millibottlenecks.

In summary, we have provided experimental evidence highlighting two classes of millibottlenecks:

- Those that cause VLRTs because of CTQO

- Those that cause Localized Latency Requests, which we found manifest at lower latency thresholds without dropped packets in CTQO.

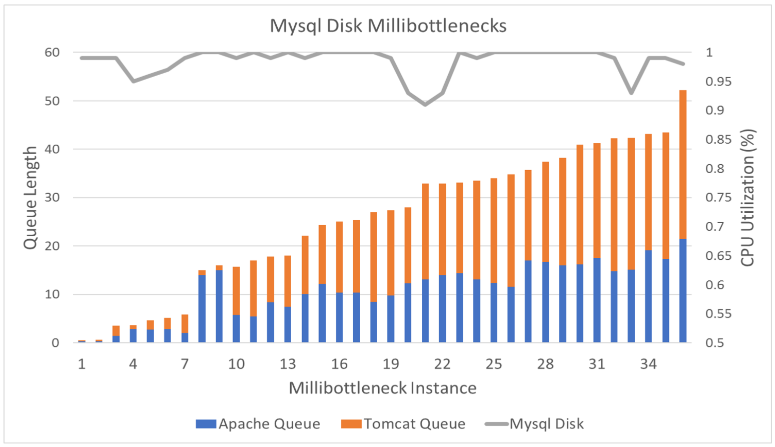


Figure 33 – MySQL Disk millibottlenecks coinciding with VLRT requests and queue growth. We see upstream queue growth overlapping with all millibottlenecks, suggesting LLR are not present.

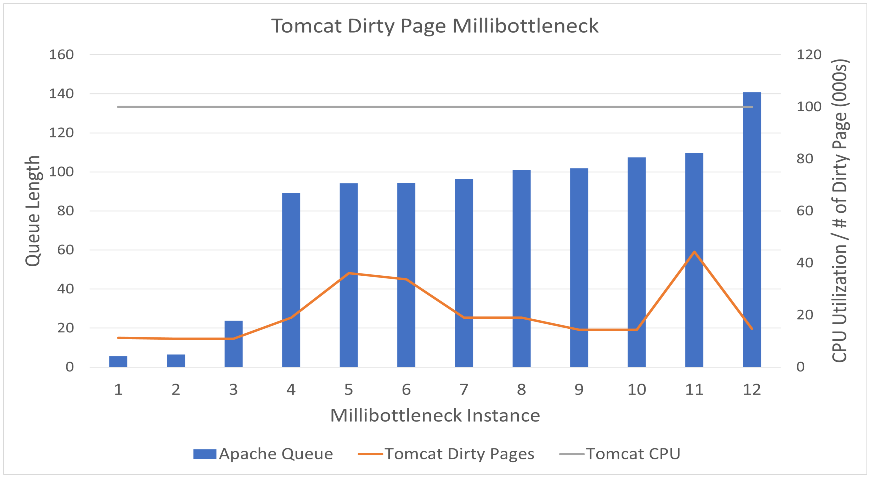


Figure 34 – Tomcat Memory (Dirty Page) millibottlenecks co-occurring with VLRT requests, and Apache queue increases occur during each.

4.5 Future Work and Conclusion

Before concluding, we discuss several opportunities for future work. Our robust measurement schema and flexible approach for handling the schema variety inherent to performance logs enables us to extend PerfDB to support systems experimentation on microservices and streaming systems. The data management infrastructure provided by PerfDB requires little to no augmentation to support data from other benchmark application systems.

PerfDB has two external system dependencies. First, it needs a supporting database management instance, and it requires access to a file system. Currently, PerfDB infrastructure uses sqlite and the local file system; however, these are user-specified configuration parameters. The current implementation makes no assumptions about the parsers or data pipeline used to process experiment benchmark monitoring data. However, whatever tooling is used needs to produce outputs that comport to PerfDB's well-defined, explicit data retrieval interface.

PerfDB's exposes a data retrieval API implemented in Python. To use the API, the DBMS instance and a pointer to the storage location need to be specified either by the user or through a configuration file. First, experiments retrieved through PerfDB need to be registered in PerfDB's experimental metadata relation. This schema includes attributes that are independent of benchmark or topology. For example, it captures an experiment's beginning and ending times in addition to a file system pointer to an experiment's artifacts. Secondly, the interface specifies three measurement relations that must (at a minimum) be materialized after processing experimental artifacts. A grammar for defining measurement schema attributes is also part of the interface. The data retrieval API uses this grammar to retrieve Resource and Event measurement data from corresponding measurement relations across heterogenous systems' topologies. Finally, PerfDB specifies a millibottleneck schema to persist instances of millibottlenecks. This schema can support materializing millibottlenecks that are found in heterogenous systems' topologies.

Besides extending PerfDB to support additional benchmark systems, we want to explore the population of Localized Latency Requests and their frequency to understand why they can be unaccounted. Lastly, we are currently working on a follow-up paper where we apply machine learning techniques to isolate and diagnose millibottlenecks with maximum recall and acceptable precision despite performance data noise.

To summarize, the two main contributions consists of the experimental verification of Localized Latency Requests and the PerfDB platform that enabled a methodical study of LLRs. First, we define the LLRs as those with latency between 100ms and 500ms. Through a series of population studies and analyses of individual millibottlenecks, we presented experimental evidence that confirm their association with Localized Latency Requests for the first time. Second, we demonstrated how the PerfDB performance data management system enabled those studies on LLRs and large-scale confirmation of ongoing research on the association of millibottlenecks with VLRT requests through CTQO.

CHAPTER 5. PERFML

5.1 Introduction

Internet scale systems long tail latency challenges can negatively impact business. For example, Amazon has reported [2] that a marginal increase of 100ms in page load time correlates with approximately a 1% reduction in sales. Google has reported a similar revenue-latency relationship. It has found that 500ms of additional delay returning search results could reduce revenue by up to 20% [3]. Given the size of their respective customer bases, these platform companies need to reduce their 95th, 99th and 99.9th percentile latencies as close to zero as possible [1] [4].

To mitigate the effects of long tail latency problems, Dean et al prescribed several coarse-grained solutions such as leverage fault tolerance and resource redundancy [1] [25]. While effective, these techniques come at significant cost. For example, NRDC reported that average data center utilization was 12-18% from 2008 – 2012 [26].

The millibottleneck theory of performance bugs explains the latency long tail phenomena as being due in part to requests, which under different resource contexts complete in tens of milliseconds, instead complete on the order of hundreds or thousands of milliseconds [5]. We call these requests that comprise part of systems' response time long tail, Very Long Response Time requests (VLRT). The transient resource saturations, called millibottlenecks according to the theory, are responsible for the appearance of VLRT. We have isolated the root cause of their appearance across the system's stack, including CPU dynamic voltage frequency scaling (DVFS) control [6], Java garbage collection (GC), “noisy neighbors” due to virtual machine consolidation [7], dirty page flushing [8], memory thrashing due to rapid succession of page faults [9] and writing log data to disks [10] [11].

Approaching latency issues by addressing root cause has three important benefits. First, isolating and eliminating performance bugs (in the first place) reduces the system redundancy costs and improves the overall datacenter return on investment.

According to the most recent Berkeley View on Serverless Computing, performance unpredictability is an enormous barrier to adoption for cloud-native architectures like serverless and microservices [28]. It is one of the three most cited reasons practitioners do not migrate application workloads to serverless computing topologies. Mitigating tail latency and identifying its underlying causal mechanisms is critical to achieving broader adoption and better management of serverless and other cloud-native technologies.

Previous work on millibottlenecks have analyzed individual periods or experiments. This unit of analysis is challenging due to these phenomena fleeting nature and the intricate system instrumentation needed to isolate and reproduce them. However, developing more robust explanations and models for these phenomena require studying them as a population, i.e. exploring them across the system experiment configuration space. This introduces another level of complexity related to data management and analytics. For example, data originating from different experiments needs to be integrated across common attributes, and visualizations need to account for many instances of the same type of millibottleneck.

Finally, we believe systematically studying performance bugs as a group can help us to potentially develop new theories or models about system design and implementation. Recently, AWS experienced a prolonged outage due to cascading service failures due to intra- and inter-container dependencies. Fine-grained performance phenomena might be valuable early indicators of larger system problems, i.e. the proverbial “canary in the coal mine.” Studying fine-grained performance phenomena could help to identify the most valuable and important early warning signals of system performance degradation.

This paper describes three contributions. First, we detail how our approach can successfully navigate the challenges inherent to system experimental data. Our second contribution is methodological, which is our ensemble-based classification system, PerfML. We demonstrate how we use our domain knowledge to partition our sample space to reduce the heterogeneity of the feature space. We also show how we convert a fine-grained system performance anomaly detection problem into a traditional supervised multi-class classification problem. Moreover, we explain how we transform this problem into two separate decision problems that enable us to use a hierarchy of binary, *one-versus-all* classifiers, which mitigate the impact of multiple, imbalanced classes in our sample space.

Our third and final contribution is an evaluation of PerfML across a number of dimensions. First, we assess its ability to successfully detect and diagnose known types of millibottlenecks. Secondly, we explore the effect of different choices of classification learning algorithms for the base learners on overall ensemble classifier recall, precision and F1. We also explore the effect of various decision thresholds on these metrics. We conclude with a comparison of PerfML to other, current system performance analysis approaches, and we highlight PerfML's utility with a demonstration on an unlabeled data set.

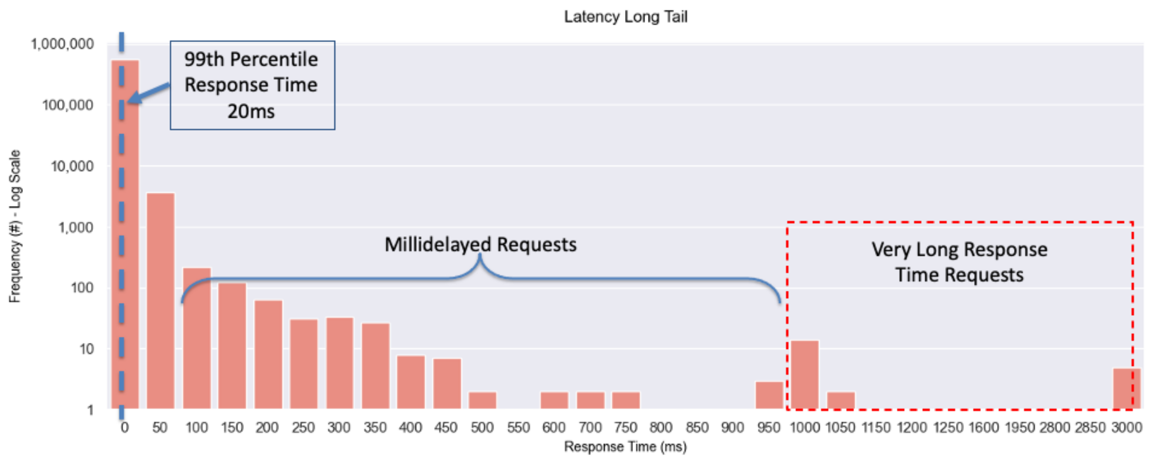


Figure 35 – Latency long tail of experimental periods in our data catalogue. We see the appearance of VLRT and LLR with 99% periods completing 20 ms or less.

5.2 Problem Definition

5.2.1 Latency Long Tail Request

The response time long tail consists of requests that take more time to return than the 99th (or 95th) percentile of system latency. Some of these requests take this long due to request-specific characteristics, for example, a search query composed of rare terms. VLRT requests, which arise due to queueing effects, are the other set of requests comprising the latency long tail that complete in hundreds or thousands of milliseconds but complete in tens of milliseconds when they are executed separately.

Our group's previous work focused on the study of individual millibottlenecks, extended to the millibottlenecks of the same class, (the same contended resource), that caused VLRT requests [5] [6] [7] [29] [8] [9] [10] [11] [30]. In this work, we conduct a series of population studies, and explore a phenomenon called Relatively Long Tail Requests (LLR). Unlike VLRT, LLRs are not associated with queue growth, but as depicted by the red dashed region in Figure 35 are still apart of the latency long tail. We define LLR more formally later.

Long tail latency. To estimate system latency for each sampling period, we use an aggregate measure called *Point-in-Time Response Time (PIT).* This measure of system latency is defined by the wall clock time requests take to complete a system round trip. To calculate this metric, we average the round-trip time of all requests beginning within a specified interval. In our case, we use fixed width 50ms intervals.

5.2.2 Millibottlenecks

We model short resource saturations, order of tens of milliseconds, through the concept, Millibottlenecks. They materialize due to shared system resource competition. Because of system dependencies, these short contentions can propagate and become amplified by contributing to the long-tail latency of systems in the form of VLRT requests.

We define a Millibottleneck instance as a tuple: {Start Time, Duration, Contended Resource, and Location} where each attribute is defined as follows:

- *Contended Resource* is a temporarily, highly utilized resource, e.g. at least 60% utilization.
- *Location* marks the component experiencing resource contention.

- *Start Time* marks the beginning of the resource contention, and *Duration* denotes the amount of time the contention endures.

To determine the beginning and end of millibottleneck periods, we employ a simple heuristic. If two consecutive data points exceed a saturation threshold, they are grouped together. We continue the grouping process until a data point does not meet the provided saturation threshold, i.e. the system has transitioned out of a saturated state.

5.2.3 Detection and Diagnosis Problem

Using system performance logs, we want to identify when and where fine-grained performance anomalies like Millibottlenecks occur. Moreover, we want to isolate the root cause, i.e. the component and resource experiencing transient resource contention.

Labels. We have found millibottlenecks can occur for a variety of reasons, including CPU dynamic voltage frequency scaling (DVFS) control [6], Java garbage collection (GC), “noisy neighbors” because of virtual machine consolidation [7], dirty page flushing [8], memory thrashing due to rapid succession of page faults [9] and writing log data to disks [10] [11]. Systems’ nondeterministic execution provide the statistical circumstances for millibottlenecks to materialize, i.e. very short resource contentions materialize due to the “right” sequence of system events occurring at “just the right time.”

The diversity of millibottlenecks and the noisy process surrounding their appearance has motivated us to explore machine learning techniques. As such, we pose the problem of diagnosing millibottlenecks across diverse experimental systems’ topologies as a single-label, multi-class classification problem. Each type of millibottleneck corresponds to its own label.

In traditional supervised machine learning problems, we would train one monolithic model capable of multi-class classification for this detection and diagnosis problem.

However, we choose to separate our decision problem into two sequential classification problems—detection then diagnosis. This approach imposes a hierarchy on the resultant classes. Knowing that a millibottleneck occurs can be used to trigger diagnosis. This approach enables us to use sets of features that are well suited for each sub-problem. For example, topology-specific features can provide strong signals for knowing some type of millibottleneck is occurring. Conversely, component-specific features, which are different across components, are important for diagnosing the specific type of millibottleneck. Splitting the problem into two enables us to construct specialized models for diagnosing millibottlenecks specific to a component. Another, important byproduct of this partitioning is it mitigates the heterogenous input data problem mentioned earlier. Without this separation, a single model would need to either remove features, which could be important to diagnosis, or provide a method for interpolating missing values, potentially leading to decreased accuracy.

5.3 Data Description and Challenges

We conduct experimental systems research using benchmark application systems. We have executed tens of thousands of experiments, and Table 8 describes some of the key dimensions of this large experimental systems catalogue. It consists of experiments from executions of a Reddit-style bulletin board system named RUBBoS, which is a monolithic, n-tier application benchmark. It relies on Apache, Tomcat, CJDBC and MySQL components. We deployed our benchmark application in various topological configurations, which are depicted under the "Topology" heading. The numbers listed in table correspond to the number of deployed Apache, Tomcat, CJDBC and Mysql components, in that order. These experiments were executed for 3 minutes, and they featured workloads ranging from 100 to 20,000 concurrent client sessions

Table 8 – PerfML Experiment Catalog

Measure	Value
# of Experiment Periods	1.2M+
# of Millibottleneck Periods	30,000+
Performance Data Size (GB)	2000
Benchmark	Rubbos
Benchmark Workload Range	100-20,000
Benchmark Runtime (Avg.)	3 mins
Hardware	p3000, d430, d710
Topologies	111, 1111, 1112, 121, 441
Components	Apache, Tomcat, CJDBC, Mysql

Note: This table is an augmented version of Table 4.

5.3.1 Input Data

As detailed in 2.2, we decorate the experimental systems benchmark application referenced in Table 8 using a collection of event and resource monitors, i.e. sensors. These software monitors record system events and state in logs at fine-grained time scales for each benchmark execution. Then, we transform these log data into temporal relations such that each observation at time, t, is a m-dimensional vector where each dimension corresponds to a measurement from a particular monitor, for example, the CPU utilization for one core for one physical host at time *t*. When one of several types of millibottlenecks occur at time t, we represent each one with its own label, c_i, from a set of C class labels. We include in this set a special label to accommodate periods when the system does not experience millibottlenecks.

Stated formally, each observation, $\overrightarrow{x_t}$, at time, *t*, is denoted $(\overrightarrow{x_t},\ y_t)$ where $\overrightarrow{x_t}$ is an element of an M-dimensional feature space and y_t represents the target class label, c_i. Concatenating individual experimental data sets together results in a *series* of finite, multivariate timeseries. We denote this collection as **X**.

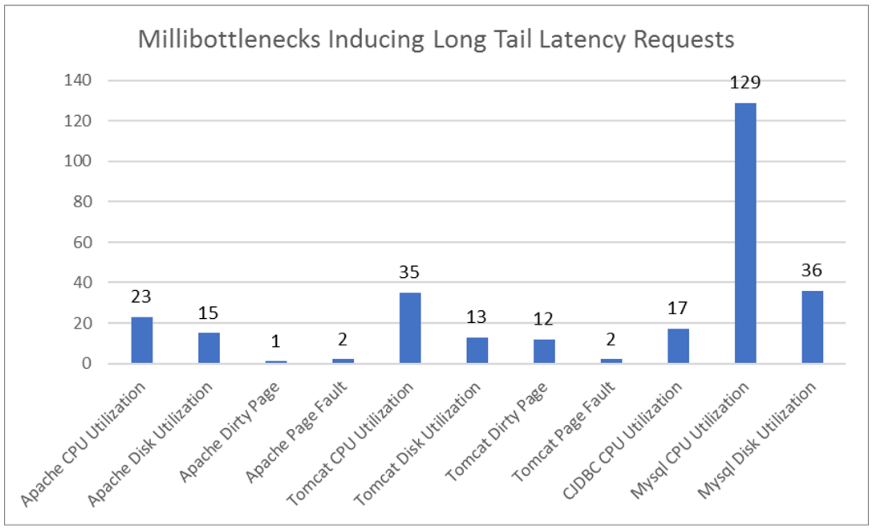


Figure 36 – Millibottlenecks found in our experiment collection stratified by type that induce tail latency requests represent imbalanced classes in our data.

5.3.2 Imbalance Multi-Class Hierarchical Membership Data

Our millibottlenecks are not uniformly distributed in our experimental data. As shown in Figure 36, some millibottlenecks appear substantially more often than others. In addition, we need the ability to detect millibottlenecks that induce latency requests like

VLRT and LLR and those that lead to no perceptible impact on latency. Figure 37 differentiates Millibottlenecks stratified by their co-occurrence with VLRT. Once again, we see a significant difference in the frequencies between these two situations. Consequently, a robust millibottleneck detection approach cannot solely rely on the latency of requests as a signal for millibottlenecks. In addition, knowing that a millibottleneck occurs (over some interval) is an important prior to diagnosing its root cause. For the purposes of classification, this means the positive class representing the presence of millibottlenecks is the parent to each class representing each type of millibottleneck.

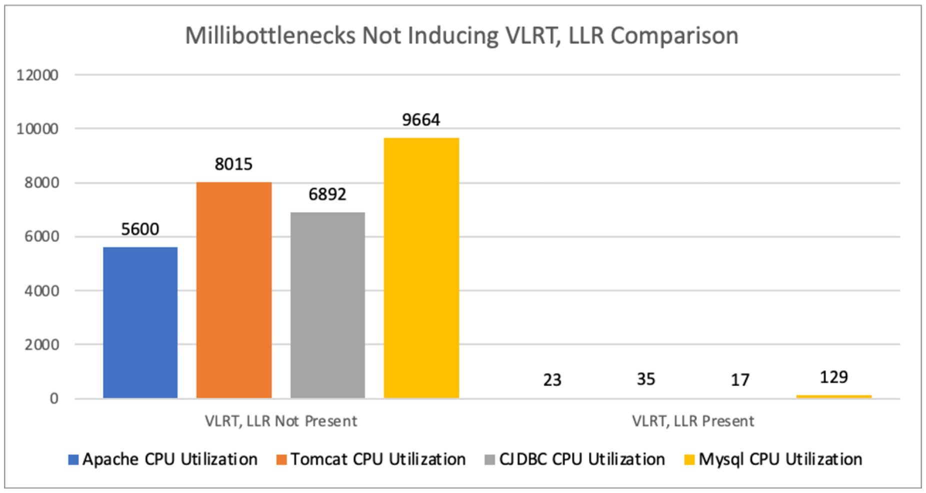


Figure 37 – Incidence of Various Millibottlenecks stratified by the presence of VLRT

5.3.3 Heterogeneous Feature Space

PerfML encompasses a large experimental systems data space as indicated by Table 8. This large configuration space leads to experimental schema diversity across the collection. Figure 38 shows the distribution of experimental schema in our sample space, **X**. We see 347 different experimental schemas are represented across 380 experiments.

Only one schema has significant density with 42 experiments sharing the same schema. This variability is due to a variety of reasons.

First, the number of resource-related attributes might change from machine to machine due to architectural and kernel disparities across machine types despite using the same monitor, version and bootstrap parameterization. Secondly, drift may occur due to experiment-specific utilization. For example, some monitors might omit metrics for underutilized resources. In the context of single machine learning models, schema diversity leads to the missing features problem.

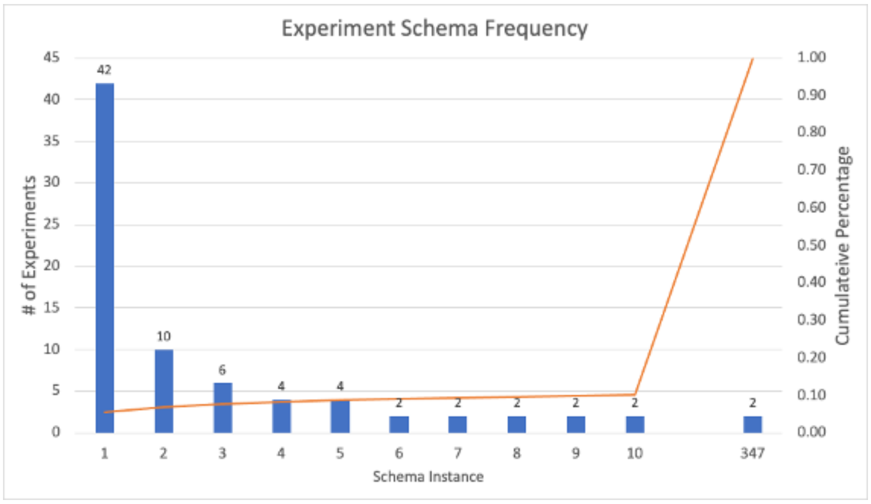


Figure 38 – Schema distribution for experiment samples X. The blue boxes represent the number of experiments that share the give schema. The orange line represents the cumulative distribution of schema instances.

Missing Features Example. To highlight this issue, we consider one experiment might use machines with 2 CPU cores while another one might use machines with 8 CPU cores. The experimental space spanning both experiments would include "NA" values for CPU cores 3 thru 8 for the first experiment. Prior to using one machine learning model to train over the data in this space, these values would need to be imputed or the attributes removed. Removing the attributes might impair the model's ability to detect CPU millibottlenecks related to contention experienced by cores 3 thru 8. Imputing values could lead to erroneous predictions related to experiments with 2-core machines.

Besides the heterogeneity due to differences in resource monitoring, the systems topologies can be a factor. As highlighted by the table, experiments can range in the number and kinds of nodes deployed. In addition to this topological range, systems can have entirely different architectural patterns. For example, microservice-based applications rely on the composition of many, fine-grained services to answer client requests or complete transactions. Even though the systems currently contained in the catalogue have different numbers of nodes and components leading to differences among experimental schema, they all adhere to a monolithic architectural pattern.

We capture our domain knowledge about experimental system composition as a Directed Acyclic Graph (DAG). We term this graph the "System Stack Graph," and it is meant to capture the layered design of cloud-based systems as shown in Figure 39. As we discussed, systems experiment samples originate from a variety of components arranged in a particular sequence based on the function they provide. They can have different numbers and types of nodes depending on application needs. The graph captures the relationships among system-specific topologies, their component composition and their monitored resources.

We use the concept of a "level" to describe a group of nodes that are equidistant from the root. The first layer in the figure differentiates systems based on their *topology*, i.e., the quantity of tiers and nodes and the dependencies among them. For example, a three-tier topology features a web server, an application server and a database server in that order.

The next level differentiates topologies according to their constituent *components*. For example, Apache is a web server, and it can be used for this function in three-tier, four-tier or even microservice-based topologies. Since components can be used in multiple different topologies, we use a graph-based representation instead of a tree to capture the topology-to-component relationship accurately.

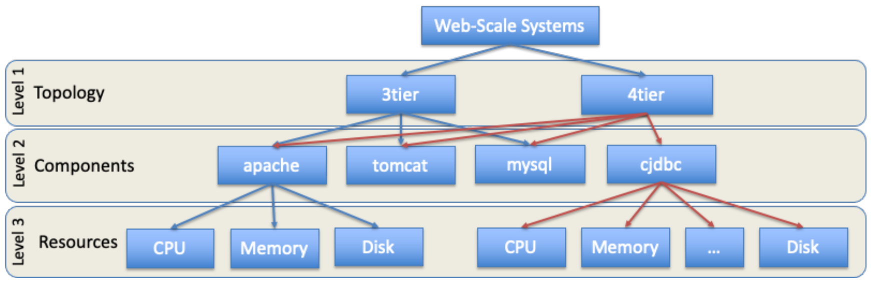


Figure 39 – System Stack Graph representing the systems topologies, components and resources spanning experiment samples X.

The last level describes the potential resources that can be monitored in our systems experiments, for example, CPU and Disk utilization, Memory consumption and network usage to name just a few. Since components have different execution models and use resources differently (not to mention different monitoring choices by component) result in components have their own monitored resources "neighbors."

5.4 PerfML Approach

5.4.1 Overview

PerfML pictured in Figure 40 has three parts: millibottleneck schema, perftables and a data management platform we call perfstore. We begin by describing the schema necessary for experimentally verifying millibottlenecks and their relationship to VLRTs and LLRs.

5.4.2 perftables and PerfDB

We use *perftables* relation induction approach to automatically transform the performance log data output by our infrastructure's event and resource monitors into relational structures [31].

PerfDB ingests these relations for each experiment. PerfDB provides a data management interface that supports over 400 unique system experiments comprising 2TB of performance data and over 600 different performance attributes across these systems. Within this data set, we have identified over 30,000 millibottleneck periods—occupying about 2% of all experimental time periods.

Despite these cross-experiment schema complexities, PerfDB's millibottleneck data model provides a convenient data abstraction for isolating, diagnosing, and studying millibottlenecks and their association with the latency long-tail as we detail in the following section. PerfDB has a metadata schema to store pointers to four experiment-level schemas: one containing *latency* information, another containing components' *queue lengths*, another containing *resource metrics* and the final one stores identified *millibottleneck instances*. We observe the first three relations provide the necessary attributes (time, duration, component and contended resource) for materializing tuples in the millibottleneck table. Moreover, they provide latency-related information to study the relationship between millibottlenecks and latency long-tail phenomena like VLRT. Finally, this common set of relations for each experiment's data provides a convenient set of abstractions for querying and returning partitions of the entire experimental sample space to PerfML.

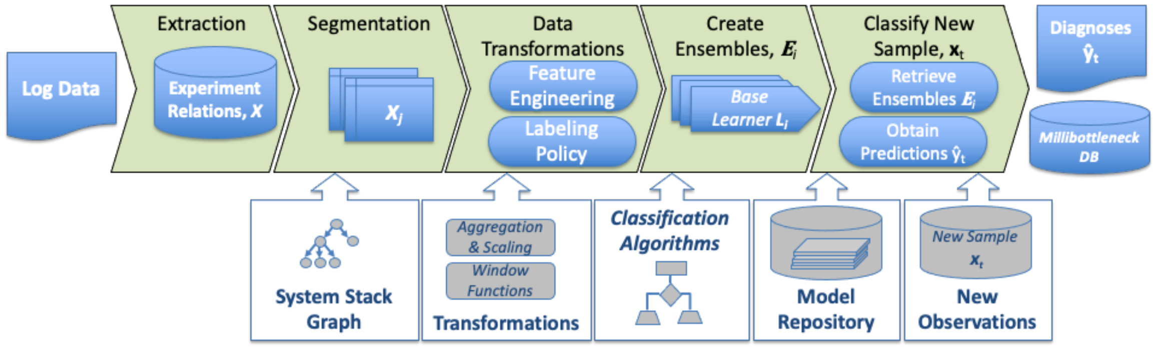


Figure 40 – PerfML data flow. Event and Performance Monitoring Logs are extracted into relations. Using the system stack graph, partitions of the experimental space are created. We transform the data to support training ensembles over each partition. We traverse the system stack graph and retrieve appropriate models to (i) isolate millibottleneck periods (ii) diagnose the type of millibottleneck respectively.

5.4.3 Segmenting Experimental Data

The traditional approach in machine learning to train a classifier, M, is to use the entire data set represented by the root node in our graph. Instead, we propose sub-dividing the sample space by vertically partitioning it using the attributes described by each of the vertices in the System Stack Graph (SSG).

We use the System Stack Graph to guide the partitioning of the sample set **X** into multiple subsets. We note that this graph describes systems currently within our experimental catalog; however, it can be extended to include new topologies, components and resources.

For the *topology* level, we generate subsets $\boldsymbol{X}_{ij}$ for each topology where j indexes the vertex that represents a set of attributes on level i. We use the notation $\boldsymbol{X}_j$ when it is obvious from context which vertex j to which we are referring. By vertically partitioning the sample space, we are able to reduce the heterogeneity of the resource and event observations.

This first partitioning can be thought of happening across the experimental topology dimension of **X**. Experiments with topologies that match the topology-level vertex are selected first. For example, by considering the three-tier topology on its own, the samples in the relevant subset $\boldsymbol{X}_j$ will have data for Apache, Mysql and Tomcat. However, $\boldsymbol{X}_j$ would not contain data for CJDBC, since no middleware component comprises a three-tier topology. Moreover, each subset $\boldsymbol{X}_j$ no longer contains classes that are irrelevant to the system type, i.e. no CJDBC CPU, Disk or Memory millibottlenecks in the three-tier sample space. In the original space **X**, three-tier topology samples would have "NA" for the dimensions in the feature space related to the middleware component, e.g. CJDBC. As such, we observe that the heterogeneity of the feature space for this $\boldsymbol{Xj}$ is less than in the original **X**.

To partition the sample space **X** using the System Stack Graph, we traverse the graph using Depth First Search. As such, the next partition happens across the deployed component dimension of **X**. This involves vertically segmenting the experiment space into component-specific subsets. Experiments with the component matching the component-level vertex are then selected. Returning to the example, experiments, regardless of topology, using the Apache web server would be included in $\boldsymbol{Xij}$. We generate a sample subset $\boldsymbol{X}_{ij}$ for each node on each "level" in the graph.

The final partition is another vertical segmentation. This time the partition occurs across the monitored resource dimension of X. Experiments with monitored resources matching the component-specific resource vertex are then selected. In the running example, experiments where Apache CPU resources were monitored would be included in this partition of X.

A partition may contain only 1 experiment and within that experiment it many only have a few periods when a certain millibottleneck occurs. Conditions for training the classifier includes there must be at least 2 experiments and at least a minimum number of samples are millibottlenecks. We use a threshold, *s*, to indicate this minimum percentage of samples that contain the millibottleneck of interest. Only sub-spaces that meet this minimum threshold are retained. We handle the case of a sample subspace, ***Xj***, not being retained for vertex, j, during ensemble training.

5.4.4 Feature Engineering

Prior to training a classifier, PerfML performs additional pre-processing on each ***Xj*** in the form of expert feature engineering. Features are *M*-dimensional vectors resulting from the application of transformation functions to experimental schema attributes. These expert features fulfill three requirements.

First, they fulfill specific technical requirements for machine learning classification algorithms. For example, we remove *sparse*, *no variance* and *linearly dependent* features, and we apply standard scaling procedures like *minmax* scaling. Secondly, these features provide important signals for isolating and detecting millibottlenecks. Finally, our features reduce the feature space heterogeneity.

To reduce the feature space heterogeneity, we apply aggregate functions, such as *min*, *max* and *mean*, to semantically similar attributes. For example, a sample from a 2-core and 8-core node would have a CPU Utilization measurement for each of its cores. We can combine these samples by only using the maximum core utilization level from each sample. Applying an aggregation function to each set of semantically similar attributes like CPU Utilization enables us to train models using samples from nodes hosting the same component.

Time Windows. The last step in the pre-processing involves creating time windows over the time series, i.e. *upsampling* the time series or reducing its frequency. We transform $\overrightarrow{x_t}$ by applying an aggregate function to a sequence L for the i^{th} feature, $[x_{i,t-L+1}, \ldots, x_{i,t-1}, x_{i,t}]$. We up-sample each of our features to 100ms.

5.4.5 *Creating Ensembles E_j*

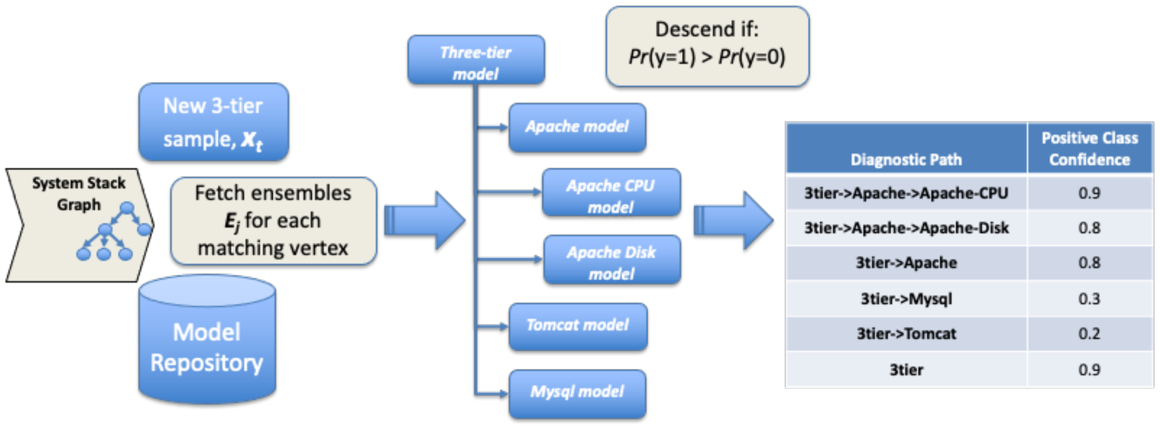
We train a classifier, M_j, for each experimental sample subspace, X_j. While we could leverage any supervised classification algorithm, Random Forests have some attractive properties for our problem domain.

First, they are robust to features with different ranges, which enable them to be trained on unscaled data. They can emit probabilistic outputs. Probabilities are a reflection of the classifier confidence in its class label prediction. Tree-based methods provide comprehensibility and transparency into their predictions. This makes them a good choice where predictions need to be able to be interpreted by analysts. Ultimately, we want to understand why a millibottleneck occurs, so having an understanding into the classifier's decision or prediction could facilitate root cause analysis. We see from the frequency of millibottlenecks in Figure 36 that there are relatively few memory-related millibottleneck samples. Random Forests integrated sampling techniques reduce the chance of overfitting in this case despite the small number of positive examples.

To train a model, M_j, for each subset X_j, we once again traverse the SSG using DFS. This time we retrieve the sample subset for each vertex. We train one base learner, L_j, for X_j. For instance, we train one base learner for Apache CPU for data from experiments in which Apache CPU millibottlenecks occurred. We note each subspace only contains two classes.

Sibling Labeling Policy. Prior to training, we re-label all periods using a sibling labeling policy. Under this policy, a vertex and its neighbors are considered to be members in one class while vertices at the same level (and their neighbors) are considered members of another class. Using the sample space for Apache, $\boldsymbol{Xj}$, as an example, a sibling policy results in all periods with Apache millibottlenecks to be labeled with the positive class, and periods with either no millibottleneck or another type of millibottleneck to be labeled with the negative class [CITE Silla]. As such, we have converted our multi-class supervised classification problem into a series of *one-versus-all* binary classification problems.

A model, M_j, is trained and retained if and only if a partition exists for X_j. We store the trained ensemble E_j in our model database. We store the index j for the associated j^{th} vertex along with the model, so that it is associated with the appropriate system stack abstraction layer. We also retain the label of the system stack abstraction layer in addition to the constituent vertex's label.



Diagnostic Path	Positive Class Confidence
3tier->Apache->Apache-CPU	0.9
3tier->Apache->Apache-Disk	0.8
3tier->Apache	0.8
3tier->Mysql	0.3
3tier->Tomcat	0.2
3tier	0.9

Figure 41 – Computing diagnostic paths for a new sample from a 3-tier system topology. Ensembles for each vertex in the system stack graph are retrieved for each matching vertex. Depth-first search continues as long as the positive class probability exceeds the negative class probability, and diagnostic paths are ordered according to their length and confidence.

5.4.6 Classify New Samples

Given a new observation, we begin by traversing the System Stack Graph in a depth-first fashion to determine the appropriate system stack abstraction *j* to which the observation belongs. The sample belongs to a particular system stack abstraction *j* if the features for sample, x_t, match all of the attributes described by vertex, v_j. For example, a sample from a 3-tier topology would match all of the attributes associated with the 3-tier vertex. Samples that do not match any vertex are disregarded. Next, we retrieve the ensemble, E_j, from our model database. When scoring a new sample, we need to consider two possible scenarios: an ensemble, E_j, exists and an ensemble does not exist for a given vertex.

In the first situation, we pass the sample, x_t, to the base learner(s) of the retrieved ensemble, E_j, and it returns the probability the sample belongs to the positive class. If the positive class probability is greater than the probability for the negative class, the depth first search continues. The traversal enables us to capture the base learners' predictions. During the traversal, the paths of vertices where the positive class probability exceeds the negative class probability are recorded.

Descent continues until the negative class probability exceeds the positive class probability. When a negative class probability exceeds a positive class probability, none of the vertices' neighbors are added to the traversal stack, and the neighbors are marked as having been visited. The traversal concludes once all vertices have been visited.

The other scenario is when no ensemble, E_j, has been trained for a vertex, v_j, to which the sample belongs. In this situation, the descent halts and all neighbors are marked as having been visited.

5.4.7 Obtain Diagnosis

PerfML has two output modes. It can diagnose the most likely prediction or millibottleneck, and it can return the top *k* predictions. In the previous step, a depth-first traversal is used to identify paths where the ensembles associated with vertices yielded positive class probabilities that exceeded those of the negative class. We refer to this collection of paths as "diagnostic paths."

To diagnose a millibottleneck among the possible diagnostic paths, we order the diagnostic paths in descending order according to their length and each path's terminal vertex probability. For the purposes of our evaluation, the most likely millibottleneck diagnosis corresponds to the terminal vertex for the top ranked diagnostic path. In this respect, the terminal vertex corresponding to the highest probability prediction is the most confident prediction, controlling for path length.

No Millibottleneck diagnoses (or predictions) have a path only containing the root vertex. In this case, the probability associated with the terminal vertex of this special path is the negative class probability returned by the sample's matching topological ensemble.

Besides the top ranked path, PerfML can return the top *k* paths along with the probabilities of the paths' intermediate vertices. All of the predictions and paths are persisted in the PerfML's millibottleneck schema. The top ranked prediction is materialized as a millibottleneck instance.

Table 9 – Data Set Metrics

	Evaluation Data Set Description		
Data Set	***# of Experiments***	***# of Observations***	***True Positives (Millibottlenecks)***
Wise-rubbos-beta	300	1,100,000	~30K
Elba-rubbos	3	10,800	~400
WedMake	90	324,000	--

5.5 PerfML Evaluation

We use three different data sets to evaluate the effectiveness of PerfML. These data sets are described in Table 9. The "Wise-rubbos-beta" data set is unique, because labels were able to be determined using a combination of expert (and deterministic) rules and manual inspection. We implemented a set of labeling functions to bootstrap our learning models, and we applied them to this data set where we knew *a priori* that millibottlenecks, which could be identified with these functions, existed. Not all millibottlenecks can be labeled in this manner and require manual investigation.

The “Elba-rubbos” data set contains a small number of systems’ experiments. Consequently, we were able to hand label all the millibottleneck periods. The final data set, “WebMake” contains a modest number of systems’ experiments. These experiments were similar in topology to those in the first data set, but unlike those, these lacked the sufficient metadata like JVM version to apply our labeling functions accurately. Given the volume of observations, hand-labeling could not be accomplished efficiently.

We assessed classifier performance using standard classifier evaluation metrics: *Precision*, *Recall* and *F-measure* (F1), which is the harmonic mean of Precision and Recall. *False Negatives (FN)* for Millibottleneck detection occur if a millibottleneck was NOT predicted to occur and it actually occurred during a given time period. *False Positives (FP)* for Millibottleneck detection occur if a millibottleneck was predicted to occur and it did NOT actually occur. *True Positives* (TP) occur when the prediction and actual ground truth indicate millibottlenecks occurred.

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F\text{-}measure = \frac{2*Precision*Recall}{Precision+Recall}$$

In the context of our decision problem, missing a millibottleneck when it occurs (False Negative) is more costly than a False Positive. The only cost of a False Positive is analysis time; however, analyzing millibottlenecks with PerfML is at least an order of magnitude faster than our prior graphical and basic statistical methods.

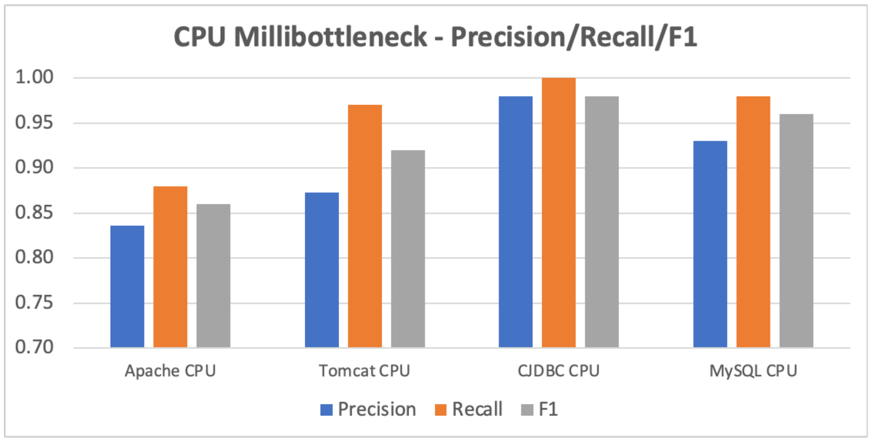


Figure 42 – PerfML Ensemble Precision, Recall and F1 Performance on Test Set for CPU Millibottleneck Diagnosis across components.

5.5.1 PerfML Performance

Our first evaluation involves assessing the performance of PerfML in diagnosing known millibottlenecks. For this evaluation, we use the "Wise-rubbos-beta" data set, which is described earlier. Since our experiments are finite, multivariate time series data sets, we must preserve its temporal ordering when splitting our sample space for training and testing. As such, we randomly split the data set at the experiment level into 80% for training and 20% for testing and validation. Then, we use the training set to train PerfML and its constituent learners. Next, we tabulate the Precision, Recall and F1 metrics using PerfML's predictions on the test set. Finally, we repeat this process for 10 trials and calculate the average Precision, Recall and F1.

Figure 42 shows PerfML's average test set across all 10 trials. PerfML achieves high recall for diagnosing different types of millibottlenecks while maintaining reasonable precision. In all cases, its recall exceeds 88% and precision is never less than 84%. From this, we can reasonably conclude PerfML can diagnose millibottlenecks with high recall.

5.5.2 PerfML Parameters

Next, we assess how the choice of learning algorithm impacts PerfML's aggregate performance. For this evaluation, we used the same data set, Wise-rubbos-beta, and training and testing procedure as the first evaluation. The following describes the hyperparameters for each type of supervised learning algorithm we tried.

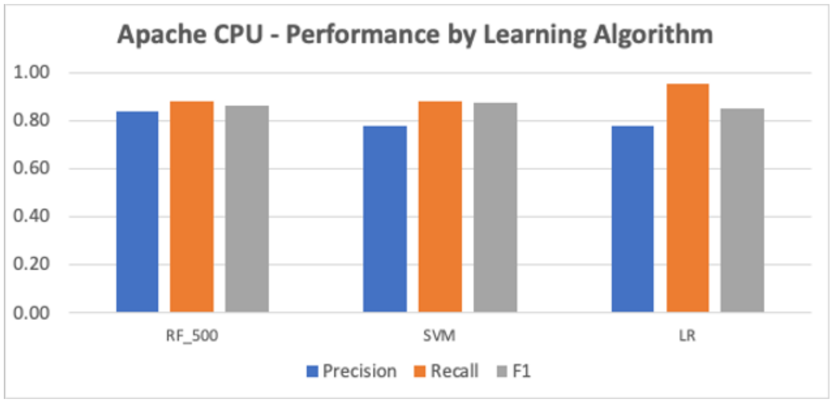


Figure 43 – Precision, Recall and F1 Performance on Test Set for Apache CPU Millibottleneck Diagnosis for different choices of base learners.

Logistic Regression. We used a linear search to set an optimal λ, the penalty parameter for an L2-regularized model. We report the results for the value that provided the best results.

Support Vector Machines. We tried multiple kernels, including linear and RBF, and the RBF kernel provided the best results for our evaluation.

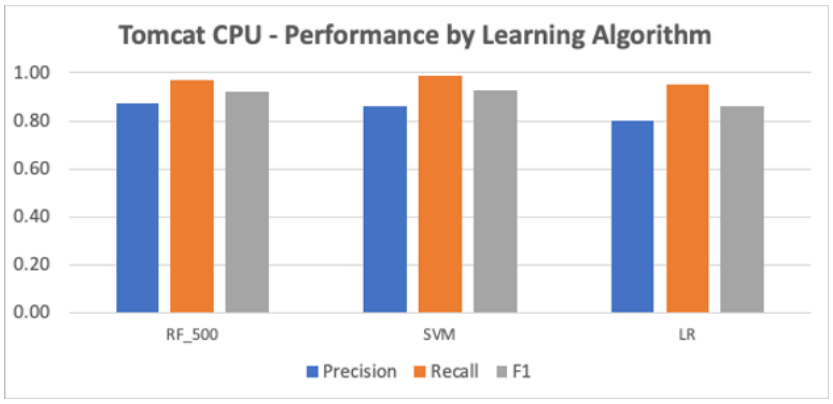


Figure 44 – PerfML Precision, Recall and F1 Performance on Test Set for Tomcat CPU Millibottleneck Diagnosis for different choices of base learners.

Random Forests. The number of trees, the minimum samples per leaf, and the number of variables randomly sampled for building a tree are three important hyperparameters. We used the Akaike Information Criterion (AIC) to search for optimal parameters, and we found 500 trees and a minimum of two samples per leaf provided the best results. We used the square root of the feature space size, a commonly used heuristic, for the number of sampled features.

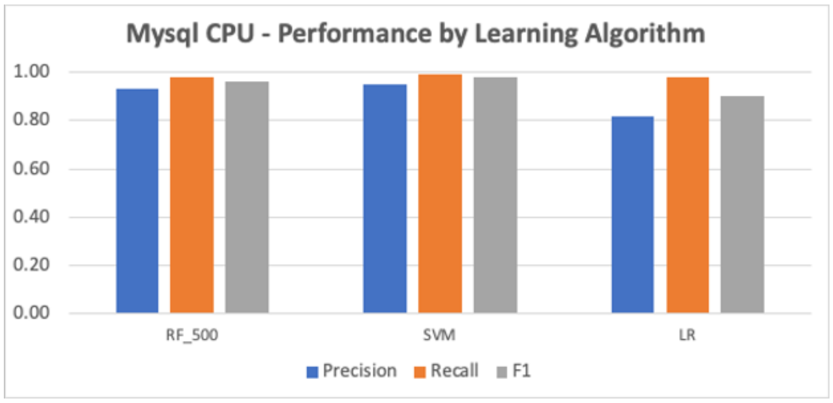


Figure 45 – PerfML Precision, Recall and F1 Performance on Test Set for Mysql CPU Millibottleneck Diagnosis for different choices of base learners.

Figure 43, Figure 44, Figure 45 and Figure 46 shows PerfML's average test set performance for each of the described learning algorithm choices for Apache CPU, Tomcat

CPU, MySQL CPU and CJDBC CPU millibottlenecks respectively. As these figures illustrate, the choice of supervised learning algorithm used to train PerfML's constituent learners does not demonstrably impact PerfML's aggregate performance on diagnosing each type of CPU millibottleneck. However, Random Forests and Support Vector Machines perform consistently better than Logistic Regression with Random Forests performing only marginally better overall. It has about a 1% higher average F1 score than the SVM model.

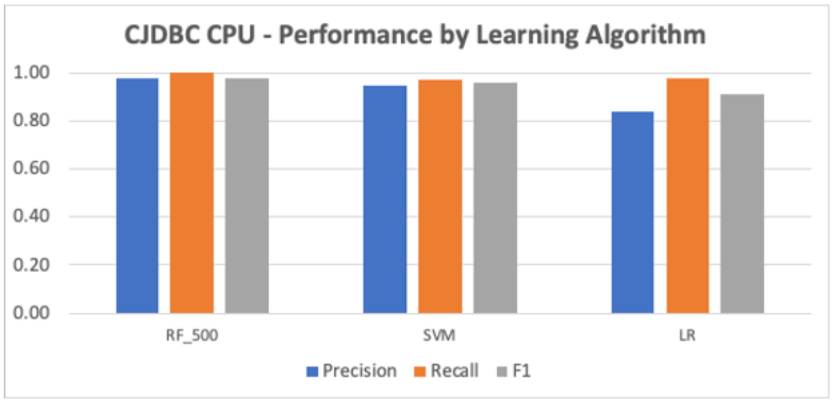


Figure 46 – PerfML Precision, Recall and F1 Peformance on Test Set for CJDBC CPU Millibottleneck Diagnosis for different choices of base learners.

We also evaluated the final probability decision threshold PerfML uses to make its final diagnosis. This amounts to assessing the probability associated with the k^{th} class where *argmax*$_k$ *Pr(y=k)*. For this evaluation, we use the test set from the trial associated with the best performing model from the first evaluation. The X-axis in the Figure 47 is the probability associated with k^{th} class for each prediction. The Y-axis is the cumulative density of test set predictions. The graph illustrates that over 80% of the final diagnostic predictions on this test set had a probability exceeding 0.8. In other words, the maximum probability that corresponds to the k^{th} millibottleneck class exceeded 0.8 in over 80% of the test set predictions. This suggests most of PerfML's predictions have significant support among the data used for training its constituent models. Therefore, changing the default probability threshold from 0.5 would only marginally impact overall classifier performance, i.e. little to not impact on PerfML predictions until the threshold exceeded 0.8.

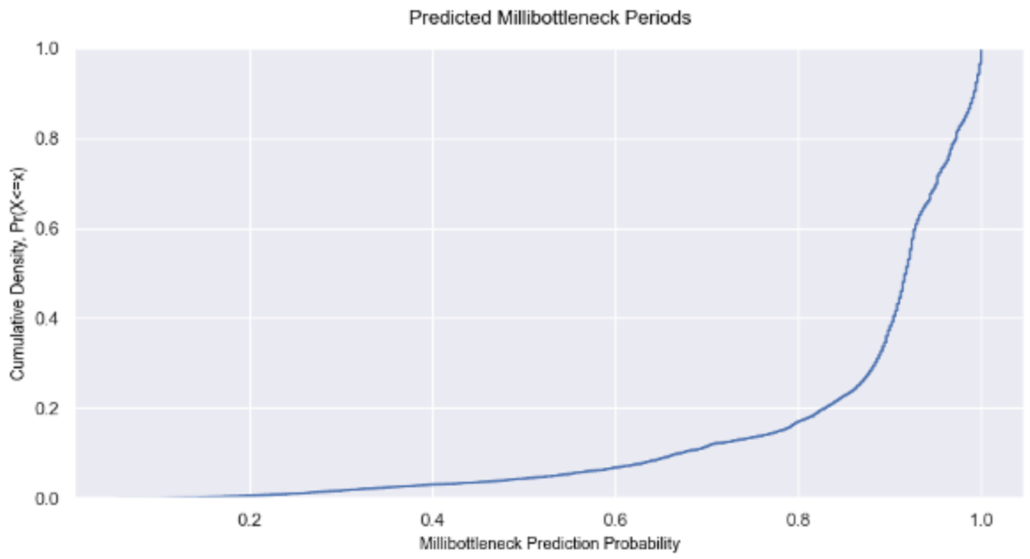


Figure 47 – PerfML prediction probability threshold

5.5.3 PerfML State-of-Art Comparison

In this portion of the evaluation, we compared PerfML's performance to two other system performance anomaly detection systems. The LSTM model is like the one featured in DeepLog [32], and the PCA method is like the one used in a 2009 SOSP paper [33].

Long-Short Term Memory (LSTM) deep learning networks are a type of Recurrent Neural Network (RNN) that retain some "memory" over long sequences by improving upon how weights are modeled which enable them to mitigate the disappearing gradients problem. These models have been applied to many sequence-based domains including Natural Language, Computer Vision and Temporal Forecasting.

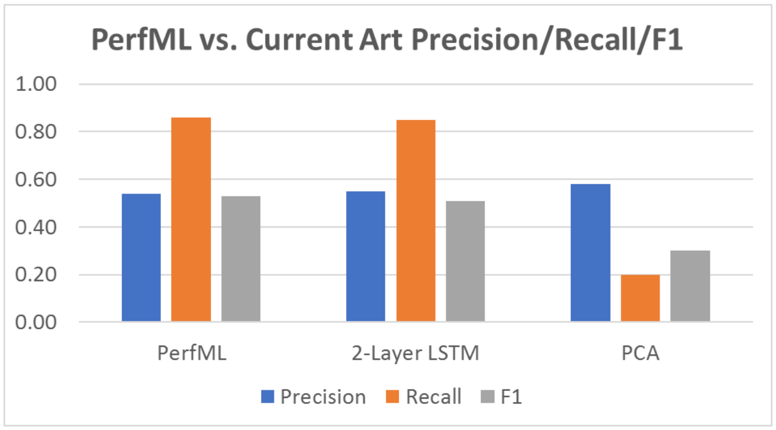


Figure 48 –Precision, Recall and F1 Performance on Test Set of PerfML versus other state-of-art methods.

DeepLog is a two-layer LSTM model. Each layer had 100 LSTM cells. The primary difference between our implementation and the paper's was our input space. The paper used key-value pair sequences extracted from console logs. We fed sequences of resource metrics and queue sizes derived from monitoring logs. We used the same hyperparameters as detailed in the paper: input sequence length of 2 (100ms time windows in our case), a mini-batch size equal to 2 and 50 epochs.

Our DeepLog-like implementation uses 2 layers and 64 memory units per layer just like Deep Log. They used a window size of length 10; however, in our implementation, we used a shorter window size since the performance improved when we used a window of size 2. We used a mini-batch of size 2 and 50 epochs. DeepLog did not specify the number of epochs, and it appears they tried a batch size of 1. We also tried a batch size of 1, but our LSTM did not perform as well.

The other approach by Xu et al. leverages unsupervised dimensionality reduction, Principal Components Analysis, and q-statistic from the statistical process control community to identify systems anomalies from console logs. An observation is deemed abnormal if the squared length of the projection of the vector onto the anomaly subspace is greater than q_a. q_a is the threshold statistic that guarantees the false positive probability is no more than α–assuming the data is multivariate Gaussian. However, this statistic has been shown to be robust even when the data does not conform to this Gaussian assumption. We used the same q-statistic threshold-based approach for identifying millibottlenecks. We used our resource-related features as an analog to their state matrix. We formed a state matrix by stacking resource-related feature vectors from multiple experiments for each type of millibottleneck. We used SVD to determine *k* principal components that accounted for 95% of the variance in the data. Like Xu et al., we found the abnormal subspace by retaining the remaining *n-k* dimensions. Finally, we used the same α threshold of 0.01 and we computed the squared prediction error residual function to compute q_a as specified in Xu et al [33]. Each observation in the test set is projected onto the abnormal space and we used the same decision criteria as in the related paper [33].

We trained all three systems using the entire *Wise-rubbos-beta* data set. Then, we used the entire *Elba-rubbos* set to evaluate the performance of each system.

Figure 48 compares the Precision, Recall and F1 for PerfML, our DeepLog-like 2-layer LSTM implementation and the PCA-based method of Xu et al. We see that PerfML performs better than the PCA-base approach and is comparable to the DeepLog-like 2-Layer LSTM approach. One advantage our method has over the deep learning approach is models are trained over subsapces. Therefore, adding support for new topologies, components and millibottlenecks amounts to train base learners for the respective ensembles. Monolithic models require re-training over the entire sample space.

5.5.4 PerfML Case Study

Our final evaluation involves a demonstration of identifying millibottlenecks in an unlabeled data set, namely “WedMake.” Experiments in this set originated from the same benchmark as the training data, but there were a few differences. First, component queue size measurements originated from TCP instead of HTTP-level monitors like the experiments in the other data sets. As such, feature like component queue length and Point-in-Time Response Time were calculated using finer-grained data. This is only a comparative issue, i.e. trying to compare unscaled queue sizes from these experiments to those in the other data sets. Secondly, experiments in this set had soft resource ranges with higher, upper bounds than in previous experiments. For example, the number of Apache and Mysql sockets ranged from 128 to 512, and the number of connection pool threads was one value from this set: {250, 330, 450, 650, 1050}. Experiments from the other data sets had socket and connection pool thread settings from the lower end of these ranges. Finally, applying our labeling functions would have yielding noisier labels, since these experiments lacked the necessary metadata like each experiment’s JVM version to apply these functions reliably. Given the volume of experiments, hand labeling was infeasible.

As we see in Table 10, PerfML detected millibottlenecks in approximately 7.5% of the observations. We note the most frequently detected millibottleneck was a MySQL CPU millibottleneck with an average duration of 400ms. The other two types were Tomcat and Apache CPU millibottlenecks with similar magnitude average durations. The prevalence of millibottlenecks in this set was slightly more than in either of the other sets. However, the types and average durations were consistent with the millibottlenecks identified in the other data sets. Regardless, we have demonstrated PerfML’s ability to identify millibottleneck periods in data generated in different experimental settings and configuration.

Table 10 – Finding Known Millibottlenecks

Type	Evaluation Data Set Description		
	Number	***Frequency***	***Avg. Duration (ms)***
MySQL CPU	8284	6.5%	400
Tomcat CPU	1425	1.1%	350
Apache CPU	43	0.03%	109

5.6 Future Work and Conclusion

We intend to improve PerfML's coverage by adding support for data generated by microservices and streaming benchmarks. Unlike PerfDB, PerfML requires more adaptation to support new benchmarks. The System State Graph is the primary data structure we use to partition the heterogenous feature space that spans our experimental data. Currently, PerfML's implementation assumes one, static SSG instance. There is not mechanism to mutate this structure programmatically. Moreover, various control flow blocks reference specific literals in this SSG. These references would need to be replaced with calls to an SSG object. PerfML's analytical layer also needs to be extended or refactored. It also contains specific references to literals representing different components in the static SSG. These would also need to be replaced with proper references to an SSG object. Finally, the analytical layer only supports identifying and diagnosing millibottlenecks in monolithic (n-tier), cloud-based systems. The current set of features and models might apply to different execution contexts, but this needs to be studied and evaluated.

Our work on PerfML consisted of three contributions. First, we demonstrated how our approach addresses the heterogenous feature space and hierarchical imbalanced classes inherent to diverse systems' performance data. Secondly, we showed PerfML's high recall and acceptable precision for detecting and diagnosing millibottlenecks. We concluded with a case study using new, i.e. previously unobserved, experimental data to demonstrate PerfML's ability to detect known millibottlenecks and the latency pathologies they induce.

CHAPTER 6. RELATED WORK

Long-tail latency problem. Dean et al. identified long tail latency as a problem common to large scale system topologies. The authors mention variability as root cause, and they argue, “We expect such uncorrelated phenomena [the phenomena that causes variability] are rather common in large-scale systems.” However, they stop short of providing more specific evidence of their root cause [1].

Symptomatic Treatments. Dean et al. prescribed several approaches for bypassing tail latency, specifically employing fault tolerance [1] and resource redundancy techniques [25]. While these strategies are effective at reducing the impact of long tail latency problems, they come with significant cost, for example, lower overall datacenter utilization. According to an NRDC report, from 2008 – 2012, average data center server utilization ranged from 12-18% [26].

Statistical Methods. More recently, Qiu et al provided robust statistical analysis of fine-grained performance anomalies [34]. Instead of reconstructing point-in-time latency or other fine grained latency metrics, the authors use cumulative measures of latency to differentiate the behavior of microservices. Moreover, they use correlation to pinpoint the most likely microservice responsible for a spike in end-to-end latency. They do not show the microservices’ queue lengths in relationship to one other over time. Finally, they monitor a very small set of resources to identify a small number of performance anomalies. Stated differently, they consider a small set of concrete hypotheses to explain specific phenomena. Our population studies go beyond their closed world assumption, since our testable hypothesis includes the closed world of transient resource saturations.

Perhaps, the reason why Dean et al. and others have not sought to diagnose long-tail latency root cause is due to the data management challenges that we have illustrated through our work building PerfML. The data management problems that underlie diagnosing millibottlenecks and latency pathologies like Very Long Response Time Requests and Less Long Requests transcend the system performance domain. As such, there is related work relevant to our data pipeline found across several relevant communities, namely the Systems, Database and Machine Learning communities.

6.1 Systems

In the systems community, there are two relevant fields to performance analysis and debugging.

System Tracing. The first area leverages end-to-end tracing techniques. Google introduced Dapper about 10 years ago, which leveraged sampling to reduce the overhead associated with diagnosing fine-grained performance bugs [35]. X-ray instruments systems' binaries and captures block-level information to estimate the likelihood it executed due to a performance anomaly [36]. Aguilera et al. use statistical methods to correlate observed delays to request traces among systems' modules in distributed systems [37]. Pip and X-trace use end-to-end request flow tracing for performance anomaly diagnosis [38] [39] [40]. Chow et al. rely on a minimal measurement schema injected into software component logs to analyze the end-to-end performance of large-scale Internet services [41]. Sambasivan et. al produced a robust reference of end-to-end request flow tracing techniques [42]. More recently, Canopy uses both performance and execution traces to help isolate coarse-grained performance phenomena [43].

Log Analysis. The second area leverages systems' artifacts like logs to infer relationships between system behavior and performance and reconstructed causal relations. Generally, these approaches learn a model either online or offline from execution logs with performance bugs labeled accordingly. Xu et al. used console logs to identify anomalous executions across the Google infrastructure. They applied statistical control methods combined with Principal Component Analysis (PCA) to detect periods of anomalous system performance [33]. Cohen et al. use Bayesian models to study performance anomalies [44]. Several approaches employ machine learning to detect specific types of performance anomalies from log files [45] [46] [32]. More recently, Gan et al. use as combination of console logs, microbenchmarks, and multi-layered deep neural networks to identify performance anomalies in large, microservice-based topologies [47]. DeepLog mines console logs to construct workflow models and identify anomalous execution patterns using deep neural nets and outlier thresholds based on statistical models [20]. One recent approach does not employ machine learning but instead relies on reconstructing programmers' event logging to profile system behavior [48].

6.2 Database

Work related to data and information extraction from the database community is particularly relevant to *perftables*.

Log Data Extraction. Approaches from previous work in automated information extraction has generally fallen into one of two categories: wrapper induction and supervised learning techniques [49] [50]. Wrapper induction techniques have been applied to web data extraction using structural regularities among HTML tags to separate data from its presentation [50] [51] [52] [53]. For example, groups of specific HTML tags signal more regular sub-structures like tables and lists. In this respect, the layout-based patterns perform a similar function to groups of HTML tags in the web data domain [50]. Work on extracting relations from web lists also shares some parallels with work on information extraction [54]. Work informed by programming language technique feature similar work on log data extraction. Some work has relied on source code interposition techniques to decorate logging statements corresponding to specific strings inserted into the output [55]. Others have synthesized transformations to automatically generate a transformation program from user provided transformation actions. RecordBreaker is one such example of such an approach [56]. Lastly, Datamaran uses parse trees to generate regular expressions, which are used to isolate log data structures [23]. Instead of "learning" structure (bottom-up) like Datamaran, we project structure onto the text (top-down) using a collection of pattern templates. Our templates correspond to frequently occurring layouts, which are similar to the HTML-based inductive styles without the benefit of structure of web-based data.

Supervised techniques have also been applied to log data extraction domain. This previous work usually formulates the machine learning problem as uncovering some unobserved "template" (structure) from the observed text (data) [57] [58]. There are two general limitations with such approaches. First, they are dependent on the availability of data, so in this sense, these methods are biased based on the composition of the corpus [54] [59]. Secondly, using supervised machine learning methods usually means manually curating labels for a training data set, and this data set should contain enough variety to account for potential bias. Unfortunately, to the best of our knowledge no such data set for performance monitoring data is publicly available, and given the enormous parameter space, it is not likely to be sufficiently varied if one did exist.

6.3 Machine Learning

Our model for detecting and diagnosing fine-grained performance anomalies falls under the broad family of ensemble methods. In the machine learning community, work in ensemble methods features work related to multi-class classification and mitigating challenges such as: class imbalance, concept drift and curse of dimensionality including heterogenous data [60].

Ensemble Methods. Ensemble methods can be organized by how their constituent learners are trained and/or how predictions are made [60]. Output manipulation refers to the techniques used by base learners to combine their decisions into one decision for the ensemble. In the literature, this is typically referred to as ensemble fusion, and a special case is ensemble selection, i.e. using the decision of one base learner as the ensemble's overall decision [61]. Input manipulation and partitioning refers to training separate base learners with different training data subsets. Learning algorithm manipulation denotes changing models' hyperparameters to enhance learner diversity. Hybridization refers to the situation when two or more of these other techniques are used to construct an ensemble [60]. We explored relevant work along two of these dimensions: output manipulation and input partitioning.

Output Manipulation. Mixture of Experts is a special type of ensemble method for integrating the outputs of base models into a single output. The approach grew out of the Neural Network community. It was originally posed to decompose the input space such that each expert examines a different portion of the input space. Using error and gating functions, the most suitable base model is determined for each portion of the input space. Each base model is responsible for providing a prediction for its respective space even though some subspaces may overlap [62]. One specific technique involved using clustering to assign experts to particular portions of the input space [63]. Next, we looked at classification systems with multiple feature extractors and multiple classifiers. These approaches relied on either randomization techniques [64] or diversity metrics like mutual information [65] to create vertical and horizontal subspaces to train learners on these specific portions of the input space. The final Mixture of Experts approach that seemed to relate to our classification problem used an *associative switching* mechanism. This special gating function ensures only one expert makes a prediction for the entire ensemble. The switch opens and closes gates attached to each of the experts. For a given *x*, only one gate is opened—that of the most likely prediction. In our case, this mechanism where one expert provides its prediction based on a winner-take-all approach [66] could be used to select a millibottleneck-specific expert.

Input Partitioning. Work in ensembles involving training base learners in specific partitions or portions of the input space seem particularly relevant given the diversity of our millibottlenecks and their corresponding signals. Most ensemble methods refer to some approach for selecting subsets of samples to train members. Generally, input partitioning occurs by either selecting subsets of tuples or rows (horizontal) or subsets of attributes or features (vertical). Some ensemble methods like Random Forests use randomness to select subsets [60]. Some work in the data mining domain [67] use ranges of values to partition a heterogenous feature space. Several of these works are manufacturing domain focused;

however, the problems and methods are relevant to ours [67]. Instead of randomly selecting samples or features, we have domain knowledge that can help to sub-divide the input space. For example, this kind of knowledge could be exploited via vertical partitioning, which could be used to pin specific team members to particular portions of the feature space

One such recent work uses the manufacturing domain as motivation for a set of problems quite similar to ours [68]. Their approach is most similar to ours despite the domain difference. They represent domain knowledge in a tree-based structure, and they use it to partition the data input space. Each partition can have significant class imbalance. While their data exhibits more imbalance, the labels do not have the same hierarchical dependencies ours do. Secondly, we have temporal dependencies we need to respect. Lastly, our data is 1000x times larger, so methods to partition the data set without multiple, full scans of the data set are important.

CHAPTER 7. CONCLUSION AND FUTURE WORK

We began by highlighting an important performance pathology, the long-tail latency problem. We provided our millibottleneck theory to explain at least a portion of the requests that comprise the latency long tail in systems. Next, we motivated our contributions by discussing our measurement toolkit, which provides the data for studying millibottlenecks (detail in Chapter 2). We explained that the data management complexities inherent to this data go beyond studying system performance anomalies themselves. We proceeded by discussing the two significant contributions we made to studying and diagnosing millibottlenecks in experimental computer science big data.

Our first principal contribution, PerfDB, provided the necessary data management and integration infrastructure for us to efficiently integrate data across hundreds of experiments spanning hundreds of experimental schemas (detail in Chapter 4). PerfDB's data pipeline includes a flexible performance log parser, *perftables*, which provides a narrow but important building block for studying millibottlenecks across different experiments (detail in Chapter 3). We showed this parser covered over 98% of the resource monitoring data generated by our measurement toolkit. We evaluated our contribution by using the integrated experimental data to conduct two phenomenological studies. First, we completed a large population study to confirm millibottlenecks cause VLRT vis-à-vis CTQO (queue amplification). Then, we enabled a study of the association between millibottlenecks and Millidelayed requests.

PerfML, our ensemble-based, supervised machine learning system, was our second and more impactful methodological contribution. It handles data characterized by heterogenous feature space and hierarchical, imbalanced classes—characteristics of our experimental data (detail in Chapter 5). We trained PerfML using data collected from hundreds of experiments, and each experiment had its own unique schema. To support training PerfML, we used a combination of manual, semi-automated and automated labeling methods to assemble a high quality, labeled experimental data set. We demonstrated that PerfML could isolate different types of millibottlenecks with high recall and acceptable precision, i.e. over 85% and 80% respectively. We also showed that PerfML performs as well as state-of-the-art methods including deep learning with the added benefit of improved extensibility.

We have begun extending PerfDB and PerfML to support microservice and streaming benchmark applications. In sections 4.5 and 5.6, we detailed how our infrastructure can cover these other benchmark applications despite having different execution models. As such, our infrastructure supports studying the connection between latency phenomena like VLRT and Millidelayed requests and millibottlenecks across a range of systems' semantics and dependencies. Moreover, PerfDB and PerfML provide the data management and analytical infrastructure to construct empirical-based performance profiles for commonly deployed components like nginx or Apache. With this type of information, systems' designers and implementers would be able to pursue a more evidence-based, scientific approach toward their work.

REFERENCES

[1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM,* vol. 56, pp. 74-80, 2013.

[2] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer,* vol. 40, pp. 103-105, 2007.

[3] R. Kohavi, R. M. Henne and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review,* vol. 41, pp. 205-220, 2007.

[5] C. Pu, J. Kimball, C.-A. Lai, T. Zhu, J. Li, J. Park, Q. Wang, D. Jayasinghe, P. Xiong, S. Malkowski and others, "The millibottleneck theory of performance bugs, and its

experimental verification," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[6] Q. Wang, Y. Kanemasa, J. Li, C. A. Lai, M. Matsubara and C. Pu, "Impact of DVFS on n-tier application performance," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[7] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba and C. Pu, "An experimental study of rapidly alternating bottlenecks in n-tier applications," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013.

[8] T. Zhu, J. Li, J. Kimball, J. Park, C.-A. Lai, C. Pu and Q. Wang, "Limitations of load balancing mechanisms for n-tier systems in the presence of millibottlenecks," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[9] J. Park, Q. Wang, J. Li, C.-A. Lai, T. Zhu and C. Pu, "Performance interference of memory thrashing in virtualized cloud environments: A study of consolidated n-tier applications," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.

[10] C. A. Lai, Q. Wang, J. Kimball, J. Li, J. Park and C. Pu, "Io performance interference among consolidated n-tier applications: Sharing is better than isolation for disks," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014.

[11] C.-A. Lai, J. Kimball, T. Zhu, Q. Wang and C. Pu, "milliScope: A fine-grained monitoring framework for performance debugging of n-tier Web services," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[12] D. Jayasinghe, J. Kimball, S. Choudhary, T. Zhu and C. Pu, "An automated approach to create, store, and analyze large-scale experimental data in clouds," in *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, 2013.

[13] D. Jayasinghe, J. Kimball, T. Zhu, S. Choudhary and P. Calton, "An infrastructure for automating large-scale performance studies and data processing," in *2013 IEEE International Conference on Big Data*, 2013.

[14] D. Jayasinghe, S. Malkowski, J. Li, Q. Wang, Z. Wang and C. Pu, "Variations in performance and scalability: An experimental study in iaas clouds using multi-tier workloads," *IEEE Transactions on Services Computing,* vol. 7, no. 2, pp. 293-306, 2013.

[15] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park and C. Pu, "Expertus: A generator approach to automate performance testing in iaas clouds," in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012.

[16] R. A. Lima, J. Kimball, J. E. Ferreira and C. Pu, "Systematic construction, execution, and reproduction of complex performance benchmarks," in *International Conference on Cloud Computing*, 2019.

[17] R. A. Lima, J. Kimball, J. E. Ferreira and C. Pu, "Wise Toolkit: Enabling Microservice-Based System Performance Experiments," in *International Conference on Cloud Computing*, 2020.

[18] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[19] Q. Wang, S. Malkowski, Y. Kanemasa, D. Jayasinghe, P. Xiong, C. Pu, M. Kawaba and L. Harada, "The impact of soft resource allocation on n-tier application scalability," in *2011 IEEE International Parallel \& Distributed Processing Symposium*, 2011.

[20] M. Du, F. Li, G. Zheng and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[21] J. Rivera and R. Meulen, "Gartner says beware of the data lake fallacy," *Gartner http://www. gartner. com/newsroom/id/2809117,* 2014.

[22] B. Stein and A. Morrison, "The enterprise data lake: Better integration and deeper analytics," *PwC Technology Forecast: Rethinking integration,* vol. 1, p. 18, 2014.

[23] Y. Gao, S. Huang and A. G. Parameswaran, "Navigating the Data Lake with DATAMARAN - Automatically Extracting Structure from Log Datasets.," *SIGMOD Conference,* 2018.

[24] Q. Wang, Y. Kanemasa, J. Li, C.-A. Lai, C.-A. Cho, Y. Nomura and C. Pu, "Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance," in *2014 Conference on Timely Results in Operating Systems ({TRIOS} 14)*, 2014.

[25] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and others, "A view of cloud computing," *Communications of the ACM,* vol. 53, pp. 50-58, 2010.

[26] J. Whitney and P. Delforge, "Data center efficiency assessment," *Issue paper on NRDC (The Natural Resource Defense Council),* 2014.

[27] J. Ashley, "Using AI to Reveal Insights in Enterprise Log File Data," 2020.

[28] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar and others, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383,* 2019.

[29] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*, 2013.

[30] J. Li, Q. Wang, C. A. Lai, J. Park, D. Yokoyama and C. Pu, "The impact of software resource allocation on consolidated n-tier applications," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014.

[31] J. Kimball and C. Pu, "A Method and Tool for Automated Induction of Relations from Quantitative Performance Logs," in *International Conference on Cloud Computing*, 2019.

[32] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis and H. Zhang, "Automated IT system failure prediction: A deep learning approach," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016.

[33] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[34] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk and R. K. Iyer, "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Virtual, 2020.

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

[36] M. Attariyan, M. Chow and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012.

[37] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review,* vol. 37, pp. 74-89, 2003.

[38] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems.," in *NSDI*, 2006.

[39] R. Fonseca, M. J. Freedman and G. Porter, "Experiences with Tracing Causality in Networked Services.," *INM/WREN,* vol. 10, 2010.

[40] D. Novaković, N. Vasić, S. Novaković, D. Kostić and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *2013 USENIX Annual Technical Conference ({USENIX ATC} 13)*, 2013.

[41] M. Chow, D. Meisner, J. Flinn, D. Peek and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014.

[42] R. R. Sambasivan, R. Fonseca, I. Shafer and G. R. Ganger, "So, you want to trace your distributed system? Key design insights from years of practical experience," *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14,* 2014.

[43] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi and others, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[44] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly and J. Symons, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.," in *OSDI*, 2004.

[45] K. Nagaraj, C. Killian and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[46] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGARCH Computer Architecture News,* vol. 44, pp. 489-502, 2016.

[47] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[48] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[49] E. Agichtein and L. Gravano, "Snowball: Extracting relations from large plain-text collections," in *Proceedings of the fifth ACM conference on Digital libraries*, 2000.

[50] A. Arasu and H. Garcia-Molina, "Extracting Structured Data from Web Pages.," *SIGMOD Conference,* 2003.

[51] L. Liu, C. Pu and W. Han, "XWRAP: An XML-enabled wrapper construction system for web information sources," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, 2000.

[52] W. Han, D. Buttler and C. Pu, "Wrapping web data into XML," *ACM SIGMOD Record,* vol. 30, pp. 33-38, 2001.

[53] P. Senellart, A. Mittal, D. Muschick, R. Gilleron and M. Tommasi, "Automatic wrapper induction from hidden-web sources with domain knowledge," in *Proceedings of the 10th ACM workshop on Web information and data management*, 2008.

[54] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu and Y. Zhang, "Webtables: exploring the power of tables on the web," *Proceedings of the VLDB Endowment,* vol. 1, pp. 538-549, 2008.

[55] P. He, J. Zhu, Z. Zheng and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *2017 IEEE International Conference on Web Services (ICWS)*.

[56] K. Fisher, D. Walker, K. Q. Zhu and P. White, "From dirt to shovels - fully automatic tool generation from ad hoc data.," *POPL,* 2008.

[57] X. Chu, Y. He, K. Chakrabarti and K. Ganjam, "TEGRA," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, New York, New York, USA, 2015.

[58] E. Cortez, D. Oliveira, A. S. Silva, E. S. Moura and A. H. F. Laender, "Joint unsupervised structure discovery and information extraction.," *SIGMOD Conference,* 2011.

[59] H. Elmeleegy, J. Madhavan and A. Y. Halevy, "Harvesting Relational Tables from Lists on the Web.," *PVLDB,* 2009.

[60] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery,* vol. 8, no. 4, 2018.

[61] L. Rokach, Ensemble learning: Pattern classification using ensemble methods, vol. 85, World Scientific, 2019.

[62] S. E. Yuksel, J. N. Wilson and P. D. Gader, "Twenty years of mixture of experts," *IEEE transactions on neural networks and learning systems,* vol. 23, no. 8, pp. 1177-1193, 2012.

[63] B. Tang, M. I. Heywood and M. Shepherd, "Input partitioning to mixture of experts," in *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN'02)*, 2002.

[64] S. R. Kheradpisheh, F. Sharifizadeh, A. Nowzari-Dalini, M. Ganjtabesh and R. Ebrahimpour, "Mixture of feature specified experts," *Information Fusion,* vol. 20, pp. 242-251, 2014.

[65] K. Kim, H. Lin, J. Y. Choi and K. Choi, "A design framework for hierarchical ensemble of multiple feature extractors and multiple classifiers," *Pattern Recognition,* vol. 52, pp. 1-16, 2016.

[66] K. Chen, L. Wang and H. Chi, "Methods of combining multiple classifiers with different features and their applications to text-independent speaker identification," *International Journal of Pattern Recognition and Artificial Intelligence,* vol. 11, no. 3, pp. 417-445, 1997.

[67] A. Kusiak, "Decomposition in data mining: An industrial case study," *IEEE transactions on electronics packaging manufacturing,* vol. 23, no. 4, pp. 345-353, 2000.

[68] V. Hirsch, P. Reimann and B. Mitschang, "Exploiting domain knowledge to address multi-class imbalance and a heterogeneous feature space in classification tasks for manufacturing data," *Proceedings of the VLDB Endowment,* vol. 13, no. 12, pp. 3258-3271, 2020.