Statistical Analysis of Latency Through Semantic Profiling

Jiamin Huang
University of Michigan, Ann Arbor
jiamin@umich.edu

Barzan Mozafari
University of Michigan, Ann Arbor
mozafari@umich.edu

Thomas F. Wenisch
University of Michigan, Ann Arbor
twenisch@umich.edu

Abstract

Most software profiling tools quantify *average* performance and rely on a program's control flow graph to organize and report results. However, in interactive server applications, performance *predictability* is often an equally important measure. Moreover, the end user is often concerned with the performance of a *semantically defined interval* of execution, such as a request or transaction, which may not directly map to any single function in the call graph, especially in high-performance applications that use asynchrony or event-based programming. It is difficult to distinguish functionality that lies on the critical path of a semantic interval from other activity (e.g., periodic logging or side operations) that may nevertheless appear prominent in a conventional profile. Existing profilers lack the ability to (i) aggregate results for a semantic interval and (ii) attribute its performance variance to individual functions.

We propose a profiler called VProfiler that, given the source code of a software system and programmer annotations indicating the start and end of semantic intervals of interest, is able to identify the dominant sources of latency variance in a semantic context. Using a novel abstraction, called a *variance tree*, VProfiler analyzes the thread interleaving and deconstructs overall latency variance into variances and covariances of the execution time of individual functions. It then aggregates latency variance along a backwards path of dependence relationships among threads from the end of an interval to its start. We evaluate VProfiler's effectiveness on three popular open-source projects (MySQL, Postgres, and Apache Web Server). By identifying a few culprit functions in these complex code bases, VProfiler allows us to eliminate 27%–82% of the overall latency variance of these systems with a modest programming effort.

CCS Concepts **•Software and its engineering → Software testing and debugging**

EuroSys '17, April 23-26, 2017, Belgrade, Serbia
© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064179

Keywords Performance, Predictability, Variance, Tail Latencies, Semantic Profiling

1. Introduction

Profiling tools are a key means of gaining performance insight into complex software systems. Existing profilers provide performance statistics about an application's function call hierarchy. For example, they can answer questions such as which functions are called most often from a particular context, or how much time is spent inside each function. The answers, however, are almost always expressed in terms of *average* performance [19, 34, 37, 64, 65]. Even when multiple runs are used to infer a latency histogram [68], users can still only find out which functions contribute the most to the *overall* latency in each execution time range. However, an increasing number of modern applications come with performance requirements that are hard to analyze with existing profilers. In particular, delivering predictable performance is becoming an increasingly important criterion for *real-time or interactive applications* [1, 3–7, 11, 69], where the latency of individual requests is either mission-critical or affecting user experience.[1]

Existing profilers can only study such systems in terms of their *average performance* breakdown, for example, by attributing average latency to contributions of individual functions. Such profilers offer little help in quantifying the contribution of individual functions to the overall latency variance—a problem that is much more challenging.

The second problem is that performance predictability might only matter in terms of a *semantic interval* that encapsulates the end-user's experience or interaction with the system. For example, a background I/O operation exhibiting large variance in execution time may not matter as long as the graphical user interface does not freeze and users can continue interacting with the system. Similarly, in a database system, performance predictability may only matter insofar as it concerns transaction latencies, and thus latency variance of other functionalities (e.g., periodic log flushing to a separate disk) is irrelevant to the performance profile as long as they do not affect transaction latencies. Unfortunately, se-

[1] In this paper, we do not target those real-time applications that require hard (rather than statistical) guarantees, e.g., airplane control systems.

mantic intervals (e.g., the end-to-end latency of a database transaction or a web-server request) may not always correspond to a single thread or a single top-level function. A transaction might start on one thread and be handed off to and completed on a different thread. In event-based software architectures, a user session or transaction might create multiple events on a shared work queue, whereby multiple worker threads process events in a round robin fashion. The notion of a user transaction or session is inherently a *semantic* one, and it cannot be automatically detected by a generic profiler. For example, the processing of the last event associated with a session may not necessarily correspond to the user's perception of a session end, e.g., post-commit cleanups do not affect a user's latency perception.

Although faster storage and increased hardware parallelism have helped to improve mean performance in general, mitigating performance variance has largely remained an open problem. With increasing complexity of modern applications, subtle interactions of difficult-to-analyze code paths often lead to vexing performance anomalies [1, 3–7, 11, 69]. The rising popularity of cloud-services and service-level agreements in mission-critical applications has increased the need for performance predictability. In light of these trends, we believe it is critical and timely to undertake a systematic approach to diagnosing performance variance in a semantic context.

In this work, we propose a profiling framework, called VProfiler, that can solve both problems. Given the source code of an application and a minimal effort in demarcation of semantic intervals and synchronization primitives, VProfiler identifies the dominant sources of latency variance of semantic intervals. VProfiler iteratively instruments the application source code, each time collecting fine-grain performance measurements for a different subset of functions invoked during the semantic interval of interest. By analyzing these measurements across thread interleavings, VProfiler aggregates latency variance along a backwards path of dependence relationships among threads from the end of an interval to its start. Then, using a novel abstraction, called a *variance tree*, VProfiler carefully reasons about the relationship between overall latency variance and the variances and covariances of the execution time of culprit functions, providing insight into the root causes of performance variance.

We evaluate VProfiler's effectiveness by analyzing three popular open-source projects, MySQL, Postgres, and Apache Web Server, identifying major sources of latency variance of transactions in the former two and of web requests in the latter. In addition to their popularity, we have chosen these three systems for several reasons. First, due to their massive, legacy, and poorly-documented codebases, manual inspection of these source codes is a challenging task (e.g., MySQL has 1.5M lines of code and 30K functions). Second, transactional databases and web servers are a key component of many interactive applications.

Contributions — We make the following contributions:

1. We introduce a novel abstraction, called a *variance tree*, to reason about the relationship between overall latency variance and the variances and covariances of the execution time of culprit functions. Using this abstraction, we present VProfiler as the first profiling tool that can efficiently and rigorously decompose the variance of the execution time of a semantic interval by analyzing application source code and identifying the major contributors to its variance (Section 3).
2. We use VProfiler to analyze MySQL, Postgres, and Apache Web Server, and successfully identify a handful of functions in these massive codebases that contribute the most to their latency variance (Section 4).
3. We show that VProfiler's results are insightful and lead to application-specific optimizations that significantly reduce the overall mean, variance, and 99th percentile latencies by 84%, 82%, and 50%, respectively (Section 4.5).

We discuss the scope of our work in Section 2, and introduce VProfiler in Section 3. We evaluate VProfiler's efficiency in Section 4.1. To evaluate its effectiveness, we conduct detailed case studies of variance for transaction latencies in MySQL and Postgres (Sections 4.5 and 4.6), and for web request latencies in Apache Web Server (Section 4.7). We discuss related work and conclude in Sections 5 and 6.

2. Scope

In this section, we briefly discuss the scope of our work.

Defining Predictability — There are many mathematical notions for capturing *performance predictability* in a software system. One could aim at minimizing the latency variance or tail latencies (e.g., 99th percentile). Alternatively, one could focus on bounding these quantities, e.g., ensuring that the 99th percentile remains under a fixed threshold. To obtain a standardized measure of dispersion (or spread) for a distribution, statisticians sometimes calculate the ratio of standard deviation to mean (a.k.a. coefficient of variation).

While there are many choices, in this paper we focus on identifying the sources of latency variance (and thereby standard deviation), but we only consider solutions that reduce the variance but do not increase mean latency (or reduce throughput). For example, simply padding all latencies with a large wait time will trivially reduce variance but will increase mean latency. While certain applications might tolerate an increase in mean latency in exchange for lower variance [15, 32, 38, 52, 54, 69], such tradeoffs may not be acceptable to most latency-sensitive applications. Thus, in this paper, we restrict ourselves to ideal solutions, i.e., those that reduce variance without negatively impacting mean latency or throughput. In fact, as shown in Section 4, not only do our findings reduce variance, but they also *reduce* mean latency and coefficient of variation. As reported in Section 4.8, one of the variance solutions we discovered with the aid of VProfiler has been quickly adopted (and made a default policy) by MySQL's major distributions.

Finally, while we do not directly minimize tail latencies, reducing variance serves as a surrogate for reducing high-percentiles too [71]. For example, our techniques reduce overall variance by 82%, and 99th percentile latency by 50% for the TPC-C benchmark.

Diagnosis, Not Automated Fixing — There are two steps involved in performance debugging. One is identifying the root cause of a performance problem (variance, in our case), and the other is resolving the issue. Like all profilers, VProfiler focuses on the former. While automating the second step is challenging (e.g., it requires knowing the programmer's original intention), the first step is equally important. In fact, to the best of our knowledge, no existing profiler can identify the true sources of performance variance in a systematic fashion, and VProfiler is the first in this regard (see Section 5).

Though resolving the issue ultimately requires manual inspection, the manual effort needed is often proportional to the extent to which the profiler localizes the sources of the problem. As reported in Section 4, VProfiler's findings allow us to examine only a handful of functions (out of tens of thousands) and dramatically reduce latency variance with modest programming efforts across MySQL, Postgres, and Apache Web Server.

Inherent vs. Avoidable Variance — It is important to note that performance variance is sometimes inherent and cannot be avoided. For example, processing a query that performs more work will inherently take longer than one that performs less work.[2] Avoidable sources of variance are those that are not caused by varying amounts of work, but rather due to internal artifacts in the source code, such as scheduling choices, contention, I/O, or other performance pathologies. For example, two transactions requesting similar amounts of work but experiencing different latencies indicate a performance anomaly that might be avoidable. Determining whether an identified source of latency variance is avoidable or not requires the programmer's understanding of what constitutes *inherent work* in a given application. VProfiler simply reports dominant sources of performance variance so that programmers can focus their attention on only a handful of culprit functions.

Software vs. I/O Delays — In distributed and cloud-based applications, variance in network delays can cause variance in user-perceived latencies. In VProfiler, variance in I/Os such as network traffic or disk (synchronous or asynchronous) operations manifests as variance in the functions that receive the result of the I/O operations, providing programmers an indication that I/O variance is the root cause.

3. VProfiler

With the complexity of modern software, there are many possible causes of latency variance, such as I/O operations, locks, thread scheduling, queuing delays, and varying work per request. Although there are a variety of tracing tools that provide some visibility into application internals (e.g., strace to gain visibility into I/O operations, and DTrace [35] to profile performance), these tools do not directly report performance variation or identify outlying behavior. Moreover, most tracing tools aggregate and report results according to the application call hierarchy, which often does not correspond well to user-visible performance metrics, such as request or transaction processing time. Finally, general-purpose tracing tools introduce substantial (sometimes order-of-magnitude) slowdowns, when collecting fine-grain measurements. For example, we report the overhead of DTrace in Section 3.3.4. The overhead of these tools skews application behavior and obscures root causes of latency variance. In this section, we introduce VProfiler, a novel tool for automatically instrumenting a subset of functions in an application's source code to profile execution time variance at fine time scales with minimal overhead (to preserve the behavior of the system under study).

3.1 Semantic Profiling

A key objective of VProfiler is to quantify performance means and variances over *semantic intervals* rather than report results that are tightly coupled to the application's call graph. A semantic interval is a programmer-defined execution interval that corresponds to a repeated application behavior, which the programmer wishes to profile. Our intent is that a semantic interval should correspond to a single request, session, connection, transaction, and so on, thus allowing the programmer to analyze per-request latency and variance. Note that a semantic interval may encompass concurrent execution spanning multiple threads, or include the time a particular request/context was waiting in a queue or was blocked awaiting some resource.

VProfiler comprises an online trace collection phase and an offline analysis phase. The trace collection phase gathers start and end timestamps of semantic intervals, runtime profiles of a specific set of instrumented functions, and dependency relationships among threads and tasks needed to reconstruct a latency breakdown for each semantic interval. Then, VProfiler performs an offline analysis of these traces to characterize each semantic interval and output a variance profile. If the developer determines that this profile provides insufficient detail, VProfiler selects a new set of functions to instrument to refine the variance profile, and the trace collection phase is repeated. We detail this iterative refinement procedure in Section 3.3.4.

[2] In prior work, we have studied the variance of performance caused by external factors (such as changes in the workload environment) and strategies for mitigating them [48, 49].

VProfiler conceptually divides execution on all threads into *segments*. Each segment is conceptually labeled as either executing on behalf of a single semantic interval, or it is unlabelled to indicate background activity unassociated with any particular request or execution that services multiple requests. As the notion of a semantic interval is application-specific, it must be provided to VProfiler by the programmer via manual annotation. The programmer, via a simple API, enters three kinds of annotations, indicating: (1) when a new semantic interval is created (e.g., transaction start), (2) when a semantic interval is complete (e.g., transaction commit), and (3) when a thread begins executing on behalf of a specific semantic interval (e.g., worker thread dequeues and executes an event associated with the semantic interval).

The first two of these annotations are straight-forward: they provide bounds for the semantic interval. The average performance and overall variance reported by VProfiler is the mean and variance of the time difference between these start and end annotations. However, VProfiler does not seek to merely report these overall aggregate metrics. Rather, it sub-divides the latency between these annotations and attributes it to the execution of particular functions or wait times on specific resources/queues. Furthermore, VProfiler does not require that the start and end of an interval lie on the same thread. Rather, VProfiler considers relationships across threads where one thread unblocks execution of another. It follows such dependence edges backwards from the end of the semantic interval to discover the critical path from the end annotation back to the start timestamp. VProfiler relies on instrumentation added to an application's synchronization primitives that potentially block execution (e.g., locks, condition variables, and message/task queues) to log these dependence edges. We expand on this idea more in Section 3.3.2.

The third annotation is designed specifically for task-based concurrent programming models (e.g., Intel's Threaded Building Blocks), where a semantic interval is decoupled from any particular worker thread. Rather, execution on behalf of a transaction or request proceeds as a sequence of (possibly concurrent) tasks that are dequeued from work queues. In such a framework, a program annotation indicates when a worker thread begins processing a task on behalf of a specific semantic interval. The thread is assumed to continue working on behalf of the semantic interval until another explicit annotation indicates execution on behalf of a new interval. In addition, VProfiler instruments functions that enqueue a task, recording a "created-by" relationship, thereby building a directed graph among the tasks. VProfiler uses this graph to provide a breakdown of latency and variance of execution within the semantic interval and distinguish periods where no task is executed and the semantic interval is delayed due to queuing.

Before discussing VProfiler's algorithm for profiling semantic intervals, we first discuss its model for analyzing performance variance of a function invocation, which is the fundamental building block of VProfiler's approach.

3.2 Characterizing Execution Variance

A *segment* is a contiguous time interval on a single execution thread that may be labelled as part of, at most, a single semantic interval. In this section, we discuss the concepts VProfiler employs to quantify variance with respect to a single executing segment. We describe how VProfiler analyzes entire semantic intervals in subsequent sections.

VProfiler analyzes performance variance by comparing the duration of particular function invocations in an executing segment across other invocations of the same function in different semantic intervals (i.e., it analyzes variances of invocations of the same function across different requests). VProfiler uses a novel abstraction, the *variance tree*, to reason about the relationship between latency variance and the call hierarchy rooted at a particular function invocation.

3.2.1 Variance Tree

We can gain insight into why latency variance arises in an application by subdividing and attributing execution time within a segment across the call graph, similar to a conventional execution time profile generated by tools such as gprof [34]. However, rather than identifying functions that represent a large fraction of execution time, we instead calculate the variance and covariance of each component of the call graph across many invocations to identify those functions that contribute the most to performance variability. Two key challenges arise in this approach: (1) managing the hierarchical nature of the call graph and the corresponding hierarchy that arises in the variance of execution times, and (2) ensuring that profiling overhead remains low. We first discuss the former challenge and address the latter in Section 3.3.4.

Each variance tree is rooted in a specific function invoked over the course of an application. We measure latency and its variance across invocations. For example, in an event processing system, a dispatcher function that dequeues events from a task queue and invokes the specific code associated with the task might comprise the root of the variance tree. VProfiler will build a variance tree beginning at the topmost function whose execution is included within the segment.

Figure 1 (left) depicts a sample call graph comprising a function A invoking two children B and C, and it includes the execution time in the body of A. We can label each node in a particular invocation of this call graph with its execution time, yielding the relationship that the execution time of the parent node is the sum of its children, for example:

$$E(A) = E(B) + E(C) + E(body_A) \qquad (1)$$

where E represents the execution time of a function. Figure 1 (right) shows a corresponding visualization of the variances and covariances in a variance tree representation.

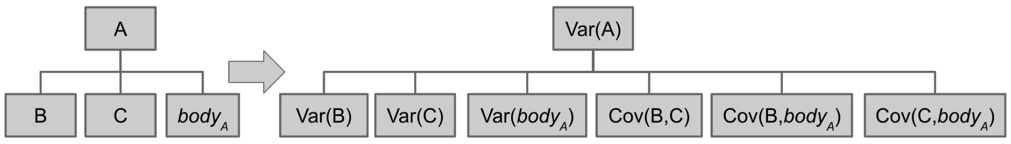

Figure 1: A call graph and its corresponding *variance tree* (here, $body_A$ represents the time spent in the body of A).

$$Var(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} Var(X_i) + 2\sum_{1\le i}\sum_{\le j\le n} Cov(X_i, X_j) \tag{2}$$

The variance tree allows VProfiler to quickly identify sub-trees that do not contribute to latency variability, as their variance is (relative to other nodes) small. Identifying the root causes of large variance, however, is not so trivial. The variance of a parent node is always larger than any of its children, so simply identifying the nodes with the highest variance is not useful for understanding the cause of that variance. Furthermore, some variance arises because invocations may perform more work and manipulate more data (e.g., a transaction accessing more records). Such variance is not an indication of a mitigable pathology as the variance is inherent; our objective is to identify sources of variance that reveal performance anomalies that lead to actionable optimization opportunities. Similarly, high covariance across pairs of functions can be an indicator of a correlation between the amount of work performed by such functions.

At a high level, our goal is to use the variance tree to identify functions (or co-varying function pairs) that (1) account for a substantial fraction of overall latency variance and (2) are informative; that is, functions where analyzing the code will reveal insight as to why variance occurs. To unify terminology, we refer to the variance of a function or co-variance of a function pair as a *factor*.

Identifying factors that account for a large fraction of their parents' variance is straightforward. What is more complicated is identifying functions that are informative. We address this question in the next section.

3.2.2 Ranking Factors

Our intuition is that functions deeper in the call graph implement narrower and more specific functionality, hence, they are more likely to reveal the root cause of latency variance. For example, consider a hypothetical function WriteLog that writes several log records to a global log buffer, but must first acquire the lock on the log buffer (Lock), copy the log data to the log buffer (CopyData), and finally release the lock (Unlock). Suppose WriteLog's variance accounts for 30% of its transaction's latency variance, but CopyData's accounts for 28%. Analyzing CopyData is likely more informative even though it accounts for slightly less variance than WriteLog, because its functionality is more specific. Further investigation may reveal that the variance arises due to the size of log data being copied, suggesting mitigation techniques that reduce log size variance.

```
Inputs : $t$: variance break-down tree,
         $k$: maximum number of functions to select,
         $d$: threshold for minimum contribution
Output: $s^*$: top $k$ most responsible factors
1  $h \leftarrow$ empty list;
2  foreach $node\ \phi \in t$ do
3      $\phi^* \leftarrow$ factor_of($\phi$);
4      if $\phi^* \notin h$ then
5          $\phi^*.contri \leftarrow t.contri$;
6          $h \leftarrow h \cup \phi^*$;
7      else
8          $\phi'.contri \leftarrow \phi'.contri + \phi.contri$;
9  foreach $\phi \in h$ do
10     $\phi.score =$ specificity($\phi$) $\cdot\ \phi.contri$;
11 Sort $h$ in descending order of $\phi.score$;
12 $s^* \leftarrow$ empty list;
13 for $i \leftarrow 1$ to $k$ do
14     $\phi \leftarrow h[i]$;
15     if $\phi.contri \ge d$ then
16         $s^* \leftarrow s^* \cup \phi$;
17 return $s^*$;
```

Algorithm 1: Factor Selection

Based on this intuition, VProfiler ranks factors using a score function that considers both the magnitude of variance attributed to the factor and its relative position in the call graph. A particular factor may appear in a call graph more than once if a function is invoked from multiple call sites. When ranking factors, VProfiler aggregates the variance/covariance across all call sites.

To quantify a factor's position within the call graph, VProfiler assigns each function a height based on the maximum depth of the call tree beneath it. For factors representing the covariance of two functions, VProfiler uses the maximum height of the two functions. It uses a specificity metric that is a decreasing function of the factor's height ϕ:

$$specificity(\phi) = (height(call\_graph) - height(\phi))^2 \tag{3}$$

where $height(call\_graph)$ is the height of the root of the call graph, and $height(\phi)$ is the factor's height. Here we use square to give specificity a higher weight.

VProfiler uses a score function that jointly considers specificity and variance:

$$score(\phi) = specificity(\phi) \sum_i V(\phi_i) \tag{4}$$

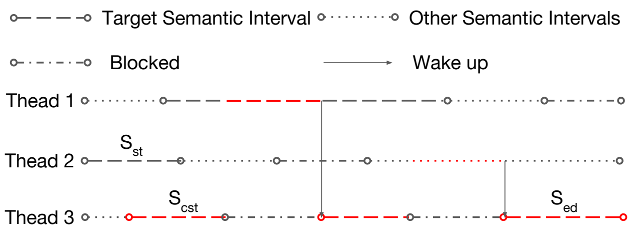

Figure 2: A critical path (marked red) constructed for a semantic interval not involving the start segment (S_{st}).

where $V(\phi_i)$ represents variance or covariance of a specific instance (call site) of a factor within the call graph.

Given the *variance tree*, we now describe an algorithm to select the top-k factors based on their score. The pseudocode is shown in Algorithm 1. For each node in the tree, we determine if the corresponding factor is already in list h. If not, we insert the factor and its (co-)variance into h. Otherwise, we accumulate the (co-)variance represented by the node into the existing element in h (lines 1 to 10). Once we have calculated total (co-)variance of each factor, we calculate their *score* values using Equation 4 (lines 11 to 13). Then, we sort factors in descending *score* order, selecting the top k whose total (co-)variance is greater than a threshold d (lines 14 to 23).

3.3 Profiling a Semantic Interval

We next describe how VProfiler aggregates latency variance of an entire semantic interval from individual execution segments. Furthermore, we describe the iterative refinement method used to recursively add details to the variance tree over a sequence of experimental trials to provide the developer a sufficiently informative latency and variance breakdown of semantic segments.

VProfiler's offline analysis phase characterizes variance in each semantic interval, starting at its final segment (containing the interval completion annotation). VProfiler then aggregates latency and variance of preceding functions on the same thread until it encounters an incoming wake-up edge indicating execution on this segment was triggered by an event elsewhere (e.g., a lock being freed). It then follows this edge, continuing aggregation along the target thread, and so on, following all incoming dependence edges. This backwards traversal terminates when it reaches the creation timestamp of the semantic interval. Note that, in complex executions, the backwards trace may not actually include the segment that created the event, as this segment may not lie on the critical path that determined the end time of the interval (i.e., the end timestamp of the interval may not be improved if its start timestamp were earlier). Figure 2 shows an example of this. Intuitively, we conceive of VProfiler as assigning "blame" for accumulated delay leading to the completion of a semantic interval; blame propagates backwards along segments and their dependencies.

3.3.1 VProfiler Workflow

Given a complete variance tree, factor selection (Algorithm 1) identifies the top factors that a developer should investigate further to identify the root causes of semantic interval variance. However, collecting a complete variance tree is infeasible due to the enormous size and complexity of call graphs in modern software systems. Instrumenting every function adds significant overhead to execution time, and the variance tree will no longer be representative of unprofiled execution. Hence, VProfiler iteratively refines the instrumentation to build variance trees, starting from their roots until the profile is sufficient for a developer to identify key sources of variance.

In addition to constructing one or more variance trees (rooted at specific functions in each run), to aggregate results over the course of a semantic interval, VProfiler must trace the following data in each run:

1. $\langle tid, sid, ts, te, state\rangle$. Each such 5-tuple describes a segment. tid is the id of the thread on which a segment is executed. sid is a unique identifier of the semantic interval assigned when the interval is created (e.g., a transaction id). ts and te are the starting and ending segment timestamps and $state$ indicates whether the segment was executing, waiting on a task or message queue, or blocked on a lock or I/O.
2. $\langle tid, sid, f, fs, fe\rangle$. Each such 5-tuple describes a function invocation that was selected for instrumentation during this run. f is a function name. fs and fe are the start and end timestamps of an invocation of f.
3. $\langle tid, tid', t\rangle$. Each such 3-tuple indicates that thread tid was woken up by thread tid' at time t.
4. $\langle tid, ts, tid', ts'\rangle$. Each such 4-tuple represents the event creation relationship. Thread tid created an event at time ts, and that event was picked up by thread tid' at time ts'.

3.3.2 Tracking Segment Dependencies

VProfiler must track segment dependencies at run-time to construct the 3- and 4-tuples indicating when a thread wakes or creates another thread. For this tracking, VProfiler requires instrumentation of synchronization operations and operations that enqueue tasks in task-based runtimes.

We abstract generic blocking synchronization primitives as having an $\mathrm{acquire(object)}$ and $\mathrm{release(object)}$ function.[3] This pattern covers a number of primitives, including locks, mutexes, condition variables, and semaphores. VProfiler instruments the acquires and releases and tracks lock ownership at run-time using a large hash map of $[\mathrm{oid} \rightarrow \mathrm{tid}]$, where oid is an identifier of the synchronization object

[3] Note that the developer must supply a comprehensive list of acquire and release function names to VProfiler to instrument.

(e.g., the lock address) and tid is the ID of the last thread that holds the object.

To track task relationships in task-based applications, VProfiler instruments the enqueue and dequeue operation on the task queues. We assume a model where the consumer threads pop objects off the queue, and block when the queue is empty. The producer threads push objects, potentially waking a worker thread to accept the task. We assume an abstract API comprising enqueue(queue, task) and dequeue(queue) functions. Here, VProfiler also maintains a hash table [task $\rightarrow$ tid] at run-time, where task is a unique task identifier and tid is the ID of its producer thread.

VProfiler does not instrument the OS scheduler. Hence, if a runnable thread is pre-empted due to CPU over-subscription, VProfiler will include the time the thread is runnable but waiting as part of the execution time of the segment. This limitation of VProfiler can be overcome by instrumenting the OS to log thread switches due to time slice expiration or pre-emption. In the workloads we study, pre-emption is rare as the number of concurrent application threads is tuned not to exceed the number of available cores, so there is no need to instrument the scheduler.

3.3.3 Aggregating Segments

Algorithm 2 shows VProfiler's pseudocode for post-processing the variance trees and segment relationship output for an individual semantic interval. This is a recursive function, and the initial call should pass in the id of semantic interval of interest, its end segment, nil as its start timestamp, and the end timestamp of the end segment as arguments. In Algorithm 2, a segment S is described by its 5-tuple: $\langle T, C, ts, te, s\rangle$, where T is the thread on which S was executed, C is a unique identifier of interval S, ts and te are the start and end timestamps and s is the state of the segment.

The high-level idea of Algorithm 2 is to construct the critical path for the given semantic interval and to analyze the execution time of the target function in each segment on the critical path. Figure 2 shows an example of how a critical path is constructed. The algorithm starts from the ending segment and follows any wake-up/created-by relationship backwards. Note that when the algorithm follows a wake-up relationship backwards, it only processes the waker segment up to the point where the blocked segment starts, and then returns to the thread of the blocked segment.

Algorithm 2 maintains a table of execution times comprising $\langle cid, f, et\rangle$ tuples where cid is the semantic interval id, f is the name of a profiled function (or other for time spent in uninstrumented functions), and et is the total execution time of the function over the course of the semantic interval (as functions may be invoked more than once). Thus, there is a single row for each (semantic interval, function) pair. We process one semantic interval at a time, aggregating time spent in each function while walking backwards along the segments included in the interval, starting at the final segment. The key step in the algorithm is monitor_exec_time. For a given semantic interval C, a thread T, and a start and end timestamp ts and te, this step finds all $\langle tid, cid, f, fs, fe\rangle$ tuples that match the semantic interval id and thread id and also overlap the segment's duration. The execution of each function overlapping the segment is then clipped against the bounds of the segment and aggregated into the execution time table. The table is then output when all semantic intervals have been processed.

```
Inputs : $sid$, the target semantic interval id,
         $S$, the segment to be aggregated,
         $B$, a hard beginning timestamp to start aggregation (could be $nil$),
         $E$, a ending timestamp to stop aggregation
function aggregate_segment ($sid, S, B, E$)
    if execute_for_target_semantic_interval($S, sid$) then
        /* max will handle nil value correctly */
        monitor_exec_time($S.C, S.T, \max(B, S.ts), E$);
    else if is_blocked($S$) then
        $S' \leftarrow$ find_waking_segment($S.T, S.te$);
        aggregate_segment($S'.C, S'.T, \max(B, S.ts), E$);
    if $B = nil \lor S$.begin() $> B$ then
        $P \leftarrow$ get_previous_segment($S$);
        while $P \neq nil$ do
            if $B \neq nil \land P.te \leq B$ then
                return;
            if $P.sid = sid$ then
                accumulate_wait_time($P.te, S.ts$);
                break;
            $P \leftarrow$ get_previous_segment($P$);
        if $P \neq nil$ then
            aggregate_segment($sid, P, B, P.te$);
        else
            $P \leftarrow$ find_generator($S$);
            if $P \neq nil$ then
                $T \leftarrow$ find_generating_time($S$);
                accumulate_wait_time($T, S.ts$);
                aggregate_segment($sid, P, B, T$);
```

Algorithm 2: Aggregate Segments

3.3.4 Iterative Refinement

In each iteration, VProfiler identifies the top-k factors when profiling a subset of functions, starting at the root of the call graph. This profile is then returned to the developer, who determines if the profile is sufficient. If not, the children of the top-k factors are added to the list of functions to be profiled, instrumentation code is automatically inserted by VProfiler, and a new profile is collected. In detail:

Initialization (Algorithm 3, line 1 to 3). VProfiler starts with an empty variance tree, and initializes the list of functions to profile with the root.

Variance Break Down (Algorithm 3, line 5 to 8). For each profiled function, VProfiler automatically instruments the code to measure the latency of all invocations of the function and the latency of each child. The variance and co-variances of these children are added to the variance tree, thereby expanding the tree by one level.

Factor Selection (Algorithm 3, line 9 to 17). After the variance tree is expanded, VProfiler performs factor selection to choose the top-k highest scoring factors within the tree, which are then reported. If the profile is insufficient, the developer requests another iteration, which adds the children of these top-k functions to the list to be profiled.

Note that, ultimately, VProfiler's output is heuristic. It identifies code that contributes to variance, but a developer must analyze this code to determine if the variance is inherent or is indicative of a performance pathology.

VProfiler uses a parser to automatically inject instrumentation code as a prolog and epilog to each function that is selected for profiling, using a source-to-source translation tool and then recompiling the binary. Our approach is similar to conventional profilers, such as gprof, except that VProfiler instruments only a subset of functions at a time.

```
Inputs :  $v$: the starting function (i.e., entry point),
          $k$: maximum number of functions to select,
          $d$: threshold for minimum contribution
Output:   $s^*$: top $k$ most responsible factors
1  $t \leftarrow$ tree with $Var(v)$ as root;
2  $l \leftarrow \{Var(v)\}$;
3  $e \leftarrow true$;
4  while $e$ do
5      foreach $factor\ f \in l$ do
6          if is_variance($f$) then
7              $c \leftarrow$ var_break_down($f$);
8              $t$.add_children($f$, $c$);
9      $s^* \leftarrow$ select_factors($t$, $k$, $d$);
10     $l$.clear();
11     $e \leftarrow false$;
12     foreach $factor\ f \in s^*$ do
13         if needs_break_down($f$) then
14             $l \leftarrow l \cup f$;
15             $e \leftarrow true$;
16         else if is_variance($f$) then
17             mark_as_selected($f$);
18 return $s^*$;
```

Algorithm 3: Iterative refinement.

3.4 Implementation

Our current implementation of VProfiler only supports C/C++ applications.[4] VProfiler takes in a list of synchronization primitives used in the application, creates instrumented wrappers for them, and replaces all synchronization function calls with the wrapper calls in order to construct the wake-up graph of the threads and created-by relationships between tasks. The programmer also uses an API provided by VProfiler to mark the creation and completion of a semantic interval, and the places where a thread starts working on behalf of a semantic interval. In addition, the programmer provides a script that copies the instrumented source code files to the source repository, compiles the source code, and runs the application. Given these inputs, VProfiler automatically instruments the appropriate functions, runs the application, and returns k factors with the highest score (k=3 by default). For each selected factor, VProfiler asks the programmer whether to investigate it further or not. (In our experience, in many cases, this decision is usually straightforward and does not require a deep understanding of the source code.) If the programmer deems it necessary, VProfiler re-instruments the selected function(s) and reruns the application to collect new measurements.

[4] We also plan to add support for Java applications in our next release.

4. Evaluation

In this section, we aim to evaluate the efficiency and effectiveness of VProfiler. Performance overhead can be directly measured (Section 4.1). However, the ultimate measure of VProfiler's effectiveness is how successful it is in leading programmers to modifications that improve performance predictability. Directly quantifying this success is subjective or at least difficult. Instead, we apply VProfiler to a few complex, real-life software systems (e.g., MySQL), and make modifications to these systems based on VProfiler's findings. The questions then become (1) if and how much these modifications succeed in reducing latency variance in these systems, and (2) how much programming effort (e.g., lines of code or hours) is involved in implementing these modifications (to measure how local and specific VProfiler's findings are). We answer the first question in Sections 4.5–4.7, and the second in Section 4.2.

Specifically, we conduct case studies on three popular open-source systems: MySQL (a thread-per-connection database), Postgres (a process-per-connection database), and Apache Web Server (an event-based server application). For MySQL and Postgres, we treat each 'transaction' as a semantic interval, while for Apache we treat each 'web request' as a semantic interval.

Finally, as a measure of practicality of our findings, in Section 4.8 we report on the real-world adoption of some of the optimizations discovered using VProfiler.

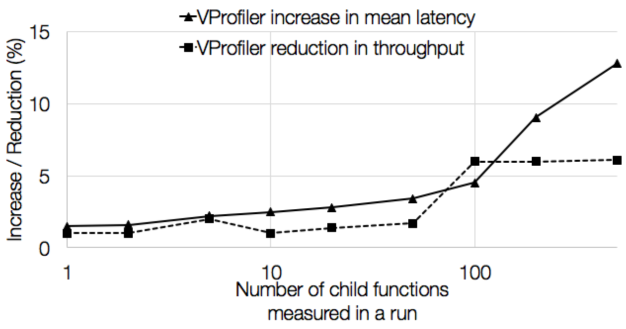

Figure 3: Profiling overhead of VProfiler.

In summary, our experiments indicate the following:

1. VProfiler's profiling overhead is an order of magnitude lower than DTrace, and its factor selection algorithm reduces the number of required runs by several orders of magnitude compared to a naïve drill-down strategy.
2. VProfiler successfully reveals the actual sources of variance in these large and complex codebases.
3. VProfiler's findings enable us to dramatically reduce latency variance and 99th percentile latencies of each system with small modifications of their source code. Interestingly, our findings even reduce mean latencies. See Table 1 for a summary.

4.1 Instrumentation Overhead

We first report the overhead introduced by VProfiler's online instrumentation. In our case studies, we almost never needed to profile a function with more than 100 children. Nonetheless, as shown in Figure 3, we studied the overhead of VProfiler for MySQL as we varied the number of children under an instrumented function from 1 to 500 running the TPC-C workload [53]. As one would expect, the overhead tends to grow as the number of children grows (since we need to measure the execution time of more functions). However, in all cases, overheads are below 14% in terms of both latency and throughput.

VProfiler overhead compares favorably to tracing tools that perform binary code injection. For example, we used DTrace [36] (a popular binary trace injection tool) to insert similar instrumentation. Although DTrace does not natively support any notion of semantic intervals, one can use it to measure the variance of a fixed set of functions and then compute variances using Equation 2. DTrace incurred 10-20x higher overheads than our source-level instrumentation (details omitted for space), and scales worse when tracing more children. VProfiler gains an advantage over DTrace because its instrumentation is minimal and inserted into the source, rather than via binary modification.

4.2 Manual Effort

The manual effort in using VProfiler includes (i) annotating the semantic intervals, (ii) inspecting the variance profile returned at each iteration, and (iii) making enhancements to address pathologies VProfiler identifies. Quantifying these efforts objectively is difficult, as they depend heavily on the programmer's familiarity with the codebase. However, here we simply report the experiences of one of the co-authors (who had no prior experience with these codebases) performing the case studies. As reported in Table 2, the annotation of semantic intervals requires only a few lines of code. We expect this to hold in most cases, as the notion of a semantic interval is typically intuitive to developers (e.g., a request or a transaction). Identifying the synchronization functions is similarly straight-forward, as codebases typically use a well-defined API for synchronization. Finally, the actual optimization modifications were quite small, due to the specificity of the functions identified by VProfiler.

4.3 Variance Trees

Table 3 reports some statistics of the final variance tree for each application. Compared to Postgres and Apache Web Server, studying latency variance in MySQL required more runs but also less time inspecting the returned profile at each run. This is because MySQL's source code is more hierarchical with many functions simply delegating the work to others. For the same reason, the height of the final variance tree of MySQL is larger than the other two. The breadth of the variance tree is mainly affected by the function that has the largest number of children. Note that with factor selection, VProfiler always looks only at k selected factors, and, therefore, most of the nodes in the variance tree are simply (yet safely) ignored.

4.4 The Choice of the Specificity Function

As discussed in Section 3.2.2, VProfiler uses a quadratic function in its factor selection phase to quantify the specificity of a factor. We experimented with several specificity formulations, including linear, quadratic, and cubic functions. We then compared the quality of their findings. The linear function assigned insufficient weight to the height of a factor, causing an important factor with 18.2% contribution to the overall variance to be missed in an early iteration. On the other hand, the cubic function ultimately yielded exactly the same factors as the quadratic function, providing no additional benefit. Consequently, we have chosen the quadratic function defined equation (3) as our default choice of the specificity function in VProfiler.

4.5 Case Study: MySQL

We first use VProfiler to analyze the source code of MySQL 5.6.23. Our study of MySQL used the TPC-C benchmark [53] in two different scenarios. The first scenario, with 128 warehouses and a 30GB buffer pool (denoted as the 128-WH configuration), modeled a deployment where memory was large enough to fit all data, while the second, with 2 warehouses and a 128MB buffer pool (denoted as the 2-WH configuration), modeled a scenario where contention on

Application	The function causing variance	Original contribution to overall variance	Proposed technique	# of modified lines of code or config	Reduction of overall mean latency	Reduction of overall latency variance	Reduction of overall 99^{th} latency
MySQL	os_event_wait	59.2%	grant the lock to the oldest trx first	189	84.0%	82.1%	50.0%
MySQL	buf_pool_mutex_enter	32.92%	replace mutex with spin lock	46	10.7%	35.5%	26.5%
MySQL	fil_flush	5%	parameter tuning	2	18.7%	27.0%	14.5%
Postgres	LWLockAcquireOrWait	76.8%	distributed logging	355	58.5%	44.8%	23.7%
Apache Web Server	apr_bucket_alloc	11%	bulk memory allocation	45	4.8%	60.0%	42.9%

Table 1: Impact of modifying each of the functions identified by VProfiler. The last three columns compare the *overall* performance of the whole system before and after modifying each function.

Application	Semantic intervals annotations	Avg. manual inspection time per run	Modified lines of code
MySQL	9 lines of code	6 minutes	235
Postgres	7 lines of code	10 minutes	355
Apache	4 lines of code	12 minutes	45

Table 2: Our manual effort while using VProfiler.

Application	Number of VProfiler runs	Variance tree height	Variance tree breadth
MySQL	37	19	245025
Postgres	16	8	16900
Apache	17	15	36

Table 3: Statistics of the final variance trees.

Config	Source of Variance	Contribution to Overall Variance
128-WH	os_event_wait [A]	**37.5%**
128-WH	os_event_wait [B]	**21.7%**
128-WH	row_ins_clust_index_entry_low	**9.3%**
2-WH	buf_pool_mutex_enter	**32.92%**
2-WH	img_btr_cur_search_to_nth_level	**8.3%**
2-WH	fil_flush	**5%**

Table 4: Key sources of variance in MySQL.

Mean Latency	Variance	99th Percentile
84.0%	82.1%	50.0%

Table 5: Comparing VATS with MySQL's original (FCFS) lock scheduling in terms of MySQL's overall latency.

the buffer pool was intense due to severe memory constraints. In both scenarios, MySQL ran on a server with 2 Intel Xeon E5-1670v2 2.1GHz virtual CPUs, receiving transactions sent from a separate client machine running the OLTP-Bench [29] tool.

Table 4 summarizes the key variance sources in MySQL identified by VProfiler. Whereas MySQL has a complex code base with over 1.5M lines of code and 30K functions, VProfiler narrows down our search by automatically identifying a handful of functions that contribute the most to the overall transaction variance. These findings clearly demonstrate the value of VProfiler: we only need to manually inspect these few functions to understand whether their execution time variance is inherent or is caused by a performance pathology that can be mitigated or avoided.

Next, we present our findings from inspecting the source of the functions identified by VProfiler.

• **os_event_wait.** This function is simply a wrapper for the pthread_cond_wait function in Linux (or equivalent functions on other platforms). One of its uses is to put a transaction thread to sleep when it needs to wait for a lock on a data record. The A and B in Table 4 refer to the two most important call sites for os_event_wait, which correspond to locks acquired during select and update statements, respectively. The high variance of these call sites reflects the high variability in the wait time for contended locks, which contributes to more than half of the overall latency variance.

Our fix is to replace MySQL's lock scheduling strategy, which is based on First Come First Served (FCFS), with a Variance-Aware-Transaction-Scheduling (VATS) [40] strategy that minimizes overall wait time variance. In short, VATS uses an Oldest Transaction First (grant the lock to the oldest transaction) strategy as an alternative to FCFS. As shown in Table 5, VATS reduces 82.1% of the overall latency variance and 50.0% of the 99th percentile latency (for the TPC-C workload).

• **row_ins_clust_index_entry_low.** This function inserts a new record into a clustered index. Its variance arises due to varying code paths taken based on the state of the index prior to the insert operation. The variance here is inherent to the index mutation, hence, we concluded that it is not a performance pathology.

• **buf_pool_mutex_enter.** This function is used to acquire a global lock on the buffer pool. Of all the call sites, the one that contributes most to the overall variance is one that happens in a function used to move a page to the head of the buffer page list in order to maintain a Least Recently Used (LRU) order in the pages. This causes high variance in the wait time for acquiring the global lock on the buffer page list.

We evaluated a potential fix, Lazy LRU Update (LLU) [40], which limits the time that buf_pool_mutex_enter waits for the lock to avoid excessive delays. The page moving operation is aborted if the lock is not granted in time and will be retried the next time the lock is successfully acquired.

We replaced the original mutex with this LLU strategy using only 46 lines of code. Figure 4 *(left)* shows that LLU eliminates 10.7% of the overall mean latency, 35.5% of the overall variance, and 26.5% of the overall 99th percentile latency compared to the original mutex-based implementation. LLU avoids long waits by delaying moving the buffer pages until the overhead is fairly cheap. This reduces the contention on the LRU data structure for memory-contended workloads.

• **btr_cur_search_to_nth_level.** This function traverses an index tree, placing a tree cursor at a given level, and leaving a shared or exclusive lock on the cursor page. A performance-critical loop in this function traverses from level to level in the index tree, and its runtime varies with the depth to which the tree must be traversed. From our inspection, we again conclude that the variance here is inherent to the index traversal, not a performance pathology.

• **fil_flush.** MySQL uses this function to force all redo log data to be written from the disk buffer to the disk before a transaction can be safely committed. The variance here is inherent to the I/O, but might be reduced by logging to faster I/O devices, such as SSDs or NVRAM [14, 51, 72].

Alternatively, by changing MySQL's log flush policy through the innodb_flush_log_at_trx_commit configuration parameter, we can defer log flushing off the critical path. By default, MySQL flushes redo logs eagerly. Figure 4(center) shows the results of using alternative *lazy flush* and *lazy write* settings, which defer only flush, or both flush and write system calls, respectively, to a log flusher thread. Note that both lazy policies risk losing committed transactions if a crash occurs (though the database will remain consistent, i.e., there will be no changes made by partial transactions). Nevertheless, this finding is an example in which VProfiler helped us to identify a configuration setting that has a large influence on transaction variance.

4.6 Case Study: Postgres

We next perform a case study using VProfiler to analyze the source code of Postgres 9.6, another complex and popular DBMS. For Postgres, we use a server with 2 Intel(R) Xeon(R) CPU E5-2450 processors and 2.10GHz cores, and use a separate client machine. We study the TPC-C benchmark with a 32-warehouse configuration and a 30 GB buffer pool. Table 6 shows the top three functions in Postgres identified by VProfiler as the main sources of variance.

Source of Variance	Contribution to Overall Variance
LWLockAcquireOrWait	**76.8%**
ReleasePredicateLocks	**6%**
ExecProcNode	**5%**

Table 6: Key sources of variance in Postgres.

• **LWLockAcquireOrWait.** This function is used to acquire an exclusive lock. Again, one specific call site contributes most of the variance. In Postgres, an exclusive lock is used to protect the redo log to ensure that only one transaction at a time can write to the log. Therefore, a natural idea is to either reduce contention on this global lock, or to allow for multiple transactions to flush simultaneously. The former may be attempted by accelerating I/O (e.g., tuning the I/O block size in Postgres, or by placing the logs on a NVRAM [14, 72] or SSD [24, 57]), whereas the latter can be attempted by a variety of distributed logging schemes (e.g., [25, 74]).

Here, we tested a simple distributed logging variant that allows Postgres to use two hard disks for storing two sets of redo logs. A transaction only needs to wait when neither of these two sets is available, in which case it waits for the one with fewer waiters. The result is shown in Figure 4 *(right)*, showing that this simple technique eliminates 58.5% of the overall mean latency, 44.8% of the overall latency variance, and 23.7% of the overall 99th percentile latency.

• **ReleasePredicateLocks.** This function releases all the *predicate locks* held by a transaction when that transaction commits (or rolls back). In the "serializable" isolation level, Postgres uses predicate locking to avoid the "phantom" problem where a read conflicts with later inserts or updates which add new rows to the selected range of the read. Varying lock conflicts can be discovered upon the release of these locks, and the execution time varies with the number and type of conflicts. However, according to VProfiler, this function is only responsible for 6% of the overall variance, so we did not pursue it further.

• **ExecProcNode.** After parsing a query, Postgres generates an execution plan for it. This plan is a tree-like structure with multiple plan nodes (e.g., scans, joins). Depending on the node type, this function dispatches to others to perform the required work. The variance of ExecProcNode, therefore, stems from differences in query plans. No single child of this function accounts for a significant fraction of variance, so we did not pursue it further.

4.7 Case Study: Apache Web Server

Finally, we used VProfiler to analyze the source code of Apache Web server 2.4.23. We used the included benchmark, ApacheBench. We simulated 1000 clients, sending a total of 20000 requests for a 169-byte static page. Table 7 reports the top root causes of variance in the request latencies. Here, the top two causes found by VProfiler were

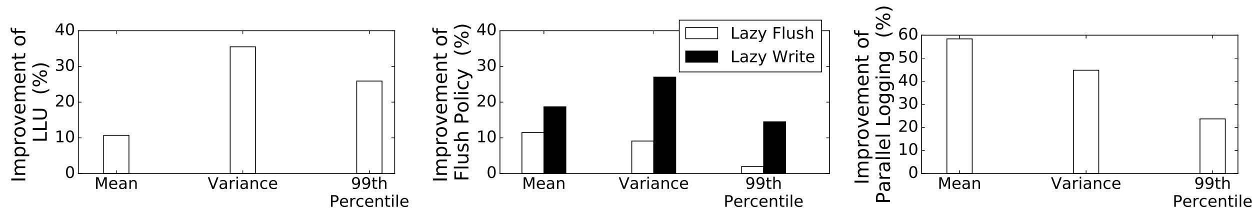

Figure 4: Effect of LLU algorithm *(left)* and varying log flush policy *(center)* on MySQL's performance (TPC-C benchmark). Effect of distributed logging *(right)* on Postgres (TPC-C).

Source of Variance	Contribution to Overall Variance
(ap_pass_brigade, apr_file_open)	**22%**
(ap_pass_brigade, basic_http_header)	**15.5%**
apr_bucket_alloc	**11.8%**

Table 7: Key sources of variance in Apache HTTPD Server.

co-variances of two functions (shown as pairs in Table 7), rather than the variance of a single function. Co-variances indicate, for example, that whenever ap_pass_brigade takes long, apr_file_open does too, and vice versa.

• **(ap_pass_brigade, apr_file_open)** The first of this function pair is a recursive function invoking a filter chain on incoming requests. The second is a wrapper function for opening files across different operating systems. When the allocator lacks sufficient memory, both functions incur delay waiting for memory allocations. Similarly, when available memory is sufficient, both become faster. Hence, though not directly related, the two functions co-vary due to the same underlying root cause. VProfiler was instrumental in allowing us to deduce this relationship.

• **(ap_pass_brigade, basic_http_header)** The next cause was again the high co-variance of two functions: ap_pass_brigade, which simply calls three other filter functions, and basic_http_header, which constructs a simple HTTP response header. The root cause of the co-variance is again memory allocations that occur in each.

• **apr_bucket_alloc** The third highest contributor to overall variance is apr_bucket_alloc, which allocates memory for a bucket (container) in the Apache Portable Runtime (APR) library. APR offers a memory management library, whereby free memory is organized in slabs by size. When a slab is exhausted, it acquires extra memory from a global free list. The latency variance here stems from the different paths taken to acquire additional memory.

In light of the observation that the root causes of variance in request latencies of Apache Web Server were all memory-allocation related, we modified the memory management library to pre-allocate larger chunks of memory in advance. This simple modification eliminates 60.0% of the overall variance of Apache Web Server's request latencies.

4.8 Real-world Adoption

After observing the dramatic impact (see Table 1) of our local modifications (revealed by VProfiler), we decided to share these results with the respective open-source communities. Starting with MySQL, our VATS scheduling has been adopted by MySQL distributions and made the default scheduling algorithm by MariaDB (starting 10.2.3+) [2]. MariaDB distributions of MySQL comprise over 2M+ installations around the world.

Meanwhile, the issue with LRU mutex contention found by VProfiler was independently identified by the MySQL community [8, 12] and addressed via multi-threaded LRU flushing [8, 12] and other techniques [9, 10]. While their solution differed from our LLU technique, the bug histories still confirm the validity of VProfiler's finding regarding the cause of the performance pathology.

5. Related Work

Call Graph Profilers — These profilers gather execution times and counts for a function and its call descendants [34]. Some tools support shared subroutines, mutual recursion, or dynamic method binding [64]. There are even domain-specific techniques, e.g., for SQL applications [39]. While call graph profilers provide valuable information for long-running operations, VProfiler aims to improve predictability and thus aggregates and reports variance.

Call Path Profilers — Call path profilers can report the resource usage of function calls in their full lexical contexts [37]. Users can learn how much of a program's time is spent in specializable calls to various functions. Techniques to reduce profiling overhead include (i) sampling [33], which collects frequency counts without full instrumentation of procedures' code, and (ii) incremental profiling, which instruments only a few functions of interest [19]. Some profilers [65] extend call path profiling to parallel applications, and use semantic compression to reduce time and space overhead. VProfiler analyzes function invocations in context, but it uses a technique similar to incremental profiling by monitoring only a few functions at a time to reduce overhead. However, VProfiler aggregates over a semantic interval, and selects the most interesting functions at each iteration automatically.

Event Profilers — A variety of tracing tools collect event traces similar to the inputs to VProfiler's analysis. Many of these tools support concurrent and distributed request traces (e.g., [17], [62]). Recent frameworks provide extensible tracing, allowing users to define their own events and provide a LINQ-like query language for trace analysis (e.g. [31], [43]), which allows the introduction of concepts like our semantic intervals into the trace output. However, these profilers are not focused on analysis of variance, and do not provide an equivalent abstraction to our variance tree. VProfiler's post-processing could likely be modified to adopt such tools for managing instrumentation and generating traces in lieu of our source-to-source instrumentation.

Trace Profilers and Statistical Profilers — Trace-based profilers [13, 18, 23, 26, 41, 50, 56, 61, 67, 73] can offer detailed full-system information by instrumenting the source code, from one point in a program to another. However, such profilers are usually post-mortem and the profile data is not available during execution. Moreover, their overhead in tightly-coupled parallel applications can be quite high.

Statistical or sampling-based profilers sacrifice accuracy for lower overhead and online availability: At regular intervals, they probe the program's call stack using interrupts and collect the information they need [21, 34, 44, 45, 63, 70].

VProfiler belongs in the class of trace-based profilers. Its distinguishing contribution lies in capturing semantic intervals across interleaving threads, and identifying informative high-variance functions through the use of the variance tree.

Transactional Profiling — There has been some work on transactional profiling, wherein a transaction is a unit of work similar to our more generic concept of a semantic interval. Whodunit [22] profiles transactions in generic multi-tier applications and can track transactions that flow through shared memory, events, stages, or via message passing, and identify request types that can cause high CPU usage or high contention. VProfiler's goals are quite different from Whodunit in that it seeks to locate functions in the system that cause high variance in latency, whereas the latter focuses on automatically establishing transaction contexts and identifying *request types* that might cause high CPU utilization.

Similarly, AppInsight [55] captures the concept of a transaction for mobile applications. However, AppInsight uses a very limited notion of a transaction, as a user manipulation of the UI and all the operations it triggers.

Instead of profiling transactions, there is also some work on *passively* predicting the performance of transactions using machine learning techniques [46, 47, 75].

Performance Diagnosis — DBSherlock [76] relies on outlier detection and causality analysis to diagnose the root cause of performance anomalies from telemetry data and other statistics (collected from the application and the operating system). Chopstix [20] proposes a diagnostic tool to continuously monitor low-level OS events, such as cache misses, I/O, and locking. Reconstructing these events offline helps users reproduce intermittent bugs that are hard to catch otherwise. X-ray [16] dynamically instruments program binaries and collects performance summaries to find the root cause of performance anomalies. Reference executions can also be used to identify symptoms and causes of performance anomalies [60]. Darc [68] is able to identify the root causes of any peak for a given function by analyzing its latency distribution across multiple runs and determining the major contributor of each bucket. VarianceFinder [58] locates the root causes of variances in a distributed system by using a two-tier method. However, it focuses on the variance of requests taking exactly the same execution path. Spectroscope [59] diagnoses performance changes by comparing request flow during non-problem period and problem period.

VProfiler is also a diagnostic tool that uses instrumentation to collect information that it needs. However, unlike tools that focus on detecting individual anomalies/outliers, VProfiler's approach is based on the mathematical definition of variance. In contrast, Darc finds only the outlying latency contributors, which may or may not be related to large variance contributors. For example, in our case study on Postgres, the latency of the RecordTransactionCommit function is only 10% of the ResourceOwnerRelease function, while the former contributes 37.7% more to the overall variance than the latter. VProfiler is also capable of profiling multi-threaded programs. VarianceFinder ignores the variance caused by the difference in execution paths for the same type of requests, while VProfiler can also account for such situations. Spectroscope is not applicable to our case, as there is usually no clear definition of a problem period.

Unpredictability in Multi-tier Server Stacks — Many modern applications run in a cloud environment or on top of a complex software stack [27, 66, 77]. Here, the performance unpredictability could originate in different layers of the system [28, 30, 42], or be the result of cross-stack communications. While handling this type of unpredictability is out of the scope of this paper, we believe that variance trees will shed some light on this problem and plan to pursue this direction in the future.

6. Conclusion

We presented a novel profiler, called VProfiler, for identifying the major sources of latency variance in a semantical interval of a software system. By breaking down the variance of latency into variances and covariances of functions in the source code and accounting for thread interleavings, VProfiler makes it possible to calculate the contribution of each function to the overall variance. Using VProfiler, we analyzed the codebases of three complex open-source systems, leading us to small modifications that significantly reduced performance variance in these popular applications.

7. Acknowledge

This work in part supported by National Science Foundation (grants 1544844, 1629397, and 1553169). The authors are also grateful to Jarrid Rector-Brooks for his contributions to VProfiler's implementations, and to Morgan Lovay and Shannon Riggins for their insightful comments on this manuscript, and to Mark Callaghan, Jan Lindström, and Laurynas Biveinis for their great help and guidance in integrating VATS into MySQL and MariaDB distributions.

References

[1] Erratic performance problems in oracle > 8.0.x. http://www.wikiguga.com/topic/c67f177136f542809da3106c5f875e68.

[2] Mariadb source repository. https://github.com/MariaDB/server/pull/248.

[3] Google on latency tolerant systems: Making a predictable whole out of unpredictable parts. http://highscalability.com/blog/2012/6/18/google-on-latency-tolerant-systems-making-a-predictable-whol.html, 2002.

[4] Understanding flash: Unpredictable write performance. http://flashdba.com/2014/12/10/understanding-flash-unpredictable-write-performance/, 2004.

[5] Server unpredictable random performance hiccups. https://www.percona.com/forums/questions-discussions/mysql-and-percona-server/401-server-unpredictable-random-performance-hiccups, 2006.

[6] Erratic performance of SQL Server. http://www.bigresource.com/MS_SQL-Erratic-Performance-of-SQL-Server-6rJpQdOC.html, 2007.

[7] Query execution time is unpredictable. https://community.oracle.com/thread/961135?start=0&tstart=0, 2009.

[8] Mt lru flusher. https://blueprints.launchpad.net/percona-server/+spec/mt-lru, 2011.

[9] Parallel doublewrite buffer. https://blueprints.launchpad.net/percona-server/+spec/parallel-doublewrite, 2011.

[10] Xtradb performance improvements for i/o-bound highly-concurrent workloads. https://www.percona.com/doc/percona-server/5.7/performance/xtradb_performance_improvements_for_io-bound_highly-concurrent_workloads.html, 2011.

[11] Unpredictable performance and slowness between our azure website and a sql server running on an azure vm. http://tinyurl.com/qzleb43, 2015.

[12] Percona server 5.7: multi-threaded lru flushing. https://www.percona.com/blog/2016/05/05/percona-server-5-7-multi-threaded-lru-flushing/, 2016.

[13] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, 2003.

[14] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage; recovery methods for non-volatile memory database systems. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2015.

[15] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2010.

[16] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[17] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[18] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Hot Topics in Operating Systems (HotOS)*, 2003.

[19] A. R. Bernat and B. P. Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 2007.

[20] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[21] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications (IJHPCA)*, 2000.

[22] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. *ACM SIGOPS Operating Systems Review*, 2007.

[23] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks (DSN)*, 2002.

[24] S. Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2009.

[25] D. S. Daniels, A. Z. Spector, and D. S. Thompson. Distributed logging for transaction processing. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 1987.

[26] L. De Rose, Y. Zhang, and D. A. Reed. Svpablo: A multi-language performance analysis system. In *Computer Performance Evaluation*. Springer, 1998.

[27] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[28] C. Delimitrou and C. Kozyrakis. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[29] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking rela-

tional databases. *Proceedings of the VLDB Endowment (PVLDB)*, 7, 2013.

[30] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: understanding the impact of limpware on scale-out cloud systems. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[31] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[32] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2009.

[33] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *International Conference on Supercomputing (ICS)*, 2005.

[34] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, 1982.

[35] B. Gregg. DTrace pid Provider return. http://tinyurl.com/jzpphne, 2011.

[36] B. Gregg. Dtrace tools. http://www.brendangregg.com/dtrace.html, 2011.

[37] R. J. Hall. Call path profiling. In *International Conference on Software Engineering (ICSE)*, 1992.

[38] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2005.

[39] J. M. Hill, S. A. Jarvis, C. Siniolakis, and V. P. Vasilev. Analysing an sql application with a bsplib call-graph profiling tool. In *Euro-Par98 Parallel Processing*, 1998.

[40] J. Huang, B. Mozafari, G. Schoenebeck, and T. Wenisch. A top-down approach to achieving performance predictability in database systems. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2017.

[41] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1987.

[42] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[43] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[44] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 2004.

[45] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 1995.

[46] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2013.

[47] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[48] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2015.

[49] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: An extended report. Technical report, University of Michigan, Ann Arbor, 2015.

[50] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.

[51] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[52] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman. Main-memory scan sharing for multi-core cpus. *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[53] F. Raab. Tpc benchmark c, standard specification revision 3.0. *Transaction Processing Performance Council*, 1995.

[54] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *International Conference on Data Engineering (ICDE)*, 2008.

[55] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[56] D. A. Reed, P. C. Roth, R. A. Aydt, K. A. Shields, L. F. Tavera, R. J. Noe, and B. W. Schwartz. Scalable performance analysis: The pablo performance analysis environment. In *Scalable Parallel Libraries Conference (SPLC)*, 1993.

[57] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-i/o friendly using ssds. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[58] R. R. Sambasivan and G. R. Ganger. Automated diagnosis without predictability is a recipe for failure. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[59] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[60] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS Performance Evaluation Review*, 2009.

[61] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using c++. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.

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

[63] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *International Conference on Cluster Computing (CLUSTER)*, 2002.

[64] J. M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 2004.

[65] Z. Szebenyi, F. Wolf, and B. J. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[66] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[67] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *High Performance Distributed Computing (HPDC)*, 1998.

[68] A. Traeger, I. Deras, and E. Zadok. Darc: Dynamic analysis of root causes of latency distributions. In *ACM SIGMETRICS Performance Evaluation Review*, 2008.

[69] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[70] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *ACM SIGMETRICS Performance Evaluation Review*, 2002.

[71] M. Wainwright. Chapter 2: Basic tail and concentration bounds. http://www.stat.berkeley.edu/~mjwain/stat210b/Chap2_TailBounds_Jan22_2015.pdf.

[72] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[73] J. C. Yan. Performance tuning with aims/spl minus/an automated instrumentation and monitoring system for multicomputers. In *Twenty-Seventh Hawaii International Conference (HICSS)*, 1994.

[74] R. J. Yang and Q. Luo. PTL: Partitioned logging for database storage on flash solid state drives. In *Web-Age Information Management (WAIM)*. 2012.

[75] D. Y. Yoon, B. Mozafari, and D. P. Brown. DBSeer: Pain-free database administration through workload intelligence. *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[76] D. Y. Yoon, N. Niu, and B. Mozafari. DBSherlock: A performance diagnostic tool for transactional databases. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2016.

[77] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.