A Top-Down Approach to Achieving Performance Predictability in Database Systems

Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, Thomas F. Wenisch
University of Michigan, Ann Arbor
{jiamin, mozafari, schoeneb, twenisch}@umich.edu

ABSTRACT

While much of the research on transaction processing has focused on improving overall performance in terms of throughput and mean latency, surprisingly less attention has been given to performance predictability: how often individual transactions exhibit execution latency far from the mean. Performance predictability is increasingly important when transactions lie on the critical path of latency-sensitive applications, enterprise software, or interactive web services.

In this paper, we focus on understanding and mitigating the sources of performance unpredictability in today's transactional databases. We conduct the first *quantitative study* of major sources of variance in MySQL, Postgres (two of the largest and most popular open-source products on the market), and VoltDB (a non-conventional database). We carry out our study with a tool called TProfiler that, given the source code of a database system and programmer annotations indicating the start and end of a transaction, is able to identify the dominant sources of variance in transaction latency. Based on our findings, we investigate alternative algorithms, implementations, and tuning strategies to reduce latency variance without compromising mean latency or throughput. Most notably, we propose a new lock scheduling algorithm, called Variance-Aware Transaction Scheduling (VATS), and a lazy buffer pool replacement policy. In particular, our modified MySQL exhibits significantly lower variance and 99th percentile latencies by up to 5.6× and 6.3×, respectively. Our proposal has been welcomed by the open-source community, and our VATS algorithm has already been adopted as of MySQL's 5.7.17 release (and been made the default scheduling policy in MariaDB).

1. INTRODUCTION

Transactional databases are a mission-critical component of enterprise software for efficient storage and manipulation of data. A significant portion of database research on transaction processing has focused on improving overall performance and scalability, for example, by developing new techniques for concurrency control, query optimization, indexing, and other sophisticated ideas. These strategies, however, have been vetted primarily in terms of their effect on the *average performance* of the database, such as its throughput and mean transaction latency. In other words, the focus has been on average performance and running more and faster transactions *overall*.

While peak transaction processing throughput is clearly important, the *predictability* of performance—the disparity between average and high-percentile tail latencies—has emerged as an equally important metric in situations where individual transaction latencies are mission-critical or affect end-user experience. Examples include database-backed web services or database clouds with service-level agreements.[1] However, performance predictability has often been overlooked by traditional efforts that focus on throughput and mean latency. In fact, some optimization strategies (e.g., asynchronous logging and group commit [41, 63]), have deliberately improved throughput at the expense of penalizing latency for *some* transactions. In other cases, developers have not even vetted the impact of their design and implementation decisions on performance predictability. For example, while the contribution of database components to mean latency has been studied [39], an analogous study to identify the sources of latency variance is missing. As such, today's complex DBMSs might have overlooked alternative design decisions that could deliver the same or comparable average performance, but at a significantly lower variance.

At the fine time-scale of individual transactions, the performance of existing databases is incredibly unpredictable, with orders of magnitude gaps between mean and high percentile latencies[2] (see Appendix C.1). Two approaches can be taken to achieve performance predictability.

Bottom-up vs. Top-down Approach — In a bottom-up approach, one could aim to build an entirely new DBMS from scratch that is specifically designed to be predictable [34, 65]. Despite their merits, these proposals have not had widespread adoption for transaction processing. One reason is that users are often reluctant to completely abandon well-established and matured DBMSs in exchange for academic prototypes. Another reason, however, is that these propos-

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

 ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064016

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

[2] While some of this variance is inherent and due to some transactions doing more work than others, our study reveals that dominant sources of variance are often a performance pathology and avoidable (see Section 2 for the distinction).

als promise higher throughput or predictability by sacrificing mean latency (e.g., by always using scan-only plans [65]). Despite their success in long-running analytics [58], such tradeoffs are less appealing to transactional and latency-critical applications. Instead, an ideal solution is one that delivers the same mean latency and throughput as existing solutions, but with much lower variance. Adopting such solutions, especially if compatible with existing DBMSs, would be a "no-brainer" for most users (see Section 9).

Thus, in this paper, we advocate a top-down approach, wherein we identify and mitigate the performance pathologies that lead to variance in existing, widely used transaction processing systems—an approach that can have more immediate impact on real-world deployments. As such, we address some of today's popular DBMSs in an attempt to understand their major sources of variance and seek design alternatives for overcoming variance-inducing performance pathologies. For the same reason, we also restrict ourselves to solutions that reduce variance without sacrificing mean latency or throughput. In addition to the immediate benefits to massive user-bases of these products, the insight gained in this process can inform future bottom-up attempts at designing new databases.

Challenges — A top-down approach, however, comes with its own challenges. Gaining performance insight into any software system as complex as a DBMS requires effective profiling tools. Unfortunately, existing profilers can only study a system in terms of its *average performance*, e.g., by breaking down overall run time into the average latency of individual functions. As we will show in this paper, quantifying the contribution of individual functions to overall variance is much more challenging. Moreover, latency variance of each function is only important to the performance profile insomuch as it affects transaction latencies. For example, it may not matter if a background I/O operation exhibits large variance in execution time, as long as the user-perceived latency of a transaction is unaffected.

Why Now? — It is both critical and timely to systematically study and manage performance variance of transactional databases for several reasons. First, advancements in hardware parallelism and better transaction processing techniques have enabled microsecond latencies and millions of concurrent transactions [21, 47, 73]. As mean performance improves, the impact of performance perturbations (e.g., due to a slow I/O request) relative to the latency of a transaction grows. Second, an increasing number of DBaaS providers guarantee service level agreements (SLAs), which if violated,—even for a subset of transactions—result in financial penalties [2, 4, 53, 54]. Finally, modern DBMSs have become some of the most complex software systems. As such, subtle interactions of complex code paths lead to vexing performance anomalies.

Our Solution — We systematically study the sources of variance in the call graphs of three complex transaction processing systems. To facilitate these studies, we use a profiling tool called TProfiler that, given the source code of a database management system and a minimal effort to annotate the start and ends of transactions, can identify dominant sources of latency variance. [3] To minimize the overhead of collecting fine-grain performance measurements, TProfiler runs in multiple iterations, each time instrumenting a carefully selected subset of functions invoked during transaction processing. By analyzing these measurements across thread interleavings, TProfiler aggregates latency across all functions contributing to a transaction. TProfiler reports variance results using a representation we call a *variance tree*, which enables the developer to reason about the relationship between overall latency variance and the variances and covariances of individual culprit functions.

We use TProfiler to analyze the codebases of three open-source database systems with drastically different designs: MySQL (a thread-per-connection model), Postgres (a process-per-connection model), and VoltDB (an event-based sever model). VoltDB is a more modern engine, while MySQL and Postgres are two of the most commonly-used databases today (millions of active users), each having a massive and complex codebase (e.g., MySQL has 1.5M lines of code and 30K functions). Finally, based on TProfiler's findings, we propose both generic and DBMS-specific strategies for reducing performance variance.

Contributions — We make the following contributions:

1. We analyze variance in the MySQL codebase and discover that varying delays due to lock scheduling are a dominant source of latency variance. We further identify the LRU policy as another cause of variance in memory-intensive workloads. We also analyze the Postgres and VoltDB codebases, finding various delays in logging and work queues as their top causes of latency variance, respectively (Section 4).[4]

2. Almost all DBMSs grant locks on a first-come-first-served basis, which is identified by TProfiler as a major cause of variance. We thus propose a *variance-aware transaction scheduling* (VATS) algorithm, as a general technique for reducing variance. By minimizing the L_p norm, VATS simultaneously reduces mean, variance, and high percentiles of transaction latencies. We prove that, without any prior knowledge of transactions' remaining times, VATS is the optimal strategy (Section 5).

3. We also propose DBMS-specific variance reduction strategies, including a *lazy LRU update* policy for MySQL that significantly reduces contention, and variance-aware tuning guidelines for database administrators (Section 6).

4. Through extensive experiments on five different benchmarks, we confirm that our techniques make these DBMSs significantly more predictable (and even faster) without compromising throughput, with 2.9x, 2.8x, and 1.5x lower mean, variance, and 99th percentile latencies, respectively. (and up to 6.3x, 5.6x, and 2.0x, respectively.) (Section 7). In the case of MySQL, a highly popular DBMS, our findings have already been adopted (Section 9).

2. BACKGROUND

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

[3] TProfiler is open source: https://web.eecs.umich.edu/tprofiler/

[4] While some of these findings may not seem surprising, the value of TProfiler is in *quantifying* their impact on overall variance and *narrowing* our search to a handful of functions in a massive codebase with millions of functions. For example, while there are 100's of functions that incur wait times, only a few specific instances contribute to overall variance.

Defining Predictability — There are different mathematical notions for capturing *performance predictability*. One could *minimize* latency variance or seek to impose bounds on high percentile latencies (e.g., limiting 99th percentile latency). Statisticians have also used the ratio of standard deviation to mean (a.k.a coefficient of variation) as a standardized measure of dispersion for a distribution.

Since in this paper we target statistical (rather than hard) guarantees, we focus on identifying the sources of latency variance (and thereby standard deviation). Minimizing variance also serves as a surrogate for reducing high-percentile latencies [66]. Hence, our techniques reduce both latency variance and 99th percentile latency (see Section 7).

Desirable Solutions — Simply padding all latencies with a large wait time would trivially reduce variance but would also increase mean latency, and, thus, it would have little practical value. While long-running queries and OLAP applications might tolerate an increase in mean latency in exchange for predictability or higher throughput [16, 17, 35, 58, 59, 65], the same tradeoff is less appealing to many latency-critical OLTP applications.[5] 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, not only do our findings reduce variance, but they also *reduce* mean latency and coefficient of variation (see Section 7). Such solutions are much more desirable in practice. For example, some of this paper's findings have been already adopted (and even made a default policy) by some of the largest open-source communities (Section 9).

Inherent versus Avoidable Variance — It is important to note that performance variance is sometimes inherent and cannot be avoided. For example, processing a transaction that updates 10 tables inherently involves more work than one that updates only one table.[6] Avoidable sources of variance are those that are not caused by varying amounts of work requested by the user, but are rather due to internal artifacts of the DBMS itself, such as scheduling choices, contention, I/O, or other performance pathologies in the source code. For example, two transactions requesting similar amounts of work, but experiencing different latencies, indicate a performance anomaly that might be avoidable.

3. TPROFILER

Although existing tracing tools (e.g., DTrace [37]) can provide performance insights for transaction systems, these tools are poorly suited for identifying the root causes of performance variation. Existing tracing frameworks are typically oriented towards aggregating and reporting performance results in terms of the call graph of an application. A seemingly intuitive approach to analyze and summarize performance variation might be to record and report the standard deviation or high quantile latencies of individual functions in the call graph. We find this approach dissatisfying for two reasons. First, conventional profiling tools are oblivious of the notion of a transaction, which is fundamental to the way transaction processing experts wish to reason about performance. As noted earlier, some variance is inherent and is correlated to the particular transaction type. By making transactions explicitly visible to the tracing infrastructure, it is possible to isolate and reason about inherent vs. avoidable variance. Second, because of the nature of how variance mathematically aggregates, the variance of a parent function is always strictly greater than the variance of its children. Hence, the highest-variance functions always lie at the roots of the call hierarchy. However, these functions are typically the least informative in discovering root causes of performance pathologies. Rather than operate solely on variance, TProfiler instead ranks functions using a score function that considers both variance and depth within the call hierarchy when deciding which functions to highlight as the likely root causes of performance variability.

A further key shortcoming of fine-grain performance analysis tools is that they often incur significant instrumentation overheads that slow the database system, sometimes by substantial factors. Significant slowdowns are problematic because they may alter the relative latencies of application behaviors (for example, slowing the relative performance of a memory-intensive code sequence as compared to a system call or lock acquisition). As such, the variance profile observed in an instrumented execution may differ materially from the profile that arises when the database runs at full performance.

3.1 Overview

To study an application with TProfiler, the developer performs an iterative refinement procedure consisting of three repeating steps: automatic source code instrumentation, online trace collection, and offline variance analysis. In each iteration of this flow, a subset of an application's call graph is instrumented to collect the contribution of each function to transaction latency and variance. We instrument only a subset of the call graph at a time to limit instrumentation overhead; instrumenting every function distorts the latency profile such that it is not representative of unprofiled execution. TProfiler outputs a profile that identifies the top-k highest scoring functions that account for transaction variance (we discuss the scoring function below). The developer then examines this profile to determine if the highlighted functions are sufficiently detailed to identify key sources of variance. If not (i.e., the highlighted functions are too high in the call hierarchy to be informative), the programmer indicates functions for which she wishes the variance to be further decomposed. The children of these functions are then added to the list of functions to be profiled, and the source code is automatically re-instrumented to monitor this new set of functions to collect a new profile. In our experience, this cycle must be repeated a handful of times (perhaps as much as ten) to construct the most informative profile.

TProfiler requires the programmer to manually demarcate the start and end of transactions, using a simple API. We perform this step manually because it is not clear how an automated tool can deduce a semantic notion like "transaction start" from arbitrary code. However, in all three of the transaction processing systems we study, it is quite easy for a programmer to do so. The programmer must indicate where a transaction begins and where a transaction completes. In MySQL and Postrgres, transaction execution corresponds to the invocation of a single function that is the root of a large call hierarchy. In VoltDB, transactions do not correspond

[5]This is perhaps why, despite the success of scan-only query plans in OLAP [25, 58], similar proposals [65] have not had widespread adoption in transaction processing.

[6]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 [51, 52].

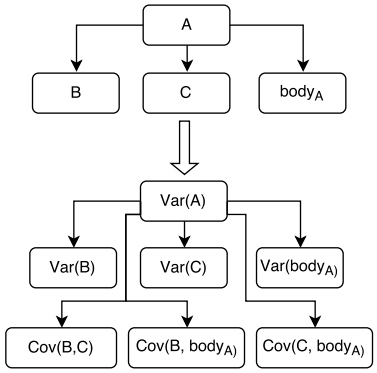

Figure 1: A call graph and its corresponding *variance tree* (here, $body_A$ represents the time spent in the body of A).

to threads or function invocations on a one-to-one basis, as VoltDB uses a task-concurrent execution model. Instead, in VoltDB, we instrument when a thread begins and ends an interval of execution on behalf of a particular transaction, based on a transaction id associated with a task. The remainder of the analysis concatenates these execution intervals. A transaction begins at the first interval labelled by its transaction id and ends at the last. Whereas these simple annotations are sufficient for the transaction processing systems studied here, they are insufficient to profile arbitrary concurrent programs, where multiple threads may concurrently execute on behalf of a single logical unit of work and labelled intervals may overlap. We have generalized our profiling and instrumentation approach to analyze the critical path in applications with such overlaps in related work [42].

3.2 Characterizing Execution Variance

TProfiler analyzes performance variance by comparing the duration of particular function invocations in a transaction across other invocations of the same function in different transactions. TProfiler uses an abstraction we have developed, the *variance tree*, to reason about the relationship between latency variance in the call hierarchy rooted at a particular function invocation. TProfiler subdivides and attributes execution time across the call graph, similar to conventional profiles from tools like `gprof` [36]. Using these fine-grain latency measurements, TProfiler then calculates the variance and covariance of each component of the call graph across the many invocations to identify functions that contribute the most variability.

Figure 1 (up) depicts a sample call graph comprising a function A invoking two children B and C, and includes the execution time in the body of A. We deconstruct and represent the variance of the call graph using expression 1. Figure 1 (bottom) shows a corresponding visualization of the variances and covariances in a variance tree.

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

The variance tree allows TProfiler to quickly identify subtrees that do not contribute to latency variability, as their variance is (relative to other nodes) small. These sub-trees do not require any further scrutiny, and there is no need to further instrument the call stack in these sub-trees to break down variance. Identifying the root causes of large variance, however, is not so trivial. As noted previously, 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 diagnosing pathologies. Moreover, some variance is inherent, due to variation in the work per transaction; such variance is not an indication of a mitigable pathology. High covariance across pairs of functions can be an indicator of a correlation between the amount of work performed by such functions.

Instead, we 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, i.e., analyzing them reveals insight about why variance occurs. To unify terminology, we refer to the variance of a function or covariance 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 how to identify functions that are informative. Our intuition is that functions deeper in the call graph implement more specific functionality, and hence are more likely to reveal the root cause of latency variance. TProfiler 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. For functions called from multiple sites, TProfiler aggregates the variance/covariance across all call sites. TProfiler assigns each function a height based on the maximum depth of the call tree beneath it (for covariance of two functions, the larger height is used). We use a specificity metric that is a decreasing function of the height of a factor ϕ:

$$specificity(\phi) = (height(call\_graph) - height(\phi))^2 \quad (2)$$

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

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

$$score(\phi) = specificity(\phi) \sum_i V(\phi_i) \quad (3)$$

where $V(\phi_i)$ represents variance or covariance of a specific call site of a factor within the call graph. TProfiler then selects the top-k factors of the tree based on their scores.

4. CASE STUDIES

In this section, we conduct a case study of MySQL and Postgres (as popular open-source DBMSs) to identify their main causes of latency variance, and defer a similar study of VoltDB to Appendix A.

4.1 Latency Variance in MySQL

In this section, we use TProfiler to analyze the source code of MySQL 5.6.23 and characterize the main sources of variance therein. Here, we only report our findings using the TPC-C benchmark. However, in Section 7 we evaluate our techniques using five different benchmarks with various degrees of complexity and contention.

Setup — We use the OLTP-Bench [32] framework to run the TPC-C workload under two configurations. First, we study a 128-warehouse configuration with a 30 GB buffer pool on a system with 2 Intel(R) Xeon(R) CPU E5-2450

Config	Function Name	Percentage of 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 1: Key sources of variance in MySQL.

processors and 2.10GHz cores. Second, we study a reduced-scale 2-warehouse configuration with a 128M buffer pool on a machine with 2 Intel Xeon E5-1670v2 2.5GHz virtual CPUs. The reduced-scale configuration exaggerates buffer pool contention, revealing latency sources that may arise in workloads with a working set significantly larger than the available memory. We refer to these configurations as 128-WH and 2-WH, respectively. In both cases, we use a separate machine to issue client requests to the MySQL server.

Summary — Table 1 summarizes the key variance sources in MySQL identified by TProfiler. Whereas MySQL has one of the most complex code bases with over 1.5M lines of code and 30K functions, TProfiler narrows down our search by automatically identifying a handful of functions that contribute the most to the overall transaction variance. This demonstrates TProfiler's value: one only needs 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 explain the role of each function found by TProfiler.

os_event_wait() — MySQL uses its own cross-platform API for synchronization; `os_event_wait` is one of the central functions in this abstraction layer. The implementation of `os_event_wait` yields little insight into why the transaction has to wait. We thus examine the context for the two most significant call sites invoking `os_event_wait` (referred to as A and B in Table 1). Both call sites occur within `lock_wait_suspend_thread`, a function used to put a thread to sleep when its associated transaction requests a lock on a record that cannot be granted due to a conflict. These two specific call sites correspond to locks acquired during select and update statements, respectively.

This implies that variability of wait time for contended locks is the largest source of variance in MySQL. Motivated by this finding, we later propose a variance-aware transaction scheduling in Section 5, which seeks to minimize variance of wait times by optimizing the order in which locks are granted to waiting threads.

row_ins_clust_index_entry_low() — This function inserts a new record into a clustered index. TProfiler reports that none of this function's children exhibit significant variance, but the main variance arises in the body of the function itself due to varying code paths taken based on the state of the index prior to the insert. The variance here is thus inherent to the index mutation, not a performance pathology.

buf_pool_mutex_enter() — This function is called by other functions when they access the buffer pool. This function is called from various sites, but the call most responsible for its variance occurs in `buf_page_make_young`, which moves a page to the head of the `LRU` list. InnoDB uses this list to maintain the order of buffer page replacements based on a variant of the least recently used algorithm. Upon certain types of accesses, a page is moved to the head of the `LRU` list. Threads must acquire a lock before modifying the list. That lock is acquired in `buf_pool_mutex_enter`. The variance in this function reflects varying wait times, while other threads are reordering the `LRU` list. In Section 6.1, we propose a strategy for mitigating this problem.

Function Name	Percentage of Overall Variance
LWLockAcquireOrWait	76.8%
ReleasePredicateLocks	6%

Table 2: Key sources of variance in Postgres.

btr_cur_search_to_nth_level() — This function traverses an index tree level by level to place a tree cursor at a given level, and then it leaves a shared or exclusive lock on the cursor page. Its runtime, thus, varies with the depth to which the tree must be traversed. The variance here is inherent to the index traversal, not a performance pathology.

fil_flush() — MySQL uses `fil_flush` to flush redo logs generated by a transaction. When the operating systems uses disk buffering, the latency variance of disk I/O is exposed in `fil_flush` (rather than the `write` system calls). The variance here is inherent to the I/O, but might be mitigated by logging to faster I/O devices, e.g., [18, 56, 67].

4.2 Latency Variance in Postgres

In this section, we use TProfiler to analyze the source code of Postgres 9.6—another popular DBMS. For Postgres, we use the same setup as in Section 4.1. Here, we use TPC-C with 32-warehouses and a 30 GB buffer pool. Table 2 shows the top two functions in the Postgres source code identified by TProfiler as the main sources of variance (the top source dominates, accounting for 76.8%).

LWLockAcquireOrWait() — Postgres uses write-ahead logging for atomicity and durability; before a transaction commits, all its redo logs must be flushed to disk. To ensure that only one transaction is flushing redo logs at a time, each transaction calls the `LWLockAcquireOrWait` function to acquire a single global lock, called `WALWriteLock`, before writing its logs. The latency variance in `LWLockAcquireOrWait` is due to varying wait times to acquire this lock. A natural solution is to either reduce contention on this global lock, or to allow multiple transactions to flush simultaneously. The former may be attempted by accelerating I/O (e.g., tuning the I/O block size or placing the logs on NVRAM [18, 67] or SSD [28, 60]), whereas the latter can be attempted by a variety of parallel logging schemes (e.g., [14, 15, 69]). Both strategies have proven effective in improving throughput and mean latencies [18, 67]. However, TProfiler's findings, regarding `LWLockAcquireOrWait`'s contribution to the overall latency variance, call for vetting these strategies in terms of improving predictability as well. We study some of these ideas for Postgres in Sections 6.2 and 7.4.

ReleasePredicateLocks() — Postgres uses predicate locking to avoid phantom problems (read conflicting with later inserts). As a transaction accesses rows in the database, locks are acquired to prevent other transactions from inserting new rows into its selected range. Upon commit, all its predicate locks are released by calling `ReleasePredicateLocks`. The execution time of this function varies with the number and type of conflicts discovered during this release phase.

Since `ReleasePredicateLocks` accounts for only 6% of overall variance, we do not pursue it further.

5. VARIANCE-AWARE TRANSACTION SCHEDULING

According to TProfiler's findings from Section 4, lock wait times account for a significant portion of overall latency variance (over 59.2% in case of MySQL). Hence, in this section we propose a lock scheduling algorithm that can dramatically reduce latency variance.

5.1 Problem Setting

Traditional databases often rely on variants of 2-phase locking (2-PL) for concurrency control [1, 5, 7]. Conceptually, each database object b has its own queue Q_b. When a transaction T requests a lock on b, the lock is immediately granted if (i) no other locks are currently held on b by other transactions, or (ii) the current locks on b are compatible with the requested lock type and there are no other transactions currently waiting in Q_b.[7] When a lock on b cannot be granted immediately, transaction T is suspended and placed in Q_b until its lock can be granted. In general, each transaction may wait in multiple queues during its lifetime, and each queue may contain multiple transactions waiting in it. Let $Q_b = \{T_1, \cdots, T_n\}$ denote the transactions currently waiting to be granted a lock on b.

Whenever all the currently held locks on b are released, the *lock scheduling* (a.k.a. *transaction scheduling*) problem is the decision regarding which transaction(s) in Q_b must be granted the lock next. The transaction scheduler might choose one of the exclusive (e.g., write) requests, or choose one or more of the inclusive ones.

The default transaction scheduling in many databases (including MySQL [9] and Postgres [12] among others) is the First-Come-First-Served (FCFS) algorithm. In FCFS, whenever the lock on b becomes available, the transaction which has arrived in Q_b the earliest, say T_e, is granted the lock. Additionally, all the other transactions in Q_b whose requests are compatible with that of T_e are also granted a lock. In other words, T_e is selected based on the amount of time it has spent in the current queue (not in the system). Fairness and simplicity have contributed to FCFS's popularity. However, FCFS does not even minimize mean latency, let alone latency variance.

Challenge of unpredictable remaining times — A key challenge in transaction scheduling is the lack of prior knowledge regarding a transaction's remaining time. In other words, when a transaction arrives in Q_b, the system is only aware of its *age* (i.e., elapsed since its birth), but does not know when it will finish and release its locks once it is granted a lock on b. For example, it may need to wait on a few other locks before it can proceed to completion. In fact, our studies reveal that there is very little correlation between a transaction's age and its overall latency in practice (Appendix C.2). Thus, any scheduling strategy must account for the fact that remaining times are unknown and hard to estimate.

A Convex Loss Function — As discussed in Section 2, our ultimate goal is to reduce latency variance and tail latencies. However, solely minimizing variance as a loss function may lead to undesirable side effects. For example, a scheduling algorithm that deliberately adds a large delay to every completed transaction (before allowing it to leave the system) will have a near-zero variance. However, it will also significantly increase mean latency, and is, hence, impractical. To exclude such algorithms, a more effective loss function is the so-called L_p norm, which if minimized, will indirectly reduce both mean and variance (and, thereby, tail) latencies. When n transactions finish with latencies $\langle l_1, \cdots, l_n \rangle$, their L_p norm (denoted as $||.||_p$) is defined as

$$L_p = ||\langle l_1, \cdots, l_n \rangle||_p = (\sum_{i=1}^{n} |l_i|^p)^{1/p} \quad (4)$$

where $p \geq 1$ is a real-valued number. The larger the p value, the more we penalize deviations of the l_i values from the mean. For example, as $p \to \infty$, L_p norm approaches the max value of the list. A typical value of p in practice is 2. However, our results in this section hold for all $p \geq 1$ values.

5.2 Our VATS Algorithm

Let $A(T)$ denote the age of transaction T when it arrives at a queue Q_b. Q_b is the set of transactions waiting to be granted a lock on b. We define the history H_b of an object b to be the schedule of prior (and current) transactions holding a lock on b. In the following, we drop b from our notation for convenience. Let F be some advice about the future (our algorithm will not make use of such advice, but we will compare our algorithm to other algorithms that may).

A scheduler $S = (S_f, S_a)$ is a set of two functions: $S_f, S_a : H \times Q \times F \to 2^Q$. When the lock becomes available, the function S_f determines which transactions from Q should be granted a lock. S_f cannot grant two exclusive locks on b simultaneously. When a new transaction arrives at Q, the function S_a decides which transactions should be granted a lock. When other locks are currently held, S_a can only choose from transactions acquiring inclusive locks compatibile with the currently held locks.

Let $R(T)$ be a random variable indicating T's remaining time once it is granted a lock on b. Finally, let a menu M be a sequence of transactions, where each transaction has an age and an arrival time at the queue. This will define a problem instance.

We define the p-performance of a schedule S on a menu M to be the expected L_p norm of the vector of transaction completion times of S on M.

Our Algorithm — Given a menu, we aim to design a scheduler that minimizes the expected p-performance. To this end, we define our scheduler as S^{VATS}=(S_f^{VATS}, S_a^{VATS}) where:

- S_f^{VATS} grants the lock to the eldest transaction, i.e., one with the largest age.
- S_a^{VATS} never grants any locks.

In general, optimal scheduling is an NP-complete problem when the $R(T)$ values are known [57]. Additionally, the online problem of scheduling even on one processor is impossible to do with a competitive ratio of $O(1)$.[8]

[7] To prevent the writes from starving, new read requests may not be granted if there are write requests ahead of them.

[8] That is, for every scheduler S, there exists a menu M where the optimal offline algorithm performs $\omega(1)$ better than S.

Interestingly, and counter-intuitively, we show that optimal scheduling becomes easier when the remaining times are not known! Specifically, we avoid the above negative results by assuming that $R(T)$ values are i.i.d. random variables drawn from some (unknown) distribution D.[9]

We now show that our VATS algorithm performs optimally, even against algorithms that know the distribution D (i.e., algorithms that receive $F = D$ as an advice). Note that VATS does not use or need any distributional information or advice on future. Interestingly, this holds even if the menu and distribution are chosen adversarially.

THEOREM 1. *Fix any menu M, $p \geq 1$, and distribution D with finite expected L_p norm. Let the $R(T)$s be i.i.d random variables drawn from D. Then the p-performance of* VATS *is optimal against all schedulers, even those that are given D as advice about the future.*

PROOF. Assume for the sake of contradiction that there exists a menu M of ℓ transactions $T_1, T_2, \ldots, T_\ell$, where a schedule S has p-performance better than S^{VATS}. We will transform S into S^{VATS} by a series of ℓ transformations: $S = S_0 \to S_1 \to S_2 \to \cdots \to S_\ell = S^{VATS}$. We will show after each transformation that the performance of the schedule improves. This yields a contradiction to the assumption that the p-performance of S was better than that of S^{VATS}.

In the kth transformation, we modify S_{k-1} so that if ever S_{k-1} schedules a transaction $T_{k'} \neq T_k$ when T_k is the eldest transaction in the queue, then S_k will transpose the order of T_k and T'_k, but otherwise run identically to S_{k-1}.

Note that $S_\ell = S^{VATS}$, because S_ℓ will run the eldest transaction, no matter which one it is.

Let $T_{S_{k-1},1}, T_{S_{k-1},2}, \ldots, T_{S_{k-1},\ell}$ and $T_{S_k,1}, T_{S_k,2}, \ldots, T_{S_k,\ell}$ be the order of transactions scheduled in S_{k-1} and S_k respectively. Note that these may be random variables, in that the ith transaction scheduled might depend on the randomness of the scheduler, as well as the time that previous transactions held onto the lock. Let $U_S(T)$ be the time it takes between when T arrives and when the lock is first free under schedule S. Let $W_S(T)$ be the set of transactions scheduled while T is in the queue (including T) under schedule S.

To compare the performance of S_{k-1} and S_k, we create a coupling between two different drawings D_1 and D_2 of the $R(\cdot)$s so that for all i, $R_{D_1}(T_{S_{k-1},i}) = R_{D_2}(T_{S_k,i})$. First note that there is no dependency problem here because (by induction on i) under this coupling, $T_{S_{k-1},i}$ and $T_{S_k,i}$ will be scheduled at the same time. Also, since the $R(\cdot)$s are all drawn i.i.d, this is a valid coupling, which is to say that D_1 and D_2 are (marginally) drawn from the same distribution. Note that the performance of S_{k-1} and S_k are respectively,

$$\int_{D_1} \left(\sum_i |A[T_{S_{k-1},i}] + U_{S_{k-1}}(T_{S_{k-1},i}) + \sum_{T_j \in W_{S_{k-1}}(T_{S_{k-1},i})} R(T_j)|^p \right)^{1/p}$$

and

$$\int_{D_2} \left(\sum_i |A[T_{S_k,i}] + U_{S_k}(T_{S_k,i}) + \sum_{T_j \in W_{S_k}(T_{S_k,i})} R(T_j)|^p \right)^{1/p}$$

To show that the first is greater than the second, we fix some realization of D_1. Using our coupling, this gives us a realization of D_2. We will show that no matter what the realization is we have:

$$\sum_i |A[T_{S_{k-1},i}] + U_{S_{k-1}}(T_{S_{k-1},i}) + \sum_{T_j \in W_{S_{k-1}}(T_{S_{k-1},i})} R(T_j)|^p$$
$$< \sum_i |A[T_{S_k,i}] + U_{S_k}(T_{S_k,i}) + \sum_{T_j \in W_{S_k}(T_{S_k,i})} R(T_j)|^p$$

Note that the summands are identical except, possibly, for the terms of T_k and $T_{k'}$. Let $W_k = W_{S_{k-1}}(T_k) \cap W_{S_k}(T_k)$ be the transactions scheduled while T_k is in the queue in both schedules. Define $W_{k'}$ analogously. Let W' be the transactions scheduled between k and k'. Then, $W_{S_{k-1}}(T_k) = W_k \cup \{T_{k'}\} \cup W'$, $W_{S_{k-1}}(T_{k'}) = W_{k'}$, $W_{S_k}(T_k) = W_k$, and $W_{S_k}(T_{k'}) = W_{k'} \cup \{T_{k'}\} \cup W'$.

The rearrangement inequality states that if x_1, x_2, y are all nonnegative numbers then $|x_1 + y|^p + |x_2|^p \leq |x_1|^p + |x_2 + y|^p$ if and only if $x_1 \leq x_2$. We apply the rearrangement inequality where:

$$x_1 = A(T_{k'}) + U_{S_{k-1}}(T_{k'}) + \sum_{T_j \in W_{k'}} R_{D_1}(T_j)$$

$$x_2 = A(T_k) + U_{S_{k-1}}(T_k) + \sum_{T_j \in W_k} R_{D_2}(T_j)$$

$$y = R_{D_1}(T_{k'}) + \sum_{T_j \in W'} R(T_j) = R_{D_2}(T_k) + \sum_{T_j \in W'} R(T_j).$$

The age of $T_{k'}$ when it is scheduled in S_{k-1} is $x_1 - R_{D_1}(T_{k'})$, and the age of T_k when $T_{k'}$ is scheduled in S_{k-1} is $x_2 - R_{D_2}(T_k)$. Since, at the time $T_{k'}$ is scheduled in S_{k-1}, T_k is older than $T_{k'}$, and since $R_{D_1}(T_{k'}) = R_{D_2}(T_k)$, we have that $x_1 < x_2$.

The theorem follows by noting that in the S_{k-1} schedule, the $T_{k'}$ term is x_1 and the T_k term is $x_2 + y$; while in the S_k schedule, the $T_{k'}$ term is $x_1 + y$ and the T_k term is x_2. □

In practice, we observe that $R(T)$ has a near-zero correlation with $A(T)$ (Appendix C.2). Thus, the I.I.D. assumption of Theorem 1 seems plausible. Interestingly, even if the variance of the execution times were 0 (i.e., a correlation of -1), our theorem would be even more true, as not only would VATS gain by avoiding losses from old transactions, but it would also gain because such transactions would complete and release their locks faster.

In our implementation, we slightly modify VATS such that it grants as many locks as possible if a lock does not conflict with any of the locks in front of it in the queue (including both the granted locks and the ones still waiting), which is preserved in an eldest-first order, as a means to improve performance. We evaluate VATS in Section 7.2.

6. ADDITIONAL STRATEGIES

Based on our findings from Section 4, we present further strategies for improving performance predictability. Unlike

[9]To be more precise, what happens after transactions are granted a lock may depend on our schedule itself, as similar transactions could interact in the future on other queues. For simplicity, in this discussion we ignore this complication.

Inputs : p: the page to be moved to start of the LRU list;
b: the buffer pool

```
1 buf_pool_mutex_enter(b);
2 buf_LRU_make_block_young(b, p);
3 buf_pool_mutex_exit(b);
```

Algorithm 1: How the LRU list is updated in MySQL

our VATS algorithm which is a generic way of reducing variance in wait times, our techniques in this section are specific to MySQL, Postgres, and VoltDB. We use these techniques to illustrate TProfiler's effectiveness in localizing the sources of variance in a massive and complex codebase (e.g., MySQL or Postgres). TProfiler enables us to drastically reduce overall variance with minimal modification (ranging from changing tuning parameters to a few hundred lines of code) by examining only a handful of functions out of tens of thousands (see Section 7).

6.1 Lazy LRU Update (LLU)

As noted in Section 4.1, the lock on the LRU list is a main source of variance in MySQL when the working set exceeds the buffer pool size. Algorithm 1 shows how the LRU list is updated in MySQL. First, a mutex is acquired by calling `buf_pool_mutex_enter`, and then the page is moved to the head of the list by calling `buf_page_make_young`.

For better cache performance, MySQL does not implement the strict LRU policy. Instead, it splits the LRU list into two sublists, *young* and *old*. Replacement victims are selected from the old list, which by default contains 3/8 of the oldest pages. When a page is accessed, if it is currently in the old list, it is moved to the head of the young list, and the tail of the young list is placed at the head of the old list. To avoid frequent re-ordering of the list, MySQL does not maintain precise LRU ordering within the young list. However, when the working set exceeds 5/8 of the buffer pool, old pages are accessed frequently, and the lock on the LRU list becomes a bottleneck. Our idea is to further relax the precision of LRU tracking to avoid this contention, as described next.

To avoid excessive delays, our proposed algorithm, Lazy LRU Update (LLU), limits the time that `buf_pool_mutex_enter` waits for the lock. Specifically, we replace the mutex with a spin lock to control the wait time. When the buffer pool is sufficiently large, this lock is typically uncontended, and the overhead of a spin lock remains minimal. However, if a waiting thread cannot acquire the lock within 0.01ms, we abandon the attempt to update the global list and instead add the page to a thread-local backlog of deferred LRU updates, l. Later, when `buf_pool_mutex_enter` successfully acquires the lock for a different page, we first process the pages in l (after confirming that they have not been evicted) before moving the page that triggered the reordering.

6.2 Parallel Logging

As revealed by TProfiler in Section 4.2, over 70% of latency variance in Postgres is due to the variation of wait times in redo log flush operations. This leads us to another strategy for improving predictability: parallel logging, so that when a log file is unavailable, a transaction can write to other log files instead of having to wait. While there are sophisticated parallel logging schemes [14, 15, 69], we implement a simple variant that allows Postgres to use two hard disks for storing two sets of redo logs. A transaction only waits when neither of these sets is available, in which case it waits for the one with fewer waiters. Though parallel logging is well-studied for improving mean latencies, we vet its effectiveness in reducing latency variance in Section 7.4.

6.3 Variance-Aware Tuning

In many cases, the behavior of the culprit function identified by TProfiler can be controlled through external tuning parameters of the DBMS. Specifically, (i) in MySQL, `buf_pool_mutex_enter()` leads us to buffer pool size while `fil_flush()` leads us to `innodb_flush_log_at_trx_commit` parameter, (ii) in Postgres, `LWLockAcquireOrWait()` leads us to I/O block size, and (iii) in VoltDB, the queuing delay leads us to the number of worker threads. Interested readers are referred to Appendix B for discussion and empirical study of these parameters' impact on latency variance.

7. EXPERIMENTS

Our experiments aim to answer three key questions: (1) How effective are our techniques (VATS, LLU, parallel logging, and variance-aware tuning) in reducing tail latency and latency variance? (2) Does our reduction of latency variance come at the cost of sacrificing mean latency or throughput? (3) How effective and efficient is TProfiler compared to other profiling alternatives? In summary, our results indicate that:

- For contended workloads (TPC-C, SEATS, and TATP), our VATS algorithm makes the DBMS significantly more predictable (and even faster) without compromising throughput, with up to 6.3x, 5.6x, and 2.0x lower mean, variance, and 99th percentile latencies, respectively. As expected, for non-contended workloads (Epinions and YCSB), the choice of scheduling algorithm is immaterial. (Section 7.2)

- Our Lazy LRU Update algorithm makes MySQL faster and more predictable, with 1.4x, 1.2x, and 1.2x lower mean, variance, and 99th percentile latencies, respectively. (Section 7.3)

- Parallel logging improves both predictability and overall performance of Postgres, with 1.8x, 1.3x, and 2.4x lower mean, variance, and 99th percentile latencies, respectively. Also, variance-aware tuning can eliminate up to 88.3% of the overall latency variance. (Sections 7.4 and 7.5)

- TProfiler's profiling overhead is an order of magnitude lower than that of DTrace, and its factor selection algorithm reduces the number of required runs by several orders of magnitude compared to a naïve strategy. (Section 7.6)

These results are summarized in Table 3. We have also included additional experiments in Appendix C, showing that existing DBMSs are highly unpredictable even when running the same transactions (C.1) and there is no correlation between a transaction's remaining time and its age (C.2).

7.1 Experimental Setup

The hardware and software used for our experiments in this section are the same as Section 4. For fairness, we used the same throughput of 500 transactions per second across all workloads and algorithms. Also, to rule out the effect of external load changes on latency variance, we used

System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	os_event_wait	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	buf_pool_mutex_enter	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	fil_flush	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	LWLockAcquireOrWait	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	[waiting in queue]	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

Table 3: Impact of modifying each of the functions identified by TProfiler. The last 3 columns compare end-to-end transaction latencies before and after each modification. For example, modifying os_event_wait eliminates more than 82% of MySQL's total latency variance, i.e., the ratio of the transaction variance of original MySQL to modified MySQL is $1/(1-0.82)=5.56$.

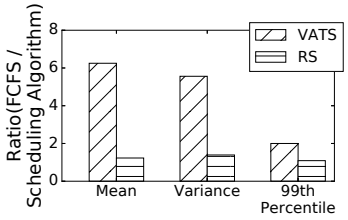

Figure 2: Effect of different scheduling algorithms on MySQL performance. For example, replacing FCFS with VATS makes MySQL 6.3x faster and 5.6x lower in variance.

the OLTP-Bench [32] tool to sustain a constant throughput throughout the experiment, and measured mean, variance, and 99th percentile latencies for each algorithm and workload. In addition to TPC-C, we also used the following workloads for a more extensive evaluation:

- **SEATS [62]:** This benchmark is a simulation of an airline ticketing system where customers search flights and make online reservations. In our experiments, we used a scale factor of 50, leading to a highly contended workload.
- **TATP [68]:** TATP models a typical caller location system used by tele-communication providers. For TATP, we used a scale factor of 10, making it a contended workload (but not as contended as TPC-C).
- **Epinions [48]:** Epinions simulates a customer review website where users interact and write reviews for various products. We used a scale factor of 500 in our experiments. This workload has a very low contention.
- **YCSB [30]:** YCSB is a set of micro-benchmarks simulating data management applications that have simple workloads but require high scalability. The scale factor used was 1200, causing little or no contention.

Given that varying lock wait times are a major problem for MySQL, we evaluate VATS using MySQL. We evaluate LLU and parallel logging using MySQL and Postgres, respectively. Finally, we study variance-aware tuning for all three: MySQL, Postgres and VoltDB. When results are similar across all workloads, we only report numbers for TPC-C as a representative workload.

7.2 Studying Different Scheduling Algorithms

We compare VATS to two other scheduling algorithms:

- **First Come First Served (FCFS):** This is the default scheduling in many DBMSs (including MySQL & Postgres).
- **Randomized Scheduling (RS):** Similar to VATS, except that transactions are sorted according to a random order rather than by age.

	Workload	Mean Latency	Variance	99th Percentile
Contended	TPCC	6.3x	5.6x	2.0x
	SEATS	1.1x	1.3x	1.1x
	TATP	1.2x	1.6x	1.3x
	Avg	**2.9x**	**2.8x**	**1.5x**
No Contention	Epinions	1.4x	2.6x	1.0x
	YCSB	1.0x	1.1x	1.1x

Table 4: Comparing VATS with MySQL's original (FCFS) lock scheduling in terms of overall transaction latency.

The results are shown in Figure 2 for TPC-C (see Table 4 for other workloads). In summary, FCFS is the least efficient scheduling algorithm for all three contended workloads. For example, for TATP, even a random scheduling (RS) improves upon FCFS by 25% in terms of latency variance. However, the randomness of RS can also be harmful. For SEATS, RS performs about 2 orders of magnitude worse than other algorithms (results omitted for space). The choice of lock scheduling algorithm does not make a difference for YCSB simply because it does not have any lock contention. In case of Epinions, the improvement is due to the fact that we place newly-granted locks at the head of the list, and thus the time for traversing the list is reduced (MySQL uses a global hash table where each bucket is a linked list storing some of the lock objects).

We have summarized VATS's improvement over FCFS in Table 4 for all workloads. Our VATS algorithm is consistently superior for contended workloads and comparable to no-contention ones. Most notably, VATS eliminates 84.1% of the entire latency variance of MySQL for TPC-C. In other words, replacing FCFS with VATS makes MySQL's latency variance 6.3x lower. On average, this number is 2.9x for all contended workloads, and 2.4x over all five workloads.

7.3 Lazy LRU Update Algorithm

In this section, we evaluate our Lazy LRU Update (LLU) algorithm. We produce a memory-contended workload using the same 2-WH configuration from Section 4.1. As shown in Figure 3(left), LLU yields a more predictable (and even slightly faster) MySQL with 1.1x, 1.6x, and 1.4x lower mean, variance, and 99th percentile latencies. This improvement is because LLU avoids extremely long waits, delaying the re-ordering of buffer pages until the overhead is fairly low. This reduces the contention on the LRU data structure for memory-contended workloads.

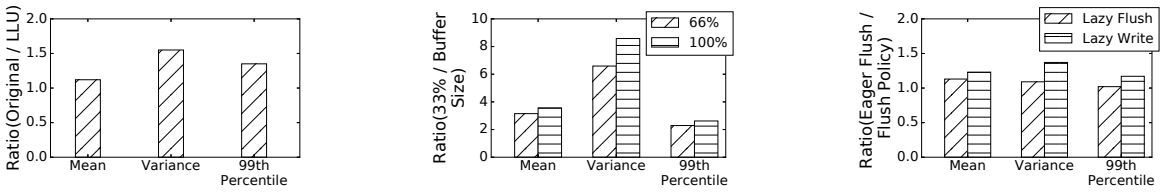

Figure 3: Effect of LLU, buffer pool size (in % of the entire database size), and log flush policy on MySQL (TPC-C).

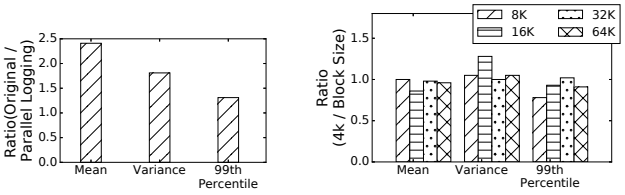

Figure 4: Effect of parallel logging and redo log block size on Postgres (TPC-C).

7.4 Parallel Logging

As discussed in Section 6.2, we implement a parallel logging scheme for Postgres. As shown in Figure 4(left), this significantly reduces mean, variance, and 99th percentile latencies, lowering them by 2.4x, 1.8x, and 1.3x, respectively.

7.5 Variance-Aware Tuning

In Section 6.3, we identified several tuning parameters in MySQL, Postgres, and VoltDB that affect latency variance.

We first investigate the buffer pool size for MySQL and TPC-C, as shown in Figure 3(center). We set the buffer pool size to 33%, 66%, and 100% of the overall database size, and report their relative performance compared to 33%. As expected, a larger buffer pool retains more data in memory, thus effectively reducing the number of page evictions, the number of I/O operations, and the degree of contention within the buffer pool. Consequently, the larger the buffer pool, the lower the mean, variance, and 99th percentile latencies. Ideally, a buffer pool as large as the entire database is recommended for both better average performance and greater predictability. However, depending on the working set size, a smaller buffer pool might be economically more appealing, while producing comparable results.

Second, we investigate MySQL's log flushing policies, as shown in Figure 3(right). The results indicate that deferring both write and flush operations to a log flusher thread minimizes transaction variances. This is not surprising: eagerly flushing logs prior to commit places highly variable disk write latencies on the transaction execution path. However, lazy flushing may lose forward progress (committed transactions) in the event of a crash.

In Postgres, another strategy for reducing the variance of redo log flushes is to accelerate the I/O operations by tuning an appropriate block size (see Section 6.3), which is by default 8 KB. Figure 4(right) shows that increasing the block size can reduce variance, but only to a certain extent. A larger block can reduce the number of write operations per transaction, but when it becomes so large that the generated log records only occupy a small portion of a block, the transaction still has to write the whole block. In such cases, the disadvantage of writing more data than needed outweighs the advantage of fewer writes.

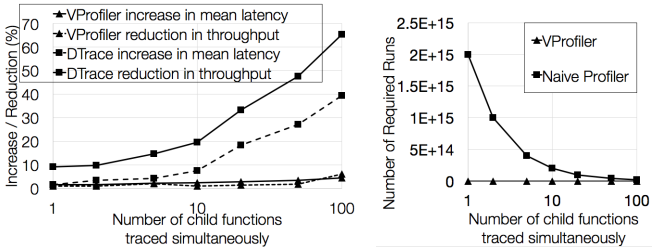

Figure 5: (Left) Profiling overhead of TProfiler vs. DTrace. (Right) Number of runs needed for the profiler to identify the main sources of variance.

Finally, we explore the effect of number of worker threads on VoltDB's performance. In a nutshell, adjusting this parameter in VoltDB can eliminate 60.9% of the total latency variance, i.e., lower it by 2.6x (see Appendix A for details).

7.6 Evaluation of TProfiler

In previous sections, we validated TProfiler's *effectiveness* by showing that our algorithmic and tuning changes, which were informed by TProfiler's findings, indeed reduce latency variance. In this section, we evaluate (i) TProfiler's performance *overhead* in measuring the execution time variance of a function, and (ii) its *efficiency* in narrowing down the search for main sources of variance.

TProfiler vs. DTrace — By instrumenting a DBMS code, TProfiler incurs a performance overhead. To quantify this overhead, we vary the number of children functions that need to be instrumented from 1 to 100, and measure both the relative drop of throughput and the relative increase in average latency. The results are shown in Figure 5(left). As a baseline, we also report the same types of overhead using DTrace, a programmable profiler for troubleshooting arbitrary software. Similar to TProfiler, one can use DTrace to measure the execution time of a parent function and its children, and then compute variances using eq. (1).

DTrace's key advantage is that, unlike TProfiler, it instruments the binary code and does not need the source code. However, this flexibility comes at a cost in the performance of the profiling code. As shown in Figure 5(left), DTrace's overhead (on both latency and throughput) is significantly higher than TProfiler, and grows rapidly with the number of traced children, whereas TProfiler's overhead stays below 6%. This is expected as DTrace must use heavy-weight mechanisms to inject generalized instrumentation code at run-time, while TProfiler inserts minimal profiling code prior to compilation of the source.

TProfiler vs. Naïve Profiling — We also compare against a naïve profiling strategy, which is similar to TProfiler, except that it decomposes every factor rather than only a few im-

portant ones. In total, there are 2×10^{15} nodes in MySQL's static call graph, 4.5×10^{14} of which are leaves. A naïve profiler has to break down every non-leaf, and thus the number of runs needed is extremely large. TProfiler's selection strategy needs significantly fewer runs to locate the main sources of variance, as confirmed in Figure 5(right).

8. RELATED WORK

Predictable Query Plans — There has been some pioneering work on enriching query optimizers to account for parameter uncertainties (caused by cardinality and cost estimates) when choosing a query plan [26, 29].

Florescu and Kossman [34] have taken the opposite direction by arguing for a radical DBMS redesign. They propose a new tiered architecture for web applications, where consistency maintenance is moved from DBMS to application layer. Others have advocated the use of table scans for all queries [19, 40, 58, 59, 65], or simply restricted themselves to query plans with a bounded worst-case [16, 17]. Although many of these techniques share scans and joins across multiple queries, and use always-on operators to reduce execution time [19, 25, 35, 40, 58, 65], they still have a negative impact on average latency. As such, these solutions are more appropriate for long-running decision support queries than transaction processing. Thus, while successfully adopted by OLAP vendors [25, 58], these proposals have not had widespread adoption by major OLTP vendors, as foregoing low latency to achieve predictability is an unattractive trade-off for many latency-critical and transactional applications (see 'Desirable Solutions' in Section 2).

Instead of requiring richer statistics or dismissing traditional query optimizers altogether, we take a top-down approach (see Section 1 for the distinction) by carefully studying the entire source code of existing database systems to quantify and mitigate their root causes of performance variance. Moreover, we seek practical solutions that reduce variance without sacrificing mean latency, a decision that has helped the real-world adoption of our proposal (Section 9).

Real-Time Databases — Once an active area of research in the 1990s, real-time databases (RTDBs) [13, 43, 55] sought real-time performance guarantees by (i) requiring each transaction to provide its own deadline, and (ii) minimizing deadline violations by restricting themselves to mechanisms that bounded worst case execution times. In contrast, we study predictability in the context of today's conventional best-effort transaction processing systems, where sacrificing throughput or mean latency to obtain hard bounds on execution time may not be an appealing trade-off.

Variance-Aware Job Scheduling — Outside a database context, theoretical literature has examined the problem of scheduling general tasks to minimize completion time variance (CTV) and waiting time variance (WTV). These formulations assume a set of jobs with *known processing times* and seek a schedule that minimizes the variance of their completion or wait times. While CTV and WTV problems are both NP-complete [22, 44], there are several heuristics [27, 33, 70], dynamic programming solutions [31, 45], and polynomial-time approximations [46]. These techniques assume an *offline* setting, and thus do not apply to our transaction scheduling problem, since the remaining and arrival time of transactions are unknown in practice. In contrast, our VATS algorithm does not require such knowledge.

Profiling Literature — There is a large body of work on profiling techniques [23, 36, 38, 61, 64]. In a nutshell, TProfiler is the first profiler to systematically break down the contribution of individual functions to the overall latency variance and, with minimal help from programmers, distinguish execution times that are relevant to transaction latencies.

Performance Diagnosis and Prediction — There are several tools that help users diagnose performance anomalies or reproduce intermittent bugs, either by monitoring fine-grained, low-level OS events [24] or by collecting statistics from the application and the OS for post-mortem analysis [20, 72]. In contrast, we focus on finding the internal causes of performance variance by instrumenting the application code and relying on the mathematical definition of variance to narrow its search space.

Instead of profiling transactions, there is also some work on passively predicting the performance of transactions using machine learning techniques [49, 50, 71]. By reducing the performance variance, our work should ultimately make performance predictions easier.

9. REAL-WORLD ADOPTION

After observing the considerable impact of our small modifications on performance predictability (Table 3), we decided to share these results with the open-source community. In particular, our VATS algorithm was quickly adopted by MySQL distributions, and has even been made a default policy by MariaDB [3]. These MySQL distributions comprise over 2M+ installations around the world.

Meanwhile, the issue with LRU mutex contention found by TProfiler was independently identified by the MySQL community and addressed by multi-threaded flushing [8, 11] and other techniques [6, 10]. While their solution differs from our LLU technique, they still confirm the validity of TProfiler's finding regarding the cause of the performance pathology.

10. CONCLUSION

We presented a novel profiler, called TProfiler, 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, TProfiler makes it possible to calculate the contribution of each function to the overall variance. Using TProfiler, we analyzed the codebases of three complex database systems, leading us to small modifications that significantly reduced performance variance in these popular databases.

11. ACKNOWLEDGEMENT

This work is in part supported by National Science Foundation through grants 1544844, 1629397, 1553169 and 1535912. The authors would like to express their gratitude to Mark Callaghan, Jan Lindström, and Laurynas Biveinis for their help and guidance in integrating VATSinto MySQl and MariaDB, to Jarrid Rector-Brooks for his contributions to TProfiler's implementation, and to Morgan Lovay and Shannon Riggins for their insightful comments on this manuscript.

Bibliography

[1] 15.5.2 InnoDB Transaction Model. http://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-model.html.

[2] Google Cloud SQL. http://code.google.com/apis/sql.

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

[4] Oracle database cloud service. http://cloud.oracle.com.

[5] Program2 Two-Phase Locking (2PL) vs. Timestamp Ordering (TSO) vs. A Real Protocol. http://cobweb.cs.uga.edu/~shasha/course/csci8370/prog2/prog2.pdf.

[6] 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.

[7] SQL Server Locking and You! https://www.brentozar.com/archive/2011/06/sql-server-locking/, 2011.

[8] MT LRU flusher. https://blueprints.launchpad.net/percona-server/+spec/mt-lru, 2016.

[9] MySQL Transaction Lock Manager Source Code. https://github.com/mysql/mysql-server/blob/5.7/storage/innobase/lock/lock0lock.cc, 2016.

[10] Parallel doublewrite buffer. https://blueprints.launchpad.net/percona-server/+spec/parallel-doublewrite, 2016.

[11] 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.

[12] Postgre Transaction Lock Manager Source Code. https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/proc.c, 2016.

[13] R. K. Abbott et al. Scheduling real-time transactions: A performance evaluation. *TODS*, 1992.

[14] R. Agrawal. A parallel logging algorithm for multiprocessor database machines. In *Database Machines*. 1985.

[15] R. Agrawal and D. J. DeWitt. *Recovery architectures for multiprocessor database machines*. ACM, 1985.

[16] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5, 2011.

[17] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, 2013.

[18] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage; recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[19] 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 *SIGMOD*, 2010.

[20] Attariyan et al. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[21] P. D. Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, UC Berkeley, 2015.

[22] Bector et al. V-shape property of optimal sequence of jobs about a common due date on a single machine. *Computers & operations research*, 1989.

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

[24] Bhatia et al. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI*, 2008.

[25] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2009.

[26] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, 2010.

[27] Chen et al. Sequencing heuristic for bicriteria scheduling in a single machine problem. *Journal of Information and Optimization Sciences*, 2006.

[28] S. Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD*, 2009.

[29] F. Chu et al. Least expected cost query optimization: An exercise in utility. In *PODS*, 1999.

[30] B. F. Cooper et al. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[31] P. De et al. On the minimization of completion time variance with a bicriteria extension. *Operations Research*, 1992.

[32] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7, 2013.

[33] S. Eilon and I. Chowdhury. Minimising waiting time variance in the single machine problem. *Management Science*, 1977.

[34] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *Sigmod Record*, 2009.

[35] G. Giannikis et al. Shareddb: killing one thousand queries with one stone. *PVLDB*, 2012.

[36] S. L. Graham et al. Gprof: A call graph execution profiler. In *Sigplan Notices*, 1982.

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

[38] R. J. Hall. Call path profiling. In *ICSE*, 1992.

[39] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[40] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[41] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *HPTS*. 1989.

[42] J. Huang, B. Mozafari, and T. Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[43] Y.-K. Kim and S. H. Son. Supporting predictability in real-time database systems. In *RTAS*, 1996.

[44] W. Kubiak. Completion time variance minimization on a single machine is difficult. *Operations Research Letters*, 1993.

[45] W. Kubiak. New results on the completion time variance minimization. *Discrete Applied Mathematics*, 1995.

[46] W. Kubiak et al. Fast fully polynomial approximation schemes for minimizing completion time variance. *Eur. Journal of Operational Research*, 2002.

[47] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 2013.

[48] P. Massa and P. Avesani. An experimental study on epinions.com community. In *NCAI*, 2005.

[49] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.
[50] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.
[51] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.
[52] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: An extended report. Technical report, University of Michigan, Ann Arbor, 2015.
[53] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *PVLDB*, 2015.
[54] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.
[55] P. O'Neil et al. A two-phase approach to predictably scheduling real-time transactions., 1996.
[56] S. Pelley et al. Storage management in the nvram era. *PVLDB*, 2013.
[57] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer Science, 2012.
[58] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 2008.
[59] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.
[60] M. Sadoghi et al. Making updates disk-i/o friendly using ssds. *PVLDB*, 2013.
[61] J. M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 2004.
[62] M. Stonebraker and A. Pavlo. The seats airline ticketing systems benchmark.
[63] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *TODS*, 1985.
[64] Z. Szebenyi et al. Space-efficient time-series call-path profiling of parallel applications. In *SC09*, 2009.
[65] P. Unterbrunner et al. Predictable performance for unpredictable workloads. *PVLDB*, 2009.
[66] M. Wainwright. Chapter 2: Basic tail and concentration bounds. http://www.stat.berkeley.edu/~mjwain/stat210b/Chap2_TailBounds_Jan22_2015.pdf.
[67] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 2014.
[68] A. Wolski. Tatp benchmark description, 2009.
[69] R. J. Yang et al. PTL: Partitioned logging for database storage on flash solid state drives. In *WAIM*. 2012.
[70] N. Ye, X. Li, T. Farley, and X. Xu. Job scheduling methods for reducing waiting time variance. *Computers & Operations Research*, 2007.
[71] D. Y. Yoon, B. Mozafari, and D. P. Brown. DBSeer: Pain-free database administration through workload intelligence. *PVLDB*, 2015.
[72] D. Y. Yoon, N. Niu, and B. Mozafari. DBSherlock: A performance diagnostic tool for transactional databases. In *SIGMOD*, 2016.
[73] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 2014.

APPENDIX

A. CASE STUDY: VOLTDB

We also used TProfiler on VoltDB's source code to identify its major sources of variance. VoltDB is an event-based system in which transactions are wrapped up as stored procedure invocations. Each event needs to wait in a queue before a worker thread is available to process it. TProfiler reports that almost 99.9% of latency variance in VoltDB is due to the variance in the waiting time of these events in different queues. This finding leads to the number of worker threads as a tuning parameter to control the queue size. As shown in Figure 7, adjusting this parameter in VoltDB can eliminate 60.9% of the total latency variance, reducing it by 2.6x.

B. VARIANCE-AWARE TUNING

As mentioned in Section 6.3, the behavior of some of the functions identified by TProfiler can be influenced through external tuning parameters of the DBMS.

First, from our investigation of `buf_pool_mutex_enter` (Section 4), we learned that buffer pool capacity (relative to the database working set) substantially impacts variance (and, of course, mean latency). Hence, we sweep buffer pool capacity from 33% to 100% of the overall database size and measure the impact on transaction variance.

Second, we learned that MySQL's policy regarding log flushing has a noticeable influence on transaction variance (Section 4). MySQL's use of buffered I/O for redo logs involves two steps: a `write` system call, and a `flush` system call. MySQL offers three policies that can be chosen through the `innodb_flush_log_at_trx_commit` parameter:

- **Eager flush:** This requires that redo logs are written and flushed by the transaction worker thread before committing the transactions.

- **Lazy flush:** Under this setting, redo logs are written by the transaction worker thread, but flush operations are deferred to a separate log flusher thread, which invokes the flush system call roughly once per second. Transactions may commit before their logs are flushed.

- **Lazy Write:** Under this setting, redo logs are prepared but not written by the transaction worker thread. Both writing and flushing the log are deferred to a log flusher thread which performs these operations once per second. Transactions may commit before their logs are written.

Note that both lazy flush and lazy write risk losing forward progress in the event of a crash; transactions executed in the previous second may be reported as committed to the user, but may be unrecoverable because their redo logs never became durable. Nevertheless, in contexts where forward progress loss can be tolerated, employing lazy flushing and writes can substantially improve the latency and predictability of transaction execution.

We also observed that much of the latency variance in Postgres is due to varying wait times of transactions when flushing their redo logs (Section 4.2). This I/O operation can be accelerated by tuning Postgres's block size parameter, which is 8KB by default. (Another solution is to use parallel logging; see Section 6.2.)

Finally, TProfiler reveals that queuing delay is the single reason for latency variance in VoltDB. Therefore, to reduce

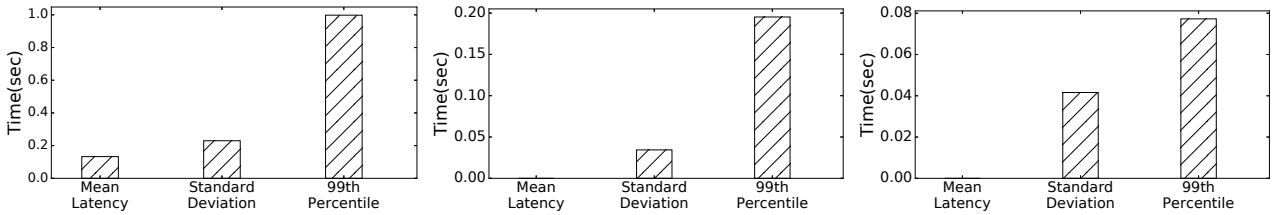

Figure 6: Mean, standard deviation, and 99th percentile latencies in MySQL (left), Postgres (center), and VoltDB (right).

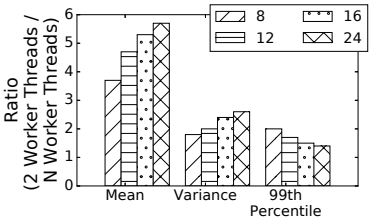

Figure 7: Effect of different numbers of worker threads on VoltDB's performance (2 is the default value).

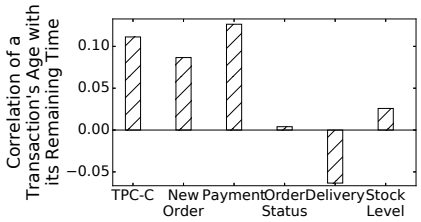

Figure 8: Correlation between a transaction's age and its remaining time for different transaction types (TPC-C).

the queue size, one can increase the number of VoltDB's worker threads (see Section A).

C. ADDITIONAL EXPERIMENTS

C.1 Performance Variance in Existing DBMSs

To quantify the extent of performance variance in out-of-the-box DBMSs, we measured their mean, standard deviation, and 99th percentile latencies for TPC-C using the same setup as Section 4. As shown in Figure 6, all three engines exhibited incredible degrees of performance disparity; the standard deviation was twice the mean (1.7x for MySQL, 1.9x for Postgres, and 3.3x for VoltDB) and the 99th percentile was an order of magnitude greater than the mean (7.5x for MySQL, 11.0x for Postgres, and 6.1x for VoltDB).

To rule out the inherent effect of running different transaction types in TPC-C on overall variance, we modified our workload to only issue New Order transactions. We even modified the New Order transactions to always issue a fixed number of queries (as opposed to a random number between 25 and 65, which is their default setting). However, even under this pure workload running at a constant rate, the ratios of standard deviation and 99th percentile to mean latencies remained similar.

C.2 Correlaction of Transaction Age and Remaining Time

One might imagine that the larger a transaction age, the smaller its remaining time. Interestingly, this is not the case in practice due to the intertwined nature of contended transactions. Figure 8 shows the correlation between a transaction's age and its remaining time at the moment when scheduling decisions are made. As shown in Figure 8, the correlation of these two values is quite small regardless of the transaction type, indicating the difficulty in predicting the remaining time of a transaction given its age.