Identifying the Major Sources of Variance in Transaction Latencies: Towards More Predictable Databases

Jiamin Huang
jiamin@umich.edu

Barzan Mozafari
mozafari@umich.edu

ABSTRACT

Transaction processing is an important part of modern database management systems(DBMSs), so a lot of work has been done to improve the performance of it. However, few people pay attention to the predictability of performance, which is fundamental to many application of database systems such as in-cloud database services that provide performance guarantee. The service providers will not be able to provide such kind of guarantees unless their services are based on database systems with predictable performance.

In this paper, we show that variance of transaction latency is a very severe problem in traditional database systems like MySQL. We propose a profiling framework called VProfiler that uses the source code of a database system to identify the main sources of variance in transaction latency. This framework works by breaking down the variance of latency into variances and covariances of the execution time of the functions in the source code, which is the best source of information about the actual cause of variance. Using MySQL as an example, we show its main sources of variance found using our profiling framework, and propose two methods to reduce the variance of latency after close investigation of these sources. Experiment results show that these two methods, when combined, achieve a 42.3% reduction in 99th percentile. Besides these methods, we also demonstrate how using faster storage devices and tuning the configuration of MySQL can further improve its performance predictability.

1. INTRODUCTION

Transactional databases are a key component of almost every enterprise software, where mission-critical applications rely on the underlying DBMS to store and manipulate data efficiently and reliably. Consequently, a significant portion of database research on transactions has focused on reducing latency and increasing throughput, e.g., through concurrency control and recovery protocols, query optimization techniques, indexing schemes, caching policies, and many other sophisticated ideas. These strategies, however, have been mostly evaluated in terms of their effect on the *average performance* of a transactional database, such as its throughput or mean latency in executing transactions. In other words, the focus has often been on running more and/or faster transactions *overall*. The effect of these strategies on the spread of the latency distribution has largely remained unvetted. Have our traditional database design principles sacrificed the tail latencies for the sake of improving average performance of transactions? Do our existing databases exhibit a notable latency variance in practice? If so, how much of this variance is due to our database implementation and how much of it is due to the the variations in the user's queries themselves? How can we measure the contribution of each component/algorithm of our database to the overall latency variance? And most importantly, do we need to compromise on the mean latency to reduce its variance or 99% quantile, and if so, how?

We believe that it is critical and particularly timely to ask these questions. First, the past four decades of research on transactions has matured to a point where microsecond latencies and hundreds of thousands of concurrent transactions are now achievable [**?**]. Second, database vendors are facing an increasing number of business-oriented clients and applications that demand quality of service guarantees (QoS).[1] Moreover, with the increasing market-share of database-as-a-service (DBaaS) offerings, cloud providers and users rely on service level agreements (SLAs) for pricing and provisioning, respectively. Finally, as they deliver a wide range of complex features to a wide range of applications, DBMSs have (understandably) become one of the most complex breed of software systems themselves. Thus, predicting query latencies in a database has been reportedly one of the most challenging tasks [**?**]. Understanding the major sources of variance in the execution time of transactions might provide invaluable insight towards designing a new generation of database systems that can deliver competitive performance but be much more predictable.[2] The benefits of a predictable database will be many, e.g., automatic provisioning tools, more accurate cost estimates (and hence, better query scheduling and planning decisions), and more reliable query progress estimators. Identifying the major sources of variance

Database Management Systems have become an essential part of almost every large software system anyone can find today. Transaction processing is one of the most important features provided by these database management systems, which allows users to group a bunch of operations into one single logical operation. A lot of work has been done on the performance of transaction processing. Predictability of performance, which is just as important as performance itself, remains neglected by researchers. An unpre-

[1]Based on oral communications with the chief architects, researchers and engineers at Teradata, HP Vertica, and Microsoft.

[2]While achieving predictable performance is also desirable for analytical workloads, we leave this to future work and only focus on transactional workloads in this paper.

dictable database management system is a potential cause of financial loss regardless of how good the overall performance is. Due to the importance of predictability, many enterprise companies are willing to sacrifice 10% to 30% of the overall performance for a more predictable database. Predictability of performance is an essential requirement for cloud-based offerings. For example, SQL Database of Microsoft Azure, a relational database-as-a-service offering, provides predictable performance guarantees in terms of Database Throughput Unit(DTU), transaction rate and consistency of response time. On the other hand, the Service Level Agreement(SLA) offered by Amazon's Relation Database Service(RDS) is only in terms of Monthly Uptime Percentage, namely availability. For disks, it offers performance guarantee only in terms of IOPS. No guarantees are provided in terms of more interesting factors, such as transaction latency(which is what the users really care about), and they will never exist unless the provides base their database services on database management systems whose performance is predictable.

We did experiments on MySQL to evaluate the predictability of their performance on transaction processing. MySQL is among the most widely used relational database management systems(RDBMS) and most popular open-source RDBMS. It provides not just one, but several storage engines for users to choose from, among which Innodb is the default and most commonly used. Innodb is a storage engine that provides standard ACID-compliant transaction features, along with foreign key support. This paper is based on the Innodb engine.

We used Amazon EC2 m3.large instances with traditional HDDs in our experiments, and chose TPC-C as our benchmark. TPC-C is a famous Online Transaction Processing(OLTP) benchmark. Queries in an OLTP system are mostly simple and standardized with relatively small number of records returned. Unlike OLTP, Online Analysis Processing(OLAP) systems are usually complex because of the aggregations involved. Queries in OLTP systems are usually generated using query templates. The following is a query template in the TPC-C Benchmark to acquire data of an item from the database.

```
SELECT I_PRICE, I_NAME , I_DATA
FROM ITEM
WHERE I_ID = ACTUAL_ID
```

When this query is executed, ACTUAL_ID would be replaced by an actual value of an I_ID, which could be different each time. Transactions in OLTP systems are usually composed of a bunch of ordered query templates, and can be categorized into different types according to the query templates they consist of. TPC-C has five types of transactions: New Order, Payment, Order Status, Delivery and Stock Level. The original TPC-C benchmark requires 45% of New Order transactions, 43% of Payment transactions, 4% of Order Status transactions, 4% of Delivery transactions and 4% of Stock Level transactions. Transactions of the same type can also have different numbers of queries(in a fixed range). These are all sources of variance in transaction latency. Therefore, besides the original TPC-C benchmark, we also have experiments that use only New Order transactions, and experiments that use New Order transactions with fixed number of queries. We measure the average, standard deviation and 99th percentile latency and use the ratio of standard deviation to mean and the ratio of 99th percentile to mean to evaluate the predictability of their performance.

As can be seen from figure 2, the standard deviation of transaction latencies is almost 3 times of the average latency. On the other hand, 99th percentile is a lot worse than standard deviation because

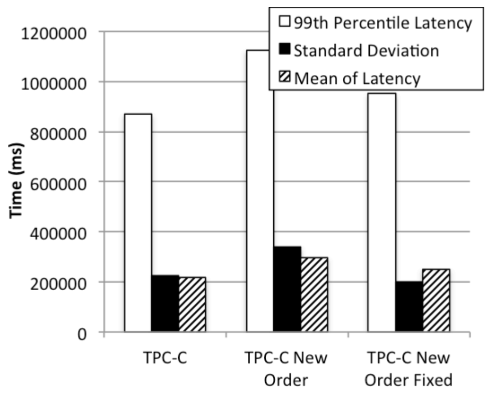


Figure 1: 99th percentile, standard deviation and average latency in MySQL

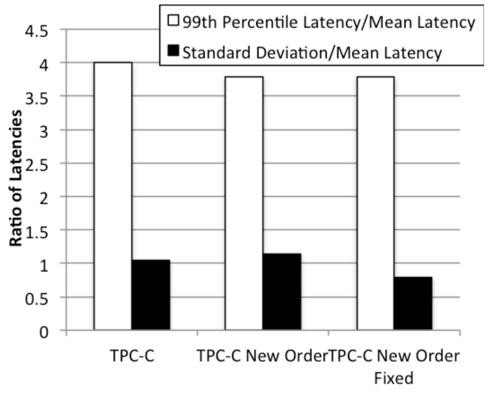


Figure 2: 99th percentile and standard deviation are several folds larger than the average latency in MySQL

no matter which benchmark we use, the 99th percentile latency is an order of magnitude larger then the average latency.

2. RELATED WORK

Although relatively ignored compared to performance of transaction processing, performance predictability has received some significant interest from the research community. Studies have been done regarding query progression indication and query performance prediction, architecture redesign and predictability improvement.

Query Progress Indication and Query Performance Prediction There has been significant work [3, 4, 11, 12, 13] about query progress indicators, which are used to estimate the progress of long-running queries. In [3, 4, 11, 12], a set of techniques are proposed to keep track of how much of a query has been completed and continuously estimate the remaining query execution time. These query progress indicator, however, are single-query indicators, which means that they only take the current workload and the estimated query into consideration, ignoring the impact from all other currently running queries. A query indicator that handles the performance impact of current queries is proposed in [13], which even considers the influence of queries expected to arrive in the future.

There are also work [1, 6, 10, 8] about predicting the performance of queries that are not yet executed. [1] proposes an approach to predict the total execution time of a query workload consisting of different types of queries that are executed concurrently. However, this paper focuses on predicting the execution time of the whole workload, and does not address the problem of predicting execution time of individual queries.

In [8], Archana *et al.* propose a system that uses machine learning techniques to predict multiple performance matrices of database queries, including(but not limited to) records used, I/O operations and most importantly, query latency. The drawback of this paper is that it does not deal with concurrent workloads. Machine learning techniques are also used in [10], which provides query latency prediction as time ranges. [6] proposes a method for predicting the performance of queries in concurrent OLAP workloads. Unlike the other papers, it does not require semantic information, but rather uses a modeling approach that depends on the analysis of query behavior in isolation, the interactions of query in pairs and sampling techniques.

Query progress indication and query performance prediction are more about predicting or estimating query latency. However, these paper make no attempt to deal with the problem of performance unpredictability, that is, they do not help to improve predictability matrices such as variance of latency, and 99th percentile of latency.

Architecture Redesign People have thought of changing the design of current database management systems to improve the predictability of performance. Surajit and Gerhard argue in [5] that the performance of current database systems are unpredictable because they are among the most complicated software systems ever built by humans and their major components seldom get redesigned when new features are added, resulting in a larger and larger and more and more complex code base that inherently causes the unpredictability of the exact behavior and performance. They propose building data management systems on RISC-style components, which are fairly simple by themselves, but can be combined to create richer components. These well-defined components with narrow functionality make it easier to predict and tune the performance of the system. However, they discuss these problems from a very high level perspective, and do not touch any implementation details. Neither do they present any design of a new data management system built upon a set of RISC-style components.

Another paper that rethinks the design of database management system is [7]. Daniela *et al.* give a summary of the requirements people have for database management systems, including performance, predictability, consistency, etc. However, due to the limitation of resources, database developers have to sacrifice some of the requirements for the others. Performance in terms of latency and throughput are usually preferred while the others receive relatively less attention. The authors point out that predictability is simply not taken into account when database developers design modern database management systems. The implementations of the classic three-tier database architecture usually do not meet the predictability requirement. Therefore they present a new three-layer architecture in which consistency is maintained in the application layer and the storage layer is only responsible for storing data. The new design makes it possible to use a lot of cheap machines in the storage layer, unlike the traditional three-tier architecture, where the database has to guarantee consistency, and must be run on only a few expensive servers.

Again, this paper addresses the problem of predictability(and other properties) from a very high level and gives no implementation details. Also, the new architecture basically delegates part of the functionality of a database to applications that uses databases, and therefore is incompatible with existing systems. Finally, this new design makes applications more difficult to build considering that they have to maintain data consistency themselves.

Predictability Improvement There has been work that proposes techniques to improve performance predictability. [2] deals with unpredictability in a more practical manner — it approaches the problem through the query optimizer. Traditionally, query optimizers use execution time as the sole heuristic for deciding which query plan to select, and ignore other important qualities, predictability being one of them. Brian Babcock *et al.* build a new optimizer that uses a probability distribution over possible selectivity, which quantifies the estimation uncertainty and allows the optimizer to select the most appropriate query plan based on the pre-defined relatively importance of performance versus predictability.

3. IDENTIFYING THE MAIN SOURCES OF VARIANCE

Database Management Systems are very complex systems, and there are many possible sources of variance, such as I/O operations, locking, thread scheduling, etc. There are tools to gather information about these events. For example, using strace, you can collect data about every I/O operation including size of data read or written, time of each I/O operation, etc. However, such kind of information is not intuitive enough for explaining the variance of latency even for the most skilled database administrators. Moreover, these tools are usually related to the kernel, thus introducing overhead such as context switching for each system call. This could be quite expensive and would also introduce noise into the latency data. These tools, therefore, are not suitable for analyzing the source of variance. We tackle this problem using a different approach.

3.1 Variance Break Down

The latency of a transaction is the time it takes to process a transaction, and the processing of a transaction can be traced down to some high level function in the system's source code that is responsible for executing the queries. In MySQL, this high level function is called `dispatch_command`. Therefore, the latency of a transaction is basically the execution time of this high level function during a transaction, and the variance of latency is the variance of the execution time of this function in different transactions.

Most of the time, a function has to call other functions to accomplish its job. A function's execution time is roughly the total execution time of all the functions it calls. In this paper, we refer to the function that calls other functions as the *parent function*, and those being called as the *child functions* of it. For example, if function X calls function Y and Z, then X would be the *parent function* of Y and Z and they would be the *child functions* of X. Let $E(f)$ denote the execution time of function f, we have

$$E(X) = E(Y) + E(Z)$$

However, strictly speaking, this is incorrect because besides calling function Y and Z, function X also has to execute some other basic statements, like variable assignments, if statements, etc. The time spent on them, although might be way smaller than the time spent on function Y and Z, must also be considered. To model this, we can add an *imaginary function* img_X to the list of children of X. In this way, we can now say that function X does nothing other than calling function Y, Z and img_X, and so

$$E(X) = E(Y) + E(Z) + E(img_X)$$

Let's say $E(X)$ has a large variance and we would like to know why. Since the only thing X does is invoking function Y, Z and img_X, we know that one or more of these functions are to blame for the variance. To find out the functions that cause $E(X)$ to have a large variance, we need to quantify the contribution of each and every *child function* of X.

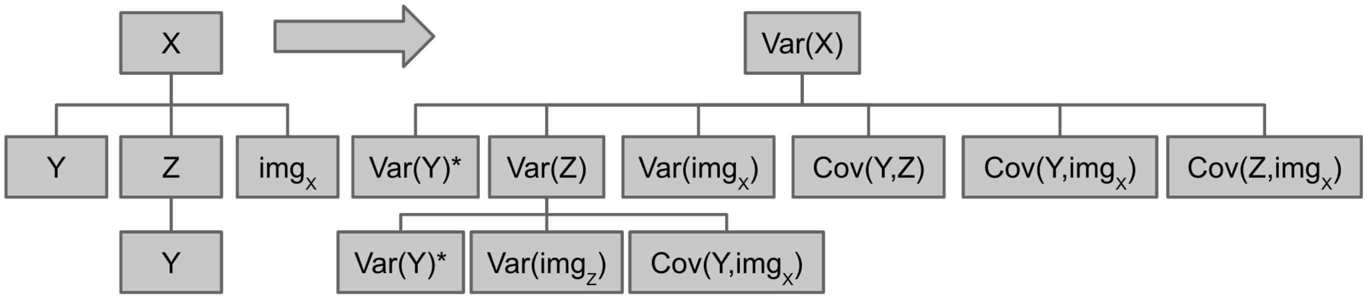


Figure 3: A static call graph and its corresponding *variance tree*

Variance can be calculated in many ways, the following one being one of them:

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

Using this formula, we can easily break down the variance of $E(X)$ into the variances and covariances of the execution time of function Y, Z and img_X, in which the variances can be further broken down by applying the same formula. For the rest of this paper, we use the term "variance of a function" as an abbreviation for the variance of the execution time of a function, and the term "covariance of two functions" as the abbreviation for the covariance of two functions' execution time for simplicity. Figure 3 shows the result of breaking down the variance of X. The graph on the left is the static call graph, which shows that function X calls functions Y, Z, and img_X. The graph on the right shows that the variance of X can be broken down into the variances of Y, Z, img_X, and the covariances of every two of them. It also shows how the variance of Z can be further broken down. We call this graph a *variance tree*.

Variance break down is fairly easy to do with the source code. Given a function, we can inserted a small piece of code into the source code of this function to retrieve the timestamp at the beginning and end of this function, and then calculate its execution time using these two timestamps easily. Similarly, for each function call in this function, we retrieve the timestamp before and after the function call and calculate the execution time accordingly.

Applying this technique on the functions in MySQL responsible for query processing allows us to see each function's contribution to the overall variance and locate those with significant contributions. Looking only at the value of variance or covariance is not enough, though. An obvious example is that function `dispatch_command` contributes to 100% of the variance of latency, but it barely gives us any useful information about the cause of the variance. This means that we also need to consider how useful a function is in helping us find out the actual cause behind the high variance.

Another thing to notice is that, a function can be called in different places in the source code, so there can be multiple nodes in the *variance tree* that represent the variance of the same function or the covariance of the same functions, each with a possibly different value. For example, in figure 3, both function X and Z calls function Y, and so in the *variance tree*, there are two nodes that both represent the variance of function Y(marked with asterisk). It is possible that to reduce the variance X, we need to change the implementation of Y, which would affect the values of both nodes. Therefore, we must consider them all together. We use the term *factor* to represent the variance of a function or the covariance of two functions during the execution of a transaction. On the other hand, the nodes in the *variance tree* represents the variance of a function or the covariance of two functions only during the execution of some particular function. We call them the *instances* of the *factor*. Each node in the *variance* corresponds to an *instance*. In figure 3, Var(Y) is a *factor*, and the two nodes marked with asterisks in the *variance tree* are *instances* of *factor* Var(Y).

The goal of finding the main sources of variance in transaction latency now turns into finding the *factors* such that:

- make significant contributions to the variance of transaction latency
- give us enough information about the cause of variance

3.2 Factor Selection

To select *factors* that satisfy the two criteria mentioned above, we need to quantify them. The first one can be quantified by the value of variance or covariance. For the rest of this paper, we use the term *contribution* to represent the value of variance of covariance. An intuition to quantify the second criterion is that, the more specific a function is about what it does, the more information it can give us about the cause of its variance. Moreover, a *child function* is usually more specific about its job than its *parent function*.

For example, a function A that writes several log records to the global log buffer must first acquire the lock on the log buffer(function B), copy the log data to the log buffer(function C), and finally release the lock(function D). Let us say function A's variance is 30% of the variance of latency and function C's variance is 28% of the variance of latency. We would find function C more interesting than function A because even though its variance is smaller than that of function A, it has narrower functionality and is more specific. It's possible that a closer look at this function can tell us that its variance is caused by the size of log data being copied to the log buffer each time, and so a solution that reduces log size variance could effectively reduce the overall variance.

Therefore, the "specificness" of a *factor* is related to the heights of the functions involved. Also, given a call graph, no matter how many different places a function occurs, all these nodes have the same height. We can use existing call graph generation tools likes CodeViz [3] to generate the call graph and pre-compute the height of any given function. Most of the time, the smaller the height is, the more specific the function is about what it does. With this intuition, we can define the "specificness" of a *factor* as a decreasing function of the heights of the functions involved. In our experiments, we use the following function:

$$speci(N) = (height(call\_graph) - height(N))^3 \quad (2)$$

In this function, $height(call\_graph)$ is the height of the sub call graph whose root is function `dispatch_command`. $height(N)$

[3]http://www.csn.ul.ie/˜mel/projects/codeviz/

```
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 $N \in t$ do
3      $N^* \leftarrow$ factor_of($h$, $N$);
4      if $N^* = NULL$ then
5          $N^* \leftarrow$ new_factor();
6          $N^*.contri \leftarrow 0$;
7          $h \leftarrow h \cup N^*$;
8      else
9          $N^*.contri \leftarrow N^*.contri + N.contri$;
10 end
11 foreach $N^* \in h$ do
12     $N^*.resp = \texttt{speci}(N^*) \cdot N^*.contri$;
13 end
14 Sort $h$ in descending order of $N^*.resp$;
15 $s^* \leftarrow$ empty list;
16 for $i \leftarrow 1$ to $k$ do
17     $N^* \leftarrow h[i]$;
18     if $N^*.contri \geq d$ then
19         $s^* \leftarrow s^* \cup N^*$;
20     else
21         break;
22 end
23 return $s^*$;
```

Algorithm 1: Factor Selection

is the height of the function if N represents the variance of function, and the larger height if N represents the covariance of two functions.

Now we can define the responsibility of a *factor* to the overall transaction latency as

$$resp(N) = speci(N) \sum_i contri(N_i) \tag{3}$$

where N is a *factor* and N_i are *instances* of this *factor* in the *variance tree*.

Given the *variance tree*, we now propose an algorithm to select the top k most responsible *factors* from this tree for the variance of latency. The pseudocode is shown in algorithm 1. The basic idea of this algorithm is very simple. We iterate over the *variance tree*, and for each node, and check if the *factor* of this node is already in list *h*. If not, we add the *factor* to *h*. If the *factor* is already in *h*, we need to add the contribution of this node(the value of variance of covaraince) to the contribution *factor* (line 1 to line 10). After we find all *factors*, we calculate their *resp* values using equation 3 (line 11 to 13). After that, we sort the *factors* in descending order of their *resp* values, and select the ones with a total contribution value greater than or equal to threshold d from the first k elements of list *h*(line 14 to 23).

3.3 VProfiler: A Variance Source Discovery Framework

The *factor selection* algorithm introduced above describes how to select the top k most responsible *factors* from a given *varaince tree*. Now we introduce a framework to identify the main sources of variance in MySQL(or other applications). We call it VProfiler, which works as follows:

```
Inputs : $v$: variance of function $dispatch\_command$,
         $k$: maximum number of functions to select,
         $d$: threshold for minimum contribution,
         $m$: threshold for minimum relatively contribution of
children
Output: $s^*$: top $k$ most responsible factors
1  $t \leftarrow$ empty tree;
2  $t$.set_root($v$);
3  $l \leftarrow$ empty list;
4  $l \leftarrow l \cup v$;
5  $e \leftarrow true$;
6  while $e$ do
7      foreach factor $f \in l$ do
8          if is_variance($f$) then
9              foreach instance $i \in f$ do
10                 $c \leftarrow$ var_break_down($i$);
11                 $t$.add_children($i$, $c$);
12                 foreach child $n \in c$ do
13                     if $n.contri/i.contri \geq m$ then
14                         $i.contri = 0$;
15                         break;
16                     end
17                 end
18             end
19         end
20     end
21     $s^* \leftarrow$ select_factors($t$, $k$, $d$);
22     $l$.clear();
23     $e \leftarrow false$;
24     foreach factor $f \in s^*$ do
25         if needs_break_down($f$) then
26             $l \leftarrow l \cup f$;
27             $e \leftarrow true$;
28         else if is_variance($f$) then
29             $h \leftarrow$ get_instance_of($t$, $f$);
30             $h.contri \leftarrow \infty$;
31     end
32 end
33 return $s^*$;
```

Algorithm 2: Work-flow of VProfiler

Initialization(Algorithm 2, line 1 to 5)

VProfiler starts with an empty *variance tree*, and then it uses the variance of the top level function `dispatch_command` as the root. VProfiler also maintains a list of *factors* that need to be further broken down(list *l* in the algorithm). This list contains only the *factor* Var(`dispatch_command`) at the beginning.

Variance Break Down(Algorithm 2, line 7 to 12)

For each factor that needs to be broken down, VProfiler finds out all its *instances* in the *variance tree*, and break each one down by modifying the source code and calculating the execution time of the functions using timestams. It then adds these nodes to the *variance tree* as the children of the corresponding node. In this way, the *variance tree* is gradually expanded.

There's one more situation we need to take care of. Using the previous example of function X, Y and Z, if the variance of X is 300 and the variance of Y is 285, then obviously we need to only focus on Y, and function X can be safely ignored. VProfiler does the similar thing. After it breaks down some *instance* of a *factor*, it first checks if any of the variance or covariance of its *child functions* takes a significant portion of the *instance*'s variance value using m as the threshold. If so, it sets the *contri* value of this *instance* to 0 because it becomes unimportant.

Factor Selection(Algorithm 2, line 13 to 23)

After the *variance tree* is expanded, VProfiler does a *factor selection* to choose the top k most responsible *factors* for the overall variance. Then, it provides the selected *factors* to the users, who needs to look at each selected *factor*. If it is the variance of a function, the users needs to look at the implementation of the function and see if it tells them enough information about the cause of the variance. If not, then they need to ask VProfiler to further break down its variance. VProfiler then add it to the list l. If this function reveals the cause of its variance, the users notify VProfiler that it's selected (Now that the users already understand the cause of its variance, this function definitely gives them "enough" information). VProfiler picks one of its *instances* in the *variance tree* and set its contribution to infinity to make sure that it will finally be selected. If there is any function that needs to be further investigated, VProfiler will continue this select-investigate loop until it finally finds out k(or less) factors that could explain the variance of latency.

Note that the factors finally returned by VProfiler are only meant to point out some functions that are highly related to the variance of latency. Close analysis is still needed to finally locate the actual problem. For example, although a factor takes into consideration the sum of all its *instances*, it could also be the case that only one or two *instances* matter while the contributions of the rest are negligible. The actual cause of variance, therefore, might be related to these important *instances* instead of the implementation of the function.

3.4 DTrace v.s. VProfiler

DTrace, which stands for *Dynamic Tracing*, is a very powerful tracing framework which can be used for performance analysis and troubleshooting on all softwares, including user-level applications like database and also kernel- level programs like operating system kernels and device drivers. DTrace is used in an event-listener manner — users specifies the event they want to listen to, along with a piece of code which will be executed when the specified event occurs. Multiple events can be specified at the same time, each called a *probe* in DTrace. Different types of probes are provided by different *providers*. For example, the *syscall* provider provides probes related to system calls, like the start and exit of the `open` system call.

The *pid* provider of DTrace allows users to dynamically patch running processes with instrumentation code without touching the static files on the disk or restarting the programs. The following is an example of a probe of the *pid* provider:

$pid : module : function\_name : \{entry, return\}$

This probe contains 4 fields, all of which can be omitted except the first one. The first field specifies the ID of the target process. The second field is the name of the module the target function is in, such as $mysqld$ or $libc$. The third field is the name of the target function, which can be either the exact name or can contain wildcard characters like "*" and "?". The fourth field should be one of entry and return, which denote the start event and exit event of the target function, respectively. When any filed is omitted, it means "match anything possible". Particularly, when the last filed is omitted, it matches both the entry probe and the return probe as well as instruction probes, which are fired for each CPU instruction executed.

Using the probes provided by the *pid* provider, we can also use DTrace to measure the execution time of a *parent function* and its *child functions*, therefore breaking down the variance of the *parent function* using formula 1.

Despite its great power in tracing both user-space and kernel-space programs and its convenience in patching running processes, there are several reasons why VProfiler is better than DTrace.

First of all, DTrace was originally developed for Solaris, and gained support by other Unix-like systems, including FreeBSD, Mac OS X, Open Solaris. Unfortunately, it's one of the tools that are not shared between Linux and Unix systems. The only Linux system that offers official DTrace support is Oracle Linux. However, neither Oracle Linux nor Unix is among the popular operating system people would choose for their servers. Popular cloud virtual machine platforms like Amazon EC2 and Microsoft Azure no longer support Unix systems. Moreover, even through DTrace is supported by Oracle Linux, a subscription account is needed to use it, which is not free. There is an unofficial port of DTrace for Linux [4]. However, up until now [5] its *pid* provider is pretty much broken, and DTrace crashes when it tries to execute any DTrace script or command containing *pid* probes.

Second, the script language DTrace uses has very limited language feature. It's dynamically typed, and does not support user-defined data type or function. What's worse, although this language supports attaching predicates to probes to decide whether or not the probes will be executed, it does not have any flow control statement like if statement or loop statement, making complex code impossible. DTrace matches functions by name. Therefore, when one function is called at multiple places of another function, it's very difficult to distinguish them in DTrace unless instruction probes are used, which would be very expensive. Moreover, when tracing the execution time of functions, we need to record the execution time of each *child function* in each transaction. To store all these data, we have to use the built-in *associative array* data type, whose performance depends on the implementation of DTrace. Even if its overhead is too huge, there is nothing the users can do to improve its performance. On the contrary, in VProfiler, we can always try to find more efficient data types or use faster algorithms to make the overhead of tracing as small as possible. Experiment results that compares the overhead of VProfiler and DTrace are shown in section 6.6.1.

Third, even if one buys a high end server to run Solaris on it, and writes the most efficient DTrace script for tracing functions, it is still possible that DTrace will disappoint the user. In our experiments with DTrace, we found that some of the return probes are lost. It turns out to be a known issue of DTrace [9]. This can happen when the target function is very complex and the compiler optimizes it using jump table, which makes it difficult for DTrace to dissemble the instructions and correctly locate the return points.

4. MAIN SOURCES OF LATENCY VARIANCE IN MYSQL

4.1 Experiment Setup

We used the framework proposed in the previous section on MySQL to find out its main sources of latency variance. We ran MySQL

[4]https:// github.com/dtrace4linux/linux

[5]Feb. 2, 2015

with the default configuration on two Amazon EC2 m3.large instances running Ubuntu 14.04 LTS, each with 2 Intel Xeon E5-1670v2 2.5GHz virtual CPUs and 7.5GiB memory. One of them had a 20GB magnetic disk while the other ones ran on a 20GB SSD. We also had another Amazon EC2 m3.medium instance to send queries to these two instances. This instance had 1 Intel Xeon E5-1670v2 2.5GHz virtual CPU, 3.75GiB memory and a 10GB SSD. We used OLTPBenchmark to run the original TPC-C benchmark on this instance and sent transactions to MySQL. We used MySQL version 5.6.21 in our experiments. In the experiments, we set the value of k to 3 and the threshold to be 5% of the variance of latency.

4.2 Main Sources of Variance

The main sources of variance we found using the framework we proposed in section 3 are shown in table 1 and 2. Note that the ones marked with an asterisk(*) are specific instances of the corresponding factors. Although we set k to 3, we actually found only two sources of variance for both. This is because the contribution of some of the *factors* with high responsibility values are too small, ie. below the threshold(5% of the overall variance), and therefore are not selected.

Source	Contribution
`Var(buf_pool_mutex_enter)`*	32.92%
`Var(img_btr_cur_search_to_nth_level)`	8.3%

Table 1: Sources of Variance on HDD

Source	Contribution(%)
`Var(buf_pool_mutex_enter)*`	16.89%
`Var(lock_wait_suspend_thread)*`	16.91%

Table 2: Sources of Variance on SSD

4.2.1 `Var(buf_pool_mutex_enter)`

Note that this main source of variance is an *instance* of a *factor*, instead of the *factor* itself. As its name suggests, this function is used to acquire the lock on the buffer pool. The buffer pool structure contains many related variables, and several lists containing the buffer pages. There are many possible operations on the buffer pool, so this function is used in a lot of different places for various purposes. Because of its generality, looking at this function by itself does not give us much useful information about the cause of its variance. We look towards its *instances* instead, and found one with a very high contribution. This function is called within the another function `buf_page_make_young`, which is used to move a page in the buffer pool to the head of the list of buffer pages. InnoDB uses a variant of the Least Recently Used(LRU) algorithm as its buffer page replacement algorithm. The buffer pages in the buffer pool are organized in a list called `LRU` list in InnoDB. As we can easily tell from its name, the pages in this list are kept in the LRU order - descending order of the latest access time of each page. Therefore, when a buffer page is accessed, it becomes the most recently used page, and should be moved to the head of the `LRU` list(InnoDB uses a variant of the LRU algorithm, so this would not happen every time a page is accessed). Since MySQL has a thread for each user, a lock on this list must be acquired before any operation can be executed. This is where the function `buf_pool_mutex_enter` comes in. Therefore, the variance here is actually the variance of the time function `buf_page_make_young` spends on waiting for the other functions to release the lock.

4.2.2 `Var(os_event_wait)`

Like the previous one, this source of variance is also a specific instance instead of a general factor. `os_event` is a high level abstraction of condition variables on different platforms, which are used in multi-threaded programs for threads to wait for some particular conditions. Therefore, this function can also be used for a lot of different purposes. This specific instance is used in a function called `lock_wait_suspend_thread`, which is related to the locking mechanism of InnoDB. InnoDB implements the Multi-Granularity Locking(MGL) mechanism. Innodb have different types of locks, including shared locks(S), exclusive locks(X), intention shared locks(IS) and intention exclusive locks(IX). Shared locks are used when transactions need to read a row while exclusive locks permit transactions to update or delete a row. Some transactions, depending on what they do, may need to acquire a lock on an entire table instead of a single record. To allow locks of different granularity to exist simultaneously, InnoDB introduces intention locks into the system. Before a transaction acquires a shared lock on a row in a table, it must first acquire an intention shared lock or a stronger lock on that table. Similarly, before it acquires an exclusive lock on a row in a table, it must first acquire the intention exclusive lock on that table. The compatibility of these different types of locks are shown in table 3('+' means compatible, and '-' means conflict). Only when the type of lock a transaction tries to acquire is compatible with the existing lock will the required lock be granted. These locks also work for indexes.

However, locks are expensive and conflicts do not always occur when transactions are executed. To reduce the overhead of locking, InnoDB introduces implicit locks into the system. Each record has a field called `trx_id`. Before a transaction performs any operation on a record, InnoDB checks if the transaction associated with the `trx_id` is active. If not, the `trx_id` of the current transaction is written to the record, and the intended operation can be carried out immediately without acquiring the lock. If the transaction is still, the implicit lock is converted into an explicit lock(which means acquiring an explicit lock of the appropriate type on the record). If the intended lock type of the current transaction conflicts with the explicit lock, the current transaction has to wait for the existing lock to be released. Innodb uses the function `os_event_wait` to suspend the thread.

4.2.3 `Var(img_btr_cur_search_to_nth_level)`

`img_btr_cur_search_to_nth_levelhis` is actually not a real function, but rather the *imaginary function* of a function called `btr_cur_search_to_nth_level`. An *imaginary function* is, as we mentioned before, a virtual function we create to represent the the work of a function done by the execution of simple statements without invoking other functions. The calculation of its time is fairly easy - subtract the sum of the execution time of all *child functions* from that of the *parent function*. The parent function, which is `btr_cur_search_to_nth_level`, is a general function for searching in a given index tree and placing a tree cursor on a given level, leaving a shared lock or exclusive lock on the cursor page. This is a rather high level function, and there's one important loop in this function for searching the index tree until it reaches a certain level. A level here corresponds to the height of a node in the tree, and leaf nodes have level 0. Therefore, the reason behind the variance of this *imaginary function* could be attributed to the variance in the desired level.

5. ACHIEVING PREDICTABILITY IN MYSQL

	S	X	IS	IX
S	+	-	+	-
X	-	-	-	-
IS	+	-	+	+
IX	-	-	+	+

Table 3: Lock Type Compatibility

```
Inputs: p: buffer page to be moved to the 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 3: `buf_page_make_young`

In the previous section, we described the main sources of variance discovered using the framework we proposed. As can be seen from the result, function `buf_pool_mutex_enter` is a problem both on magnetic disk and SSD. We studied the distribution of its execution time to see why it had such high variance, and found that most of the times, its execution time was smaller than 10ms. However, sometimes it went up as large as tens of thousands of milliseconds. For example, in one of the experiments on HDD, 96.5% of the times the execution time of this function is below 10ms, and the variance of them is only 1.33E-6% of the variance of latency. However, when taking those extreme cases into consideration, its total variance goes up to as much as 30.5% of the variance of latency. Based on this observation, we propose two methods to reduce the variance caused by this function.

5.1 Reducing Contention

Obviously we need to deal with the cases where the execution time is exceptionally high. This means that it's either waiting for some function that performs time-consuming operations on the buffer pool, waiting for a lot of functions that access the buffer pool, or both. One method is to reduce the contention between function `buf_pool_mutex_enter` and those functions. We looked into the source code of the buffer pool in InnoDB, and found a problem in the implementation of this component. The `buf_pool_t` structure in InnoDB contains more than 30 fields, and only two of them are protected by their own mutexes. The rest of the fields are protected by one single mutex(called `mutex` in the source code), including the `LRU` list. The problem is that all of these fields are not closely related to each other, and don't need to be protected by the same mutex. For example, there's a `free` list in the buffer for storing the free pages not yet used, which is also protected by `mutex`, and by the logic of the algorithm, operations on the `free` list and the `LRU` list do not conflict with each other - it's absolutely fine to perform these operations simultaneously. This single coarse-grained mutex causes a lot of unnecessary contentions among functions that do not logically conflict with each other. Therefore, to reduce the variance of function `buf_pool_mutex_enter`, we can identify the fields that are logically related to each other, and divide them into several groups. For each group, we can create a separate mutex and use this new mutex instead of the `mutex` variable in the buffer pool to protect the fields in this group. Using these finer-grained mutexes can presumably reduce the contention between function `buf_pool_mutex_enter` and other functions, thus eliminating some of the cases where the execution time shoots up by orders of magnitude.

This method has actually been implemented in *Percona Server*, an open source MySQL alternative bundled with a lot of performance and scalability improvements over the original MySQL. Splitting the `mutex` variable into multiple mutexes is implemented in *Percona Server 5.5* as an improvement of scalability of the buffer pool. We adopted its implementation of this method, and split the `mutex` variable into four other mutexes. [6] Our experiment result shows that this method improves not only the scalability, but also the predictability of MySQL.

[6]Please refer to http://www.percona.com/doc/percona-server/5.5/scalability/innodb_split_buf_pool_mutex.html for more details.

5.2 Spin Lock With Timeout

As described in section 4.3, the function we found in the experiments - `buf_pool_mutex_enter` - is called by another function `buf_page_make_young`, which is used to move a buffer page to the head of the `LRU` list to keep the pages in LRU order. Actually InnoDB uses a variant of the LRU algorithm, and does not keep the pages in the strict LRU order. It divides the `URL` list into two sublists, the young list and the old list. By default, 3/8 of the pages at the tail of the list are devoted to the old list. Pages in the old list are accessed less recently, and are swapped out of the buffer pool when new pages come in. When a page is accessed, InnoDB checks if the target page is "old" enough or not "young" enough, and moves this page to the head of the `LRU` list only if one of these two conditions is true. Therefore, pages in the `LRU` list are kept in a relaxed LRU order. We can relax this order a bit more to achieve higher predictability.

The idea is to set a limit on the time the function waits for the lock to avoid the extreme cases. We change the mutex to a spin lock in order to control the time the function spends on waiting(line 1 in algorithm 4). Using a spin lock here is not going to introduce too much overhead since more than 90% of the times the lock can be acquired within 0.01ms. When the spin lock times out and fails to acquire the lock, we give up the operation and do nothing this time. However, a page that InnoDB decides to move to the head of the `LRU` list is an important page, and is likely to be accessed again in the near future. Therefore, we can not just let it stay in the old list and slip out of the buffer pool. We have a list l for storing the pages that fail to be moved to the head of the `LRU` list, and attempt to retry the failed operations at the next time. When a failure occurs, we add the page to l. If this page has been added to l previously, we need to remove it from l and then append it to the end of l(line 2 to 7 in algorithm 4). This keeps the pages in l in the LRU order. If the lock is successfully acquired within the time limit, we need to process the list l before moving the target page. We iterate over all the pages in l from start to the end(ordering is very important here), remove each page from l, and move it to the head of the `LRU` list. But before doing that, we need to check if the page is still in the buffer pool. It's possible that because of the failure, the page remains in the old list and finally gets evicted from the buffer pool to make room for a new page, thus becoming invalid. We only move pages that are still valid to the head (line 9 to 13 in algorithm 4). After processing all the pages in list l, we finally move the target page, which is the most recently accessed page to the start of the `LRU` list.

Algorithm 3 shows exactly the original process of the function `buf_page_make_young`, which is actually very simple. Algorithm 4 shows the algorithm we described above.

5.3 Experiment Setup

The instances we used for running MySQL and the one we used to send transactions to MySQL have the same configuration as those described in section 4.1. Unless explicitly mentioned, the

```
Inputs: p: buffer page to be moved to the start of the LRU list,
        b: the buffer pool,
        l: list of pages failed to be moved,
        t: timeout for spin lock
s ← spin_for_time(b, t);
if s = failure then
    if p ∈ l then
        l.remove(p);
    end
    l.append(p);
    Return;
else
    foreach page u ∈ l do
        if in_buffer_pool(b, u) then
            buf_LRU_make_block_young(b, u);
            l.remove(u);
        end
    end
    buf_LRU_make_block_young(b, p);
buf_pool_mutex_exit(b);
```

Algorithm 4: `buf_page_make_young` with spin lock

storage devices of these instances are traditional magnetic disks. There are three different types of workloads we use in our experiments. The first one is the original TPC-C benchmark, which is an OLTP benchmark with 5 different types of transactions. Each type of transaction could have different number of queries for each transaction. The second one is TPC-C benchmark with only New Order transactions(TPC-C New Order). The number of queries in a New Order transaction is $5 + 4x$ in which x is a random number ranging from 5 to 15. The third one is TPC-C benchmark with New Order transactions that consist of exactly 45(x = 10 in the formula above) queries(TPC-C New Order Fixed).

5.4 Combing Every Possible Method

In this section, we show the best result we can have by combining every possible method we have for reducing the variance of transaction latency. We create a ramdisk, and install MySQL to that ramdisk. The `datadir` of MySQL is also set to a directory on the ramdisk. We set the value of `innodb_flush_log_at_trx_commit` to 2 and the value of `innodb_buffer_pool_size` to 1024M, and we implement the techniques we describe in section 5 in MySQL. Figure 4a and 4b show the improvement in performance under the original TPC-C benchmark, TPC-C New Order and TPC-C New Order Fixed, respectively. Figure 4c and 4d show the improvement in variance of transaction latency under these three types of workloads. In all cases, the combined method outperform the original MySQL both in performance and variance in transaction latency. Also, the variance of latency decrease as the workload changes from the original TPC-C to TPC-C New Order to TPC-C New Order Fixed. This makes sense because the discrepancy of transactions in these three types of workloads decreases as the type of transaction is fixed, and the number of queries in each transaction is fixed. Obviously, the variance of transactions in the workloads also have an impact on the variance of latency, which is quite natural.

5.5 Evaluation of Fixes for `buf_pool_mutex_enter`

In this section we evaluate the effects of the two techniques presented in section 5 on the variance of latency. We first implement these two techniques in MySQL separately and run experiments on them using all three different types of workloads. We compare the results of applying these two methods to the original MySQL, respectively. The finer-grained locking mechanism is referred to as "FG" in figure 5 and the spin lock with timeout technique is referred to as "SL" in it. Both of them show improvement in variance of latency without negative impact on the performance. Actually, they improves not only variance, but also performance, especially the average latency. The throughput of MySQL also receives a small increase. This is because these two methods both reduce contention among different functions(one by using finer-grained lock and the other by giving up lock requisition when times out), thus reducing the time the related functions spend on waiting for other functions. We then apply both of these two methods on MySQL, and a greater reduction in variance than any of these two methods alone is shown. Both standard deviation and 99th percentile reduce by more than 40% with the original TPC-C workload. For TPC-C New Order and TPC-C New Order Fixed, the reductions are more than 50%.

5.6 Using Faster Storage

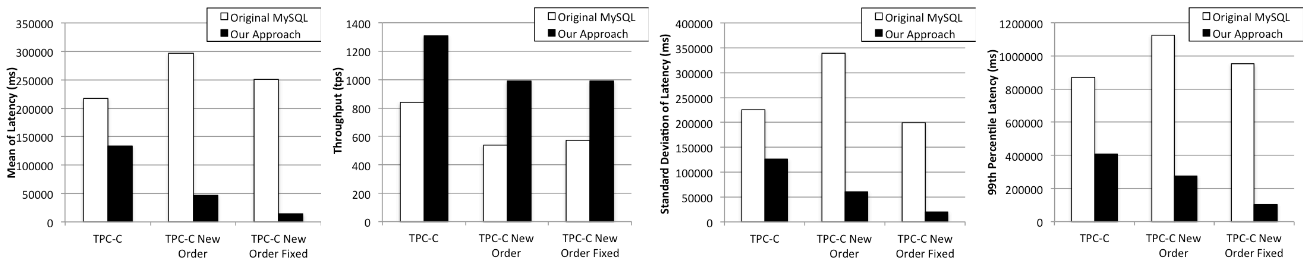
In this section we evaluate the effect of using faster storage devices on the variance of transaction latency. We run MySQL on three different instances, one with traditional HDD, one with SSD and one with a magnetic disk and a ramdisk, where MySQL is installed and all the data are stored. A ramdisk is simply a block of memory treated as a normal disk device by applications. It's a way of simulating a super-fast disk using memory, and the data would still be lost when powered off just like normal memory. As can be seen from figure 6, there's huge improvement in performance when changing from HDD to SSD and ramdisk. Also, along with that, the standard deviation and 99th percentile of latency also decrease.

5.7 Variance-away Tuning

In this section, we evaluate the effects different parameters of MySQL have on the variance of latency. We pick two parameters that we consider as closely related to the variance of latency. The first one is the size of the buffer pool. We increase the size of the buffer pool from 1GiB(default value) to 5GiB. Obviously increasing the buffer pool size keeps more data in memory, thus effectively reducing the number of buffer page replacements, the number of I/O operations and also contention within the buffer pool. As we can see from the results shown in figure 7, the larger the size of buffer pool, the lower the average latency, standard deviation and 99th percentile. Setting the size of the buffer pool to as large as possible is recommended both for better performance and for higher predictability.

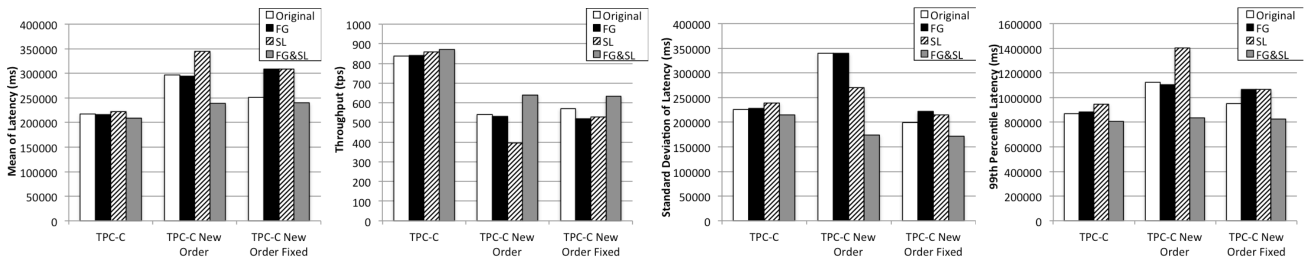
The second parameter that we choose and run experiments on is `innodb_flush_log_at_trx_commit`. This parameter affects MySQL's strategy for flushing redo logs to disk when transactions are committed. The default value of this option is 1, which means that logs are flushed to disk whenever a transaction is committed. When this value is set to 0, MySQL write the redo logs to the log files and flush the data to disk once per second, but nothing is done when transactions commit. When this value is set to 2, redo logs are written to the log file at transaction commit, but flush operations are done once per second. Although this parameter is only directly related to redo logs, changing it to 0 or 2 effectively reduces the number of I/O operations for redo logs by grouping multiple operations into one. Experiment result in figure 8 shows that setting it to 0, namely grouping multiple write operations and flush operations together, is generally better than 2, which is to group only flush operations on performance and variance of latency.

However, setting this option to 0 or 2 is risky. If the option is set to 0, which means that log data is written to file and flushed to disk



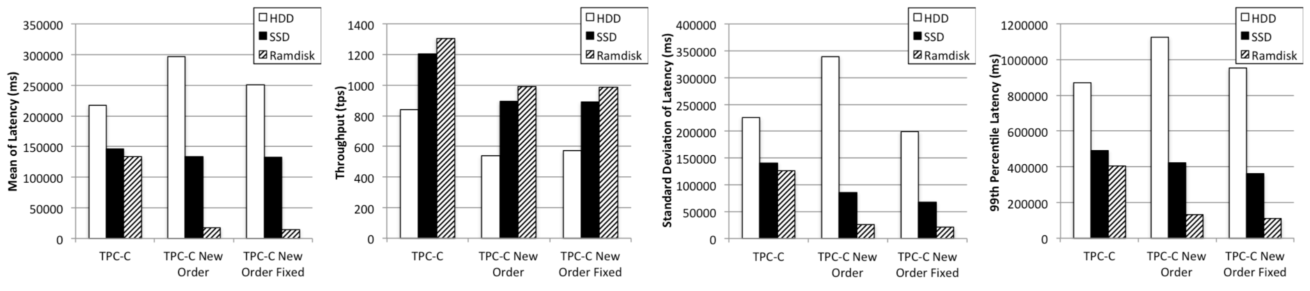
(a) Average Latency **(b)** Throughput **(c)** Standard Deviation / Mean **(d)** 99th Percentile / Mean

Figure 4: Effect of All Methods Combined



(a) Average Latency **(b)** Throughput **(c)** Standard Deviation **(d)** 99th Percentile

Figure 5: Effect of Fixes for `buf_pool_mutex_enter`



(a) Average Latency **(b)** Throughput **(c)** Standard Deviation/Mean **(d)** 99th Percentile/Mean

Figure 6: Effect of Faster Storage Device

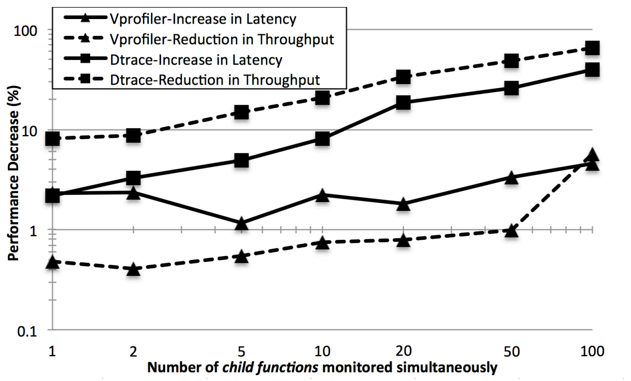


Figure 9: Profiler overhead: VProfiler vs. DTrace

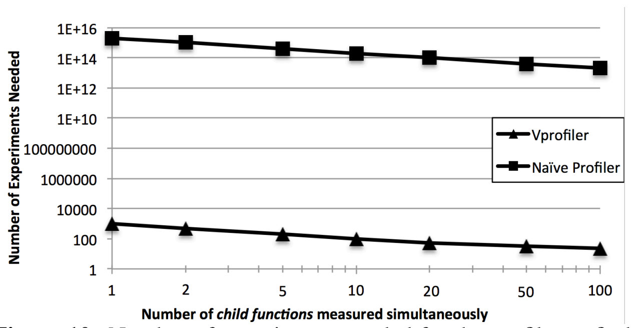


Figure 10: Number of experiments needed for the profiler to find out the main sources of variance

once per second, and MySQL crashes, all the transactions in the last second will be gone and cannot be recovered. On the other hand, when this option is set to 2, namely only write the logs to the file, the transactions in the last second will be lost when the operating system crashes(log data is safe even if MySQL crashes with this option). Therefore, this option is only suitable for applications that do not require high data consistency, such as forums, blogs, but not acceptable for applications like bank systems.

5.8 Evaluation of VProfiler

In this section, we evaluate the performance overhead VProfiler introduces into MySQL when it monitors the execution time of a *parent function* and its *child functions* to break down the *parent function's* variance, and its efficiency in finding the main sources of variance in MySQL.

5.8.1 VProfiler vs. DTrace

VProfiler breaks down the variance of a *parent function* into the variances and covariances of its *child functions* by monitoring the execution time of the *parent functions* and every *child function*. This introduces overhead to the system. In this section, we change the number of *child functions* VProfile needs to monitor and measure its performance overhead in terms of reduction average latency and throughput.

As a comparison, we also evaluate the performance overhead of DTrace. Due to its incomplete support on Linux, we had to run a virtual machine on VirtualBox with 10 Intel Xeon E5-2450 2.10GHz virtual CPUs and 15GiB of memory and run Solaris 11 on it.

In this experiment, we change the number of *child functions* to trace from 1 to 100. Figure 9 shows the experiment results. We can see that the overhead of DTrace is a lot higher than VProfiler. The overhead of DTrace in terms of reduction in latency and throughput grows rapidly as the number of child functions it traces simultaneously increase, while the overhead of VProfiler stays below 6%, because the monitoring technique VProfiler uses is pretty lightweight.

5.8.2 VProfiler vs. Naïve Profiler

In this section, we compare VProfiler to a naïve profiler, which is very similar to VProfiler but breaks down every factor when possible instead of only a few important ones. We use this profiler on MySQL and evaluate the number of experiments this profiler has to run to find out the main sources of variance with different numbers of functions to monitor simultaneously in each experiment.

There are totally 2×10^{15} nodes in the call graph(one function called in multiple places are regarded as different nodes in the graph), 4.5×10^{14} of which being leaves. Since the naïve profiler has to break down every non-leaf nodes, the number of experiments needed is extremely large. The selection strategy of VProfiler greatly reduces the number of experiment it has to run to locate the main sources of variance.

6. CONCLUSION

7. REFERENCES

[1] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Interaction-aware prediction of business intelligence workload completion times. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 413–416. IEEE, 2010.

[2] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 119–130. ACM, 2005.

[3] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When can we trust progress estimators for sql queries? In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 575–586. ACM, 2005.

[4] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2004.

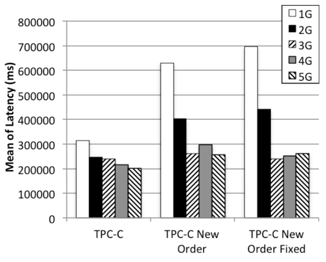
[5] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10. Citeseer, 2000.

[6] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 337–348. ACM, 2011.

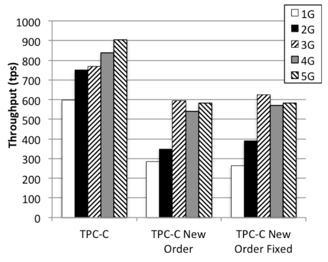
[7] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *ACM Sigmod Record*, 38(1):43–48, 2009.

[8] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael I Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

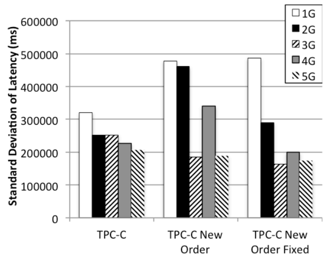
[9] Brendan Gregg. DTrace pid Provider return. `http://dtrace.org/blogs/brendan/2011/02/`



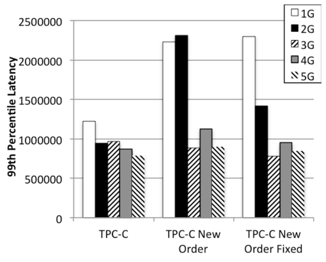
(a) Average Latency



(b) Throughput

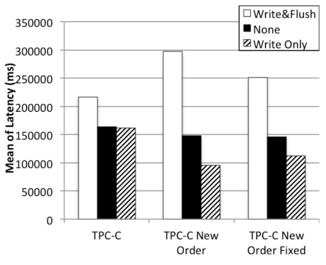


(c) Standard Deviation

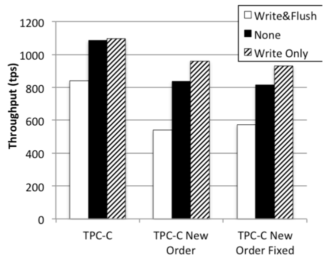


(d) 99th Percentile

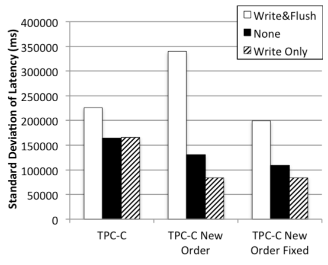
Figure 7: Effect of Buffer Pool Size



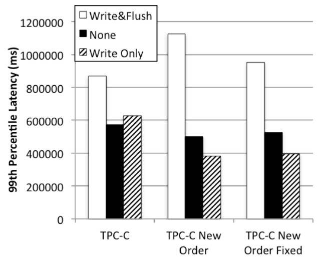
(a) Average Latency



(b) Throughput



(c) Standard Deviation



(d) 99th Percentile

Figure 8: Effect of Log Flush Policy

`14/dtrace-pid-provider-return/`, 2011. [Online; accessed 23-Feburary-2015].

[10] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. Pqr: Predicting query execution times for autonomous workload management. In *Autonomic Computing, 2008. ICAC'08. International Conference on*, pages 13–22. IEEE, 2008.

[11] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 791–802. ACM, 2004.

[12] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. Increasing the accuracy and coverage of sql progress indicators. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 853–864. IEEE, 2005.

[13] Gang Luo, Jeffrey F Naughton, and S Yu Philip. Multi-query sql progress indicators. In *Advances in Database Technology-EDBT 2006*, pages 921–941. Springer, 2006.