Finding Complex Concurrency Bugs in Large Multi-Threaded Applications

Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Parallel software is increasingly necessary to take advantage of multi-core architectures, but it is also prone to concurrency bugs which are particularly hard to avoid, find, and fix, since their occurrence depends on specific thread interleavings. In this paper we propose a concurrency bug detector that automatically identifies when an execution of a program triggers a concurrency bug. Unlike previous concurrency bug detectors, we are able to find two particularly hard classes of bugs. The first are bugs that manifest themselves by subtle violation of application semantics, such as returning an incorrect result. The second are latent bugs, which silently corrupt internal data structures, and are especially hard to detect because when these bugs are triggered they do not become immediately visible. PIKE detects these concurrency bugs by checking both the output and the internal state of the application for linearizability at the level of user requests. This paper presents this technique for finding concurrency bugs, its application in the context of a testing tool that systematically searches for such problems, and our experience in applying our approach to MySQL, a large-scale complex multi-threaded application. We were able to find several concurrency bugs in a stable version of the application, including subtle violations of application semantics, latent bugs, and incorrect error replies.

Categories and Subject Descriptors D [*2*]: 5

General Terms Algorithms, Reliability

Keywords Concurrency bugs, Latent bugs, Linearizability, Semantic bugs

1. Introduction

As processors become more and more parallel, applications must become increasingly concurrent to take advantage of this additional processing capacity. However, concurrent systems are notoriously difficult to design, test, and debug, since they are prone to concurrency bugs whose occurrence depends on specific thread interleavings.

Recent advances in the area of testing concurrent systems have provided a series of new techniques to systematically explore different thread interleavings and maximize the chances of exposing concurrency bugs [Burckhardt 2010b, Musuvathi 2008]. However, these techniques assume that concurrency bugs manifest themselves either by causing the program to crash (e.g., due to an illegal memory access) or by triggering assertions written by the developer.

This assumption, however, prevents such tools from capturing some bugs, namely those that fall into the following two important classes. The first class are bugs that manifest themselves as any violation of the application semantics which is not caught by the assertions that the programmer wrote, such as returning an incorrect result to the user. The second class are latent bugs, which silently corrupt internal data structures, and only manifest themselves potentially much later when they are triggered by a subsequent input. These two classes constitute a non-trivial fraction of the concurrency bugs found in important concurrent applications [Fonseca 2010].

In this paper we propose a new technique for finding latent and/or semantic concurrency bugs. Our thesis is that it is possible to implicitly extract a specification, even for large multi-threaded server applications, by testing if the application obeys linearizable semantics [Herlihy 1990]. Intuitively, linearizability means that concurrent requests behave as if they were executed serially, in some order that is consistent with the real-time ordering of the invocations and replies to the requests. While similar ideas have been applied to the design of tools for testing concurrency bugs, they have been limited to testing the atomicity of small sections of the program or library functions with at most hundreds of lines of code [Burckhardt 2010a, Vafeiadis 2010, Xu 2005]. We push this idea to an extreme by postulating that even a complex multi-threaded server with hundreds of thousands of lines of code can come close to obeying linearizable semantics.

By systematically testing if linearizability is upheld, we can find subtle violations of the application semantics without having to write a specification for each concurrent application. Furthermore, by checking if both the output and the internal state of the application obey the inferred semantics, we can identify not only the bugs that manifest themselves immediately as a wrong output, but also those that silently corrupt internal state. However, achieving a meaningful state comparison requires abstracting away many of the low-level details of the state representation. We accomplish this by means of simple annotations that are provided by the tester. This approach also allows the tester to progressively increase the chances of finding latent concurrency bugs by incrementally annotating the state.

We implemented PIKE, a testing tool that brings together these principles and state of the art techniques for the systematic exploration of thread interleavings. We describe the design and implementation of PIKE, and our experience in applying it to MySQL, a multi-threaded database server with hundreds of thousands of lines of code, which represents a share of 40% of the database market [Oracle a].

Our experience demonstrates that, despite the size and complexity of MySQL, in practice the semantics it provides are sufficiently similar to linearizability for our detector to be effective. Although we used only a simple battery of inputs for testing (based on the testing inputs that shipped with the application) we were able to find a considerable number of concurrency bugs in a stable version of the database. Furthermore, the effort to provide the required annotations was small, and after installing simple filters we also found the number of false positives to be modest. All of this was achieved without having to figure out which were the correct outputs (or final states) for any given inputs, since PIKE automatically extracts a specification by comparing the outputs and states of different interleavings.

The remainder of the paper is organized as follows. Section 2 presents the problem and gives an overview of our approach. In Section 3 we introduce PIKE, the tool that we built to find concurrency bugs. Section 4 describes our experience applying PIKE to MySQL. Section 5 presents the results that we obtained from our experience. Finally, Section 6 reviews the existing related work and then we conclude in Section 7.

2. Overview

This section gives an overview of the specific classes of concurrency bugs we are targeting, and the main insights behind our solution.

2.1 Problem statement

The focus of this work is to improve testing techniques for detecting concurrency bugs. By concurrency bugs we mean any deviation from the intended behavior of the application that is not triggered deterministically: triggering concurrency bugs requires that the application is given not only a specific set of inputs that cause the unintended behavior, but also that the operating system executes the application with a specific schedule of thread interleavings. This non-determinism that is introduced by thread scheduling makes the application more error-prone, on the one hand, and these concurrency bugs harder to find, on the other.

In this paper we focus on two categories of concurrency bugs that are particularly hard to detect: semantic and latent bugs.

Semantic concurrency bugs manifest themselves by violations of the application semantics (e.g., by providing a wrong reply to clients). This category of bugs excludes those that crash the application or otherwise generate an exception, for example, by performing illegal memory accesses. Semantic bugs are hard to detect because it is hard to create a specification for reasonably complex applications, and some of the deviations from the intended behavior can be quite subtle. A partial form of specification that can be leveraged to capture some of these problems are assertions, but these only capture some deviations from the intended semantics and are therefore of limited use.

The second category of concurrency bugs that we focus on are *latent* concurrency bugs [Fonseca 2010]. A latent bug is one whose effects are exposed to clients at a significantly different point from the point where it is triggered (i.e., where the problematic thread interleaving leads to the application deviating from the internal behavior that was intended by the developer). Note that the two categories are not mutually exclusive, i.e., a bug can be latent and expose itself to clients by returning a wrong reply.

Because the main focus of our paper are multi-threaded server applications, we can be more specific about what it means for triggering and exposing the bug to occur at different points. In particular, server applications follow a pattern where they receive requests from clients, start the process of handling them, and conclude handling the requests when they issue the replies. Given this pattern, we define a concurrency bug as latent when the set of requests that trigger the bug does not include the subsequent request that causes the bug to be exposed to the clients.

The fact that latent bugs require an extra request to become visible means that testing tools can easily miss these bugs. In particular, this would happen if the set of inputs used by these tools would cause the state corruption to take place but not include the input that subsequently exposes the misbehavior. For example, in many applications, write operations typically only return to clients a status code specifying whether the write was successful or not. In this case, observing the application response might not be sufficient to infer whether the write operation was properly executed or not. Furthermore, as we will see in some of the cases we found in a real application, it is often not simple to determine which subsequent inputs would be required to expose the incorrect internal state.

This supports the idea that, given the nature of latent concurrency bugs, detection tools should inspect the internal state of the application and not limit themselves to analyzing the outputs. However, it is hard to analyze the state of an application to check for its correctness, since it requires application-specific knowledge of what it means for the state to be incorrect.

2.2 Linearizability: The spec from within

To address the absence of a specification capturing these deviations from the intended application semantics we propose extracting such a specification from the behavior of the same application but under different conditions.

In particular, our thesis is that, even for a complex server application with hundreds of thousands of lines of code, the semantics that are intended by the programmer are normally close enough to linearizability [Herlihy 1990] that we can use it as a good first approximation of a specification.

To formally define linearizability, we must first define the notion of history, which is a finite sequence of events that can be either invocation of operations or responses to operations. A history is classified as sequential if its first event is an invocation, each invocation is immediately followed by a matching response, and each response is followed by an invocation. Two histories H and H' are defined as equivalent if, for every process P, the sequence of invocations and responses performed by P is the same, i.e., $H|P = H'|P$. A history H induces an irreflexive partial order $<_H$ on operations such that $o_0 <_H o_1$ if the response event for o_0 precedes the invocation of o_1. Given these definitions, a history of events in a concurrent system H is linearizable if there is an equivalent sequential history S (called a linearization of H) such that $<_H \subseteq <_S$.

Intuitively, this means that, despite its internal concurrency, the server behaves as if requests were processed in sequence, and that this processing took place instantaneously some time between the moment when the client invoked the request and received the respective reply.

Therefore, assuming the application tries to follow linearizable semantics, a testing methodology can be devised by comparing each concurrent execution of the application with all possible linearizations (i.e., all possible sequential executions of the requests) for the same input. If none of the linearizations matches the behavior of the concurrent execution, then a concurrency bug is suspected to have been triggered and an error is flagged.

Testing for linearizability would only require us to inspect the outputs of the concurrent execution against the outputs of the linearizations. This would be sufficient to capture semantic bugs, but not to capture latent bugs. To handle latent concurrency bugs, we can resort to the same principle of testing for linearizability but applying it to the state of the application. This testing methodology is summarized

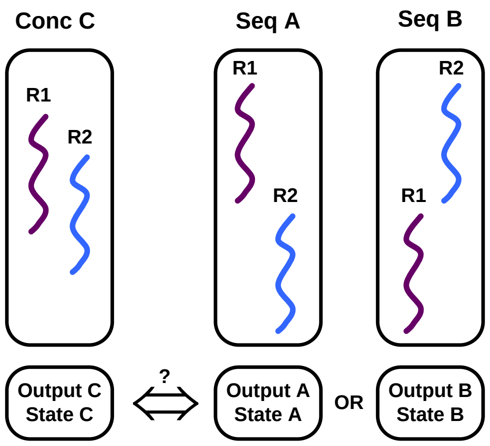


Figure 1. Checking for linearizability of state and outputs of two concurrent requests.

in Figure 1. It shows two concurrent requests, R_1 and R_2, whose execution overlaps in time (Conc C). To check if the concurrent execution is linearizable we must compare the it to all possible linearizations, namely R_1 followed by R_2 (Seq A) and R_2 followed by R_1 (Seq B). Linearizability is obeyed when both the state and the output of the concurrent execution match both components in at least one of the two linearizations.

Note that by using linearizability as a specification, we are not necessarily extracting a correct specification of the system, not only because the programmer might not have intended the application to obey linearizable semantics, but also because the sequential execution may be buggy, and consequently the deviation to the expected behavior could go undetected. The latter issue is not problematic in the case of concurrency bugs, though, since these arise from the lack of proper synchronization among multiple threads, which does not arise when executing requests without concurrency.

2.3 Capturing application state

As we mentioned, to be able to find latent bugs we need to compare both the output and the state of different executions of the application.

While outputs are fairly straightforward to compare, the same cannot be said about the state of the application. In particular, the naïve approach of simply comparing the state of the various executions bit-by-bit is doomed to fail. The reason is that by changing thread interleavings, the low-level state of the executions will quickly diverge. For instance, if we consider operations such as dynamic memory allocation, slight changes in the thread interleaving could easily change the relative order of allocation requests, and therefore the memory layout of allocated heap space would likely be different as well.

We address this by asking the tester or the developer of the application to provide a *state summary function* which captures an abstract notion of the state in a way that takes into consideration the semantics of the state and allows for a logical comparison, instead of a low-level physical comparison. As an example, a data structure that represents a set of elements should be compared across different executions in such a way that is not only oblivious to the memory layout, but, given that sets can be stored in data structures that imply an ordering such as a list, but the order in which the elements of a set are listed is irrelevant, the state summary function must be oblivious to this order.

While writing this extra code could be a burden for the tester or the developer, we found that in practice these functions are simple to write in part because the internal API of the application is reasonably well defined. Additionally we provide a small library that assists programmers in writing state summary functions for the most common types of data structures. Finally, we note that in our testing framework the state summary functions will always be scheduled until completion, without the possibility of being preempted, and therefore do not have to be synchronized with respect to the existing code nor vice-versa.

2.4 Maintaining the summary functions

Annotating the application undoubtedly requires some effort from testers. During the life-cycle of the application it might not suffice to annotate it once – it might be necessary for testers to revise the annotations when there are new versions of the application. Major updates to the application (which typically involve substantial code rewrites) are likely to require some effort to update the summaries.

But, in practice, we expect that many upgrades to the application will maintain most of the properties of the data structures as well as the interface that is used to access them. In these cases, no changes to the annotations would be required.

3. PIKE: A concurrency bug finding tool

In this section we describe how we combine our linearization approach, which analyzes both the output and the state of different interleavings for linearizability violations, with state of the art testing techniques. The result is a bug finding tool geared towards finding concurrency bugs that are traditionally hard to detect.

3.1 Systematic schedule exploration

The distinguishing property of concurrency bugs, in comparison with non-concurrency bugs, is the fact that only specific interleavings trigger these bugs. This implies that testing concurrent applications requires finding mechanisms to explore multiple thread schedules. However, with the exception of very small applications, it is not feasible to explore all possible thread interleavings because of the state explosion problem.

The traditional approach for exploring interleavings relies on stress testing and noise generation [Ben-Asher 2006]. Despite their widespread use, these techniques suffer from three important limitations. First, such techniques are not systematic, i.e., they do not try to avoid redundant or similar interleavings. Second, such techniques do not attempt to prioritize interleavings that are more likely to trigger concurrency bugs. And finally, when a bug is found, these approaches may not allow for reliable replay of the interleaving that triggered the bug.

To overcome these shortcomings, researchers have developed tools to explore thread interleavings in a more controlled manner [Burckhardt 2010b, Eytani 2007, Musuvathi 2008]. These tools try to avoid redundant interleavings, prioritize some interleavings over others, and are able to replay previously run schedules. Such features greatly contribute to improving the ability of developers to explore relevant thread schedules and uncover concurrency bugs. However, to detect when a bug is triggered the developer still has to rely on techniques like programmer-written assertions or the program generating an exception.

PIKE combines our proposed linearizability detector with the *random scheduler* algorithm proposed by Burckhardt et al. which is used by PCT [Burckhardt 2010b]. Here, we briefly describe the general idea of the *random scheduler* algorithm. We refer the reader to the original paper for a more detailed explanation.

At the start of the test run, the *random scheduler* assigns a fixed random priority to each thread and, to control the outcome of data races, it will only allow one thread to run at a time. During execution it makes sure that, at any point in time, the highest priority unblocked thread is the one that is allowed to run. Additionally to explore bugs of different *depth*, at a few random points during the execution it changes the priority of the threads. This simple approach, which offers probabilistic guarantees, has been shown both analytically and empirically to work well at uncovering concurrency bugs [Burckhardt 2010b].

3.2 Handling false positives

One of the challenges we expected to face when deploying PIKE is that linearizability would not necessarily hold for a large, complex application with rich semantics and hundreds of thousands of lines of code. These cases, if not appropriately dealt with, could lead to the tool outputting a large number of false positives.

An example of a data structure that we found to sometimes not obey linearizability is an application-level cache. In particular, this happened in situations where the application logic detected that two requests were being handled concurrently and that would cause a cache entry that one of them would create to be invalidated. In these cases, the application would conservatively not insert that entry into the cache. This behavior might have an impact on performance but does not affect correctness, i.e., an application can always choose not to insert an entry into the cache. However, if the application were to execute the same requests sequen-

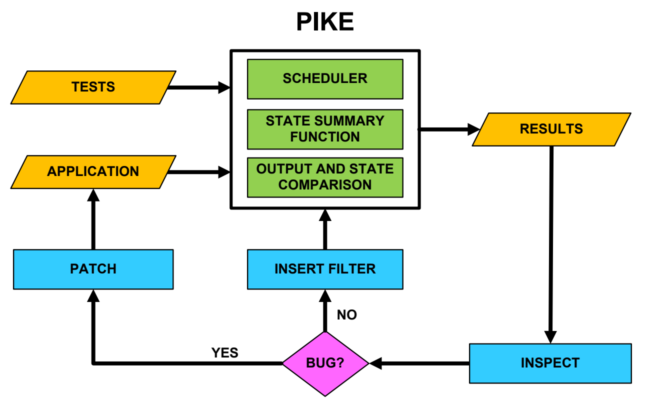


Figure 2. Overall architecture of PIKE. The system receives as inputs a multi-threaded application and a test suite, and contains a feedback loop that can be used by testers to insert filters to avoid false positives when the application deliberately violates linearizability.

tially, because no possible conflict would exist, the last request would be inserted into the cache.

To handle these cases, the state summary functions break the state up into separate components; e.g., an application-level cache would be an individual component. Furthermore, we allow the tester to write a rule that enables the linearizability test to check for inclusion, instead of equality, among the set of entries in some of the state components. In the case of the application-level cache, this rule might allow for checking whether the set of elements in the cache for the concurrent execution are contained in set of elements in the cache for at least one of the sequential executions. We found this approach to work well in practice in reducing the number of false positives to a reasonable level.

Therefore, our final system design contains a feedback loop where testers can add rules that describe such exceptions to linearizability, thus avoiding most false positives and making the problem tractable.

Figure 2 illustrates the overall process. Developers provide PIKE with the application and the testing inputs. PIKE will then run the application multiple times exploring different thread interleavings and checking for linearizability of both state and output. To conclude whether a bug was found, the developer then inspects the results produced by PIKE which include the output, the state and information about the interleaving of the various executions. In case the developer finds various cases of similar false positives he can simply insert a rule to adjust the comparison functions and re-run PIKE.

3.3 Implementation

As Figure 2 also shows, the implementation of PIKE is composed of three components: the scheduler, the *state summary function*, and the component to compare the state and output of the application.

We implemented the scheduler in about $3,000$ lines of C code. Our scheduler controls the thread interleaving by intercepting the library calls of the target application and forcing a single thread to run at a time which is randomly chosen according to the *random scheduler* algorithm.

Our scheduler takes control of the application using the *LD_PRELOAD* environment variable and intercepts the pthread library calls made by the application; i.e, the scheduling granularity is at the level of the pthread library calls. Similar levels of granularity have previously been found to produce good results at finding concurrency bugs [Musuvathi 2008].

We require application writers to identify the location where the handling code of each request begins and ends. The scheduler needs to know about these locations to force interleavings that translate into sequential executions. This information also helps in debugging the application when bugs are flagged. Since our scheduler only takes control of the application when it makes pthread calls, it could happen that the running thread (i.e., the runnable thread with highest priority) invokes a system call that does not return. In such a situation, the entire application would block – the highest priority thread would be blocked on a system call and the other threads would have previously been blocked by the scheduler. A situation where this would occur is in the location where the main thread of MySQL spawns new threads to handle new client sessions. To avoid this, we make the scheduler aware of that particular location in the MySQL code and make the scheduler block the main thread as soon as it creates all the expected client-session threads (which is dependent on the input). In comparison with the effort to annotate the application for the purpose of capturing the application state, the effort required to identify these three locations was negligible.

The random scheduler algorithm requires a few parameters to be specified [Burckhardt 2010b]. In our experiments we used the value $50,000$ as the maximum number of execution steps per run (after the initialization phase) and we used a single priority inversion point (i.e., we tuned the scheduler to find bugs with depth one). These values were empirically found to produce good results for the application we studied.

The random scheduler algorithm also requires an anti-starvation mechanism. Without this mechanism if the highest priority thread enters a busy wait cycle it would never relinquish the processor and would prevent the entire application from progressing. Examples where such situations could occur are the instances where ad-hoc synchronization methods are used [Xiong 2010]. We implemented the anti-starvation mechanism simply by reducing the priority of the running thread if it runs uninterrupted for more than a certain number of execution steps. We found this mechanism to be particularly useful during initialization periods.

Our implementation also includes a generic library for assisting in capturing the state of the application, however the exact code to capture the state is dependent on the application. In Section 4 we describe our experience with applying PIKE to MySQL.

4. Experience

This section reports on the experience of applying PIKE to find concurrency bugs in MySQL.

4.1 MySQL overview

MySQL represents a challenging case study for our testing tool for several reasons. First, it is a large, complex codebase, with about $360,000$ lines of (mostly C and C++) code and rich application semantics. Second, databases are a critical component of the IT infrastructure of many organizations and therefore it is important to maintain and improve their robustness. In particular, MySQL represents a share of 40% of the database market [Oracle a], and is by far the most popular open-source database server. Finally, MySQL is a mature application with a quality development and maintenance process. The results presented here report on applying our technique to a stable version of MySQL (version 5.0.41).

One of the characteristics of MySQL is that it supports different mechanisms, which are called storage engines, for internally representing and manipulating the state of the database. Users can control which storage engine to use dynamically by parameterizing certain requests during runtime (e.g., *Create Table*) or specifying configuration options set by an administrator. Storage engines represent a significant fraction of the source code of MySQL and implement important parts of the database functionality such as support for indexes and caches, the granularity of locks, and support for compression, replication, or encryption.

To validate our detector we chose to apply it to the MyISAM storage engine. MyISAM [Oracle c] is considered to be one of the most popular storage engines of MySQL [Oracle b] and it has also traditionally been the default storage engine [Oracle c]. In comparison to other engines, MyISAM is optimized for throughput, and is distinctive in that it does not provide the ability to group multiple operations into transactions: instead users have at their disposal explicit locking mechanisms to enforce consistency among groups of operations.

An important point to clarify regarding the semantics of the database server is that we are using PIKE to test for linearizability at the level of individual client requests (i.e., SQL operations), which may differ from the semantics that are provided at higher levels, such as transactions. In particular, it is possible for a database server to offer semantics that are weaker than linearizability, e.g., snapshot isolation at the level of transactions, but still be linearizable at the level of client requests.

4.2 MySQL internal state

As explained in Section 2, PIKE checks whether the application exhibits a linearizable behavior by comparing the internal state of different executions of the application. To achieve this goal, PIKE needs to generate a high-level representation of the internal state which is done in an application-specific way.

By analyzing the source code and based on existing studies [Fonseca 2010] we were able to identify the following data structures, which we believe capture the most important components of the application state.

The *query cache* structure contains pairs of recent instructions that read the state of the database (*SELECT* statements) and their respective results. This structure has been found by its developers to be critical for servers to achieve good performance in many common scenarios. The *query cache*, as one would expect from a cache, should invalidate the relevant entries when they become obsolete due to subsequent and conflicting writes. If the invalidation logic in the application is incorrect it is likely that such mistakes will lead to bugs in which the application returns the wrong results to clients.

The *table cache* stores a set of descriptors, each of which is an in-memory representation of a table schema. When a new thread wants to manipulate a table, it first queries the *table cache* to get a table instance directly if available. Otherwise, in case of a miss, the table schema will be loaded from disk and a new entry will be inserted into the *table cache* structure.

Another type of data structure that we annotated were the *data files*. A *data file* is a critical data structure that stores the actual records for a particular table and is maintained in persistent storage.

To quickly perform searches and find the relevant records in a table, avoiding sequentially scanning the whole table, MySQL also maintains for each table an *index file* which consists of a set of indexes. Each entry in the *index file* consists of a pair of elements. The first element is a *key* (or a group of keys) while the second element is a pointer to the appropriate record in the *data file*.

The *key cache* is a repository for frequently used blocks from the *index files* of all tables. The index block will be loaded into the *key cache* before the first access to a table. From that moment on all subsequent operations will be performed on *key cache* data and will be flushed back to disk at the appropriate time.

Finally, the *binary log* is another important data structure that we annotated. It stores a sequence of all operations that changed the database state, in their order of execution. This structure is critical for replication. Replicas keep their state in sync by shipping the *binary log* between them and re-executing the requests in the order they appear. Missing entries, wrong entries or entries in the wrong order will likely cause replicas to diverge and therefore it can seriously affect the correctness of the service. Additionally the *binary log* is important for recovery purposes.

4.3 State summary functions

To write the summary functions, which capture different parts of the state, we analyzed these different state components and classified them in two categories according to what type of data structures they represent.

Most data structures fall into the *set* category, since they are collections of elements where their order does not matter, except for the *binary log* structure which is an append-only *sequence* where the order in which the elements are added needs to be captured by the summary function.

Starting with the state components that describe sets, their summary function needs to be invoked in all places in the source code where elements are added, removed or modified to or from any of these data structures. Despite the complexity of the state, locating these turned out not to be too complicated since the source code of MySQL is reasonably well structured and there are functions that encapsulate these operations which are called from different points in the code.

At each of these points we invoke a generic summary function for sets, which is designed to provide an efficient update and comparison operation. This function maintains a cumulative hash value for the set (S) which is initialized to zero at the beginning of the execution. Then, upon adding or removing an element e, the summary function captures a hash of the deterministic parts of the element being added or removed (H_e). In this step it is important to remove sources of non-determinism like timestamps that would lead to state divergence. Some of the data structures annotated contain elements contain pointers. In these cases instead of hashing the pointers we hash the elements they point to.

Then, the value of H_e is either added or removed to the cumulative set value S. Both adding and removing is done by XORing the new value with the previous cumulative value, i.e.:

$$S_{new} = S_{old} \oplus H_e. \qquad (1)$$

This leads to a compact representation of the state of the set that allows for a trivial comparison operator simply by comparing hashes.

Operations that modify elements are handled by treating them as a sequence of an add and a remove operation.

For the *binary log*, this representation does not work because it does not capture the order in which elements were added to the sequence. Therefore, we change the above equation to capture this order by hashing the concatenation of the previous cumulative value with the new element.

$$S_{new} = SHA1(S_{old}||e). \qquad (2)$$

Finally, we also needed to extend this scheme to support containment instead of equality checks for sets. This can be easily achieved by replacing the cumulative XOR of the hash values with a counting Bloom filter [Bonomi 2006]. Alternatively, we can just list all the elements in the set and compare them exhaustively, which is what is done by our implementation.

4.4 Input generation

Like other dynamic bug finding tools, our testing technique requires exploring different inputs in an attempt to find situations in which the application behaves incorrectly. Therefore we must find a diverse set of concurrent database operations that stand a good chance of triggering bugs. Again, the rich semantics and wide interface of MySQL make it particularly challenging given that we can only practically explore a small subset of all possible inputs.

We considered different options for generating test inputs. The obvious option is to generate the inputs manually; however this can be tedious and impractical for applications like MySQL. Another option is to randomly generate inputs, possibly with the aid of grammars that steer the input generation into generating inputs that are considered more useful. This option suffers from the problem that it is not straightforward to instrument the grammar in such a way that it creates multiple concurrent requests that are likely to cause contention for some particular part of the state of the application. A third option is to use tools that analyze the application, try to understand its behavior, and then attempt to automatically generate useful inputs [Cadar 2008a, Godefroid 2005]. However, while these tools work well for small and medium size applications, it is unclear if they can currently scale to the size of a codebase like MySQL.

Therefore we pursued a fourth option. MySQL already contains a large test suite, which has been manually created by the developers and testers of the application. Some of the tests were added specifically to prevent previous bugs from recurring in subsequent versions of the application. However, these tests are sequential tests and therefore would not be useful for finding concurrency bugs. Our solution was to convert these sequential tests into concurrent tests by breaking up the sequence of requests contained in a test and executing them concurrently by separate clients.

When deciding how many concurrent clients to use in our tests, we took into account that studies show that a significant amount of the concurrency bugs found only require a small number of threads to be triggered (typically two) [Lu 2008]. A separate study also showed that only a small number of requests is sufficient to expose bugs [Sahoo 2009]. Taking these factors into consideration, and to make the process more efficient, we generated tests involving two clients and with a limited number of requests per client (typically less than ten requests and starting from an empty database).

The original complete test suit contained approximately $50,000$ requests, as counted by the number of semicolons. Using our approach we manually converted around 5% of those requests into concurrency tests, thus generating 1550 pairs of inputs from concurrent threads.

One could imagine extending MySQL's traditional testing approach to also include concurrency bugs tests instead of just deterministic tests. Similarly to Pike, the extension to the traditional testing approach would also require the use of tools to explore thread interleavings. One of the problems with this extension is that testers would have to manually specify (and update) the set of expected outputs for each test case. Pike instead finds that set automatically for arbitrary inputs. Furthermore Pike analyzes the application internal state to detect latent concurrency bugs.

In the future, we plan to explore other approaches for generating inputs and use them with PIKE.

5. Results

In this section we present the results of our experience of applying PIKE to MySQL.

5.1 Development effort

The first result we report on was the the amount of effort needed to understand the code of MySQL and develop the *state summary functions*. The annotations we inserted added up to 600 lines of code, as counted by the number of semicolons. This represents less than 0.2% of the number of semicolons in the MySQL source code.

While annotating the source code of MySQL, most of the effort was spent understanding the source code. We spent a total of about two man-months in the process of understanding both the structure and semantics of the application and annotating the source code.

5.2 Bugs found

We ran PIKE on MySQL opportunistically in a shared cluster using multiple machines (up to 15 machines). Each machine in the cluster had an AMD Opteron 2.6 GHz processor, 3 GB of RAM and was running a distribution of Linux with kernel version 2.6.32.12.

We tested MySQL by running it on 1550 inputs and for each input we configured PIKE to explore 400 different interleavings using its scheduler. The experiment lasted for about one month. Our implementation of PIKE could be optimized to reduce the computational cost in several ways. In particular, we could avoid going through the initialization phase of MySQL for each run by taking advantage of snapshoting techniques. Another way of speeding up testing could be to run PIKE on the target application previously compiled with optimization flags. The few inputs for which suspicious behavior is observed could then be re-executed with additional debugging support (on the version of the target application not optimized and with application-level debugging options enabled).

During our testing experiments PIKE was able to identify a total of 12 inputs that triggered concurrency bugs. Table 1 presents an overview of the inputs that we found to trigger incorrect behavior and in the following subsections we present our findings in more detail for different types of bugs, categorized according to their effects.

In some cases, we had different inputs that triggered bugs that showed similar effects. Because it was difficult for us to classify whether they correspond to the same bug or not, we decided to present the results in a more objective way by presenting in detail all of the inputs and effects of the bugs we found, instead of trying to count the number of distinct bugs. We then speculate about which of those inputs are likely to be triggering what could be considered the same bug.

Table 2 lists the various inputs that were flagged as positives by PIKE and that we confirmed to be caused by concurrency bugs. The table presents the requests that were concurrently executed in the test cases that triggered concurrency bugs together with the number of distinct thread schedules in which the program exhibited the incorrect behavior. Additionally, we also present information about the state and the output that were observed. Specifically, the table indicates whether the output of the concurrent execution matches the output of the sequential executions (O_A and O_B) and whether the state at the end of the concurrent execution matches the state at the end of either of the sequential executions (S_A and S_B).

Given the linearization algorithm, PIKE flags a concurrent execution as having triggered a concurrency bug if it cannot find a sequential execution (X) that produces both an output and a final state that match its own (i.e., that has O_X="Yes" and S_X="Yes"). We can see that all entries in Table 2 fail to meet this condition.

In addition to discrepancies in the output or the state of the different interleavings, we also found some cases where the execution of the application blocked, which might have been caused by deadlocks, and cases where the application crashed. We have not analyzed these cases, but they are less interesting from our standpoint since these potential bugs would also have been found by other tools like Chess [Musuvathi 2008], or tools that are designed to find deadlock bugs [Naik 2009].

In our experiments, we did not come across non-concurrency bugs, and this is not surprising for two reasons. First, we used inputs that were based on the existing regression tests contained in the MySQL source code, and therefore MySQL should have been previously tested for these or very similar inputs. Second, a non-concurrency bug, if triggered would have likely produced the same wrong results in all interleavings, regardless of the interleaving being sequential or not, and therefore our detector would not have flagged it.

One point we would like to highlight about these results is that we used a testing suite that has been applied repeatedly, albeit in a way that runs inputs sequentially. We postulate that it might be possible to be even more effective if we use a different set of inputs. The downside is that, because

External effect	Non-latent	Latent	Total
Error	2	0	2
Semantic	2	8	10

Table 1. Number of inputs found to trigger concurrency bugs according to latency and external effects.

we focused on what is not the latest version of MySQL, we found that some of the bugs have already been fixed, as we will detail next.

Next, we analyze in more detail the results for the two categories of bugs that our technique is aimed at: violations of the application semantics, and latent bugs. We further divide the first category into semantic bugs and error bugs, depending on whether the violation of the intended semantics corresponds to an incorrect but non-error reply, or a more explicit error.

5.2.1 Semantic bugs

Figure 3 illustrates a representative example of a semantic concurrency bug in MySQL that was found by our detector. In the figure the arrow indicates the interleaving that triggers the bug. This bug is triggered when the server receives a specific *SHOW TABLE* request and a *DROP* request concurrently as shown in Table 3. Figure 3 shows a simplified snippet of the source code that is involved in this concurrency bug. The first thread, while executing the *SHOW TABLE* request obtains a list of names of tables. According to the semantics of the database this returned list should contain the names of all the tables in the database whose name contains the string ”t1”. But, if before the first thread processes the list of tables names the second thread is able to execute the *remove_table()* function, the open table list becomes obsolete. This in turn means that when the first thread resumes execution it will try to call the *open_tables()* function with an argument that contains obsolete data and will not be able to access the table that was dropped. The result is that the second thread will return to the user a success message for the *DROP* request. However, the first thread will return an entry, for the now non-existent table, indicating that it exists but some of the entries will contain the value NULL.

We note that this particular instance of a semantic bug was eventually reported in the MySQL bug report database, and patched in a version that succeeded the one we tested. However, it is important to note that we did not use that information during the process of generating inputs.

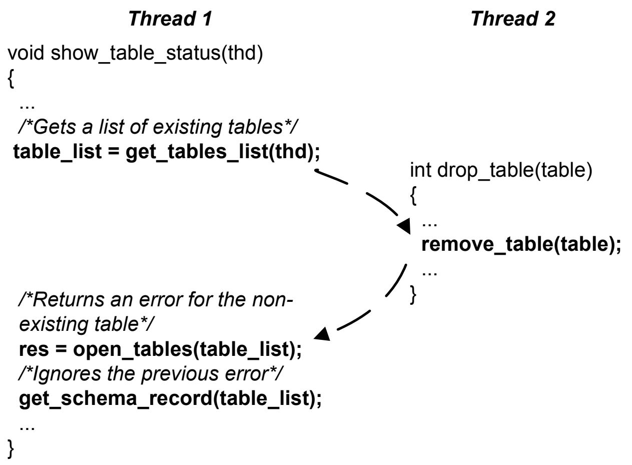
Other semantic bugs provided wrong results in even more subtle ways. For example, there were bugs where the application would simply provide wrong results based on stale data.

5.2.2 Error bugs

A sub-class of the semantic bugs that we found can be labeled as error bugs. We considered bugs to be error bugs if they manifest themselves by returning to the client an explicit error message, but an error message which is not appropriate given the requests that were executed. During our experiments we found two cases in which error concurrency bugs were triggered.

Request 1	SHOW TABLE STATUS LIKE ’t1’;
Request 2	DROP TABLE t1;

Table 3. Requests responsible for triggering the sample semantic bug


```
Thread 1
void show_table_status(thd)
{
 ...
 /*Gets a list of existing tables*/
 table_list = get_tables_list(thd);

 /*Returns an error for the non-
 existing table*/
 res = open_tables(table_list);
 /*Ignores the previous error*/
 get_schema_record(table_list);
 ...
}

Thread 2
int drop_table(table)
{
 ...
 remove_table(table);
 ...
}
```

Figure 3. Sample semantic bug

Table 4 presents the concurrent requests that were found to be responsible for one of the error bugs. This bug occurs when one of the threads attempts to execute a *CREATE LIKE* request, which is supposed to create a new and empty table with a schema that is identical to another existing table, and a specific *INSERT* request that copies data from the existing table into the new table. As illustrated in Figure 4, the first thread, while handling the *CREATE* request, first copies the definition file containing the schema for the existing table. According to the synchronization logic in MySQL, the second thread is allowed to execute the *INSERT* request even before the first thread creates the index file and data file. Because of this, while executing the *INSERT*, the second thread is unable to open the data file and returns an error to the user stating that the *data file* does not exist instead of either succeeding (by writing data) or returning a different error stating that the *table* does not exist.

This example illustrates an important point that error bugs can also be subtle and difficult to distinguish from a correct execution, despite the fact that they return an error. This is because very often an error message is a legitimate outcome of the operation, but the concurrent execution returns the wrong error message. Therefore, and unlike a situation where the application crashes or an assertion fails, we must know application-specific semantics to determine if an error reply is incorrect or not, and PIKE has proven to be effective in determining this.

Requests	EXs	Output			State		
		O_A	O_B	Effect	S_A	S_B	Latent

CREATE TABLE t2 LIKE t1; ‡
INSERT INTO t2 SELECT * FROM t1; ‡
	9	No	No	Error	No	Yes	Non-latent

INSERT INTO t3 VALUES (1,'1'),(2,'2');
SELECT DISTINCT t3.b FROM t3,t2,t1 WHERE t3.a=t1.b; †
	1	No	Yes	Semantic	No	No	Latent

CREATE TABLE t2 LIKE t1; ‡
INSERT INTO t2 SELECT * FROM t1; ‡
	2	No	No	Error	No	Yes	Non-latent

TRUNCATE TABLE t1;
SELECT * FROM t2;
	35	Yes	No	Semantic	No	No	Latent

INSERT INTO t1 (a) VALUES (10),(11),(12);
SELECT a FROM t1;
	2	No	Yes	Semantic	No	No	Latent

INSERT INTO t2 VALUES (2,0);
SELECT STRAIGHT_JOIN* FROM t1, t2 FORCE (PRIMARY); †
	3	Yes	No	Semantic	No	No	Latent

DROP TABLE t1;
SHOW TABLE STATUS LIKE 't1';
	238	No	No	Semantic	Yes	Yes	Non-latent

INSERT INTO t1 VALUES (1,1,"00:06:15"); †
SELECT a,SEC_TO_TIME(SUM(t)) FROM t1 GROUP a,b; †
	1	No	Yes	Semantic	No	No	Latent

CREATE TABLE t2 SELECT * FROM t1;
DROP TABLE t2;
	17	No	No	Semantic	No	No	Non-latent

INSERT INTO t1 (a) VALUES (REPEAT('a', 20));
SELECT LENGTH(a) FROM t1;
	3	No	Yes	Semantic	No	No	Latent

INSERT INTO t1 VALUES (80,'pendant');
SELECT COUNT(*) FROM t1 WHERE LIKE '%NDAN%'; †
	2	No	Yes	Semantic	No	No	Latent

OPTIMIZE TABLE t1;
DROP TABLE t1;
	25	Yes	No	Semantic	No	Yes	Latent

Table 2. Properties of the triggered concurrency bugs that PIKE found. The table presents the number of concurrent executions that were flagged as positive for each of the inputs (EXs). Additionally it indicates whether the output of the concurrent executions matched the output of the sequential executions (O_A and O_B) and similarly for the state of the sequential executions (S_A and S_B). (Requests marked with † have been simplified for presentation purposes, the two identical pairs of requests marked with ‡ operate on distinct states)

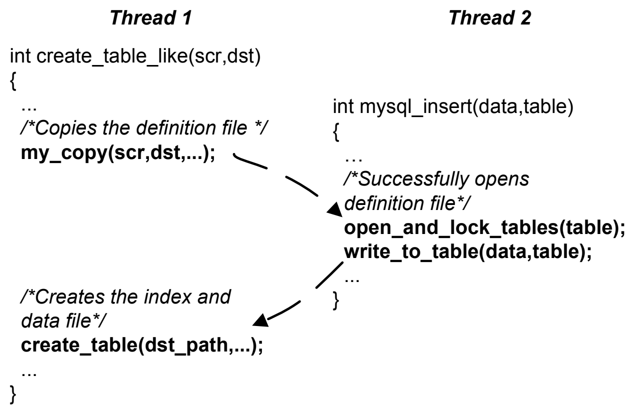
Request 1	CREATE TABLE t2 LIKE t1;
Request 2	INSERT INTO t2 SELECT * FROM t1;

Table 4. Requests responsible for triggering the sample error bug

5.2.3 Latent bugs

Surprisingly, PIKE was able to find eight different situations that triggered latent concurrency bugs. All of the latent bugs we found had the external effect of providing wrong results in subtle ways and involved the *query cache* structure. As we will describe in Section 5.3, we also found situations where the *binary log* appeared to contain an incorrect state, but we were not confident that these represented bugs (i.e., that the incorrect state would lead to incorrect behavior visible by users) and so we did not flag them as such.

As an example, one of the cases where a latent concurrency bug is triggered occurs when the requests in Table 5 are executed concurrently. The simplified source code relevant to this example is shown in Figure 5. While executing the *SELECT* request, the first thread opens the table, locks it, and in the process makes a copy for itself of the state of the table. The logic of the application allows the second thread to then concurrently insert entries at the logical end of the table. However, when the first thread resumes execution it will rely on its local (and now stale) copy of the state of that table to fetch data. In the process, the first thread will skip the newly inserted entry and provide the old results to the client without immediately violating the semantics of the



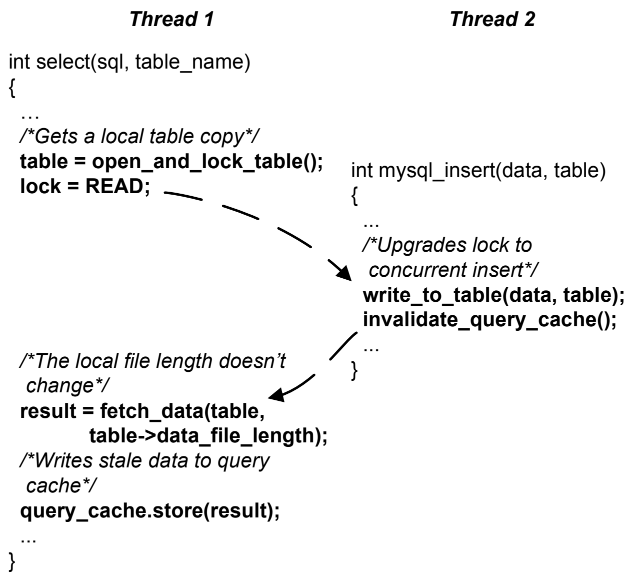
```
int create_table_like(scr,dst)
{
  ...
  /*Copies the definition file */
  my_copy(scr,dst,...);

                                  int mysql_insert(data,table)
                                  {
                                    ...
                                    /*Successfully opens
                                    definition file*/
                                    open_and_lock_tables(table);
                                    write_to_table(data,table);
                                    ...
                                  }
  /*Creates the index and
  data file*/
  create_table(dst_path,...);
  ...
}
```

Figure 4. Sample error bug

Request 1	SELECT a FROM t1;
Request 2	INSERT INTO t1 (a) VALUES (10), (11), (12);

Table 5. Requests responsible for triggering the sample latent bug



```
Thread 1

int select(sql, table_name)
{
  ...
  /*Gets a local table copy*/
  table = open_and_lock_table();
  lock = READ;

  /*The local file length doesn't
   change*/
  result = fetch_data(table,
           table->data_file_length);
  /*Writes stale data to query
   cache*/
  query_cache.store(result);
  ...
}
```

```
Thread 2

int mysql_insert(data, table)
{
  ...
  /*Upgrades lock to
   concurrent insert*/
  write_to_table(data, table);
  invalidate_query_cache();
  ...
}
```

Figure 5. Sample latent bug

application (i.e., the returned value would be consistent with the first thread having executed before the second thread). In this bug, the actual semantic violation arises from the fact that the first thread also stores the stale data which the second thread does not invalidate in the *query cache*. This means that a third thread could, at a later point in time, read the stale data from the *query cache* and expose it to the clients, violating the expected semantics of the application.

We saw the same pattern of latent bugs causing stale entries to be left in the query cache in other test cases, and we again stress that it is likely that some of the situations that triggered latent concurrency bugs could be triggering what could be considered the same bug. However, given the complexity of the application logic to both invalidate the *query cache* and to prevent certain specific concurrent requests from inserting simultaneously entries into the *query cache*, it is hard to state whether we are dealing with the same bugs objectively. Nevertheless it should be noted that the various cases that triggered latent bugs can be caused by very distinct types of requests, as can be seen in Table 2.

5.3 False positives

After the initial tests, approximately one third of the inputs generated potential false positives. Since this high fraction of false positives would make the analysis of the results impractical, we had to insert two filters to reduce the number of false positives which proved to be very effective. These filters allow testers to avoid false positives when the application deliberately violates linearizability.

The first filter we inserted was related to the table cache. Concurrent requests that try to open the same table concurrently will create distinct but identical entries in the table cache, whereas the same requests executing in sequence can reuse each other's entry. Therefore we inserted a filter that stated that the entries in the table cache for the linearized execution need to be contained in the concurrent one.

The second filter was related to the query cache, and the fact that MySQL sometimes conservatively decides not to cache entries in the query cache when two concurrent requests are executed, one of them is a query, and the other would invalidate the entry for that query in the query cache. In this case our filter says that the query cache entries in the concurrent execution must be contained in the set of entries in the linearization. Note that these may be considered performance bugs (or, at least, missed opportunities for a performance optimization), and this shows that PIKE might also be useful for analyzing and improving performance issues that may affect the application.

After inserting these two filters, the total number of false positives reported was 27. Of these, 22 are related to unexpected interactions between the framework and the application. In particular, some requests took a longer amount of time to complete, which in turn caused an execution timeout in our framework to expire. In other cases false positives were caused by non-determinism in the reply that we had not caught (e.g., calls to the current time or random number generation). A third type of false positives was caused by timeouts in the NFS volume in which our results were written, which affected the output. All of these types of false positives were reasonably easy for us to diagnose.

The remaining five false positives involved a more careful analysis. These were caused by *binary log* entries being reordered (i.e., MySQL would change some internal structures in one order and the *binary log* in another order). This turned out to be acceptable under some circumstances. Typically this happened with pairs of concurrent requests in which one of the requests executed an optimization or maintenance task (e.g., *OPTIMIZE* and *FLUSH* requests). The fact that these operations affect the performance but not the results implies that, when the *binary log* state is required (normally when a replica recovers from a fault), repeating these entries in the wrong order will not affect the output of the operations, but only the moment in the sequence of re-execution of these operations when the performance optimizations are performed.

6. Related Work

The goal of program verification is to guarantee that an implementation complies with its specification. Assuming the specification is correct (i.e., it specifies what the programmer intended), a verified program is guaranteed to be bug free. Model checking [Musuvathi 2004] is a promising technique that follows this approach by exhaustively exploring all possible states of the program. However, currently model checking has difficulty scaling to large programs.

Bug finding tools, on the other hand, although they do not guarantee that all bugs are found, are more scalable. Bug finding tools can be divided into static analysis and dynamic analysis tools depending, respectively, on whether they simply analyze the source code or actually execute the code. Static tools such as RacerX [Engler 2003] and others [Boyapati 2002, Naik 2006] have the advantage of not being limited in their analysis to the execution path determined by the input. On the other hand, dynamic analysis tools, since they actually run the code, have the advantage of having more information about the context of the execution and therefore can potentially achieve a higher accuracy (i.e., fewer false positives). PIKE is an example of a dynamic analysis tool, as are FastTrack [Flanagan 2009], LiteRace [Marino 2009] and Eraser [Savage 1997].

For testing to be successful, developers need to have good test cases. But given that manually generating tests is, in general, a tedious and difficult task, researchers have tried to automate this process by developing tools and methodologies that automatically generate test cases [Cadar 2008a;b, Godefroid 2005]. There have also been attempts to generate test cases specifically for databases [Microsoft, Mishra 2008]. However, automatic tools for test generation typically have difficulty scaling to large and complex applications.

Given the specifics of concurrency bugs, researchers have developed specific tools for handling this special class of bugs. One class of tools attempts to help programmers explore different thread interleavings. A different class of tools which also also specifically target concurrency bugs are data race detectors. We compare to each of these two classes in turn.

Typically, when a multi-threaded application runs natively, the operating system will tend to choose similar thread interleavings for different executions. To make testing more efficient, it is important to test a more diverse set of interleavings. One way of achieving this is by stress testing the application, possibly in combination with noise generators [Ben-Asher 2006]. A more sophisticated approach is to use custom schedulers that try to avoid redundant thread interleavings, prioritize some thread interleavings over others, and allow the programmer to replay a thread interleaving once it finds one that interests him (e.g., a thread interleaving that triggers bugs). Examples of tools that explore different thread interleavings in a smarter way are ConTest [Eytani 2007], CHESS [Musuvathi 2008] and PCT [Burckhardt 2010b]. However, these tools still rely on external mechanisms (e.g., assertion violations) to detect the occurrence of concurrency bugs. PIKE makes use of this approach, in particular the random scheduler algorithm of PCT, to explore different thread interleavings in a controlled way, but is complementary to them in that it enables new ways of finding bugs that do not rely on capturing exceptions or traditional assertion violations.

Some frameworks allow programmers to specify complex assertions for multi-threaded applications [Burnim 2009]. These typically enable programmers to specify invariants and to specify which parts of the code the invariants apply to. PIKE instead proposes an implicit correctness condition that relies on comparing the application behavior to the behavior during serializable executions.

A second class of tools are the data race detectors which can be roughly divided into two sub-classes depending on which algorithm they use. The first sub-class of data race detectors rely on the lockset algorithm [Savage 1997] to infer whether the programmer protected all accesses to a specific shared variable with a fixed lock. The second sub-class of data race detectors rely on the happens-before algorithm [Flanagan 2009, Marino 2009]. Recently, Erickson et al. have proposed a different data race detector that is not based on either of these algorithms, but is instead based on sampling and the use of breakpoints [Erickson 2010].

Like PIKE, data race detectors are also tools that can be useful for detecting concurrency bugs, however they have distinct features. First, these tools detect data races instead of directly detecting concurrency bugs. Since programs often contain benign data races, simply detecting data races easily leads to false positives. Furthermore the absence of data races is not a guarantee of correct synchronization [Artho 2003, Lu 2006], and hence false negatives can result. Another difference is that race detectors typically operate at the lower-level of individual memory accesses. In contrast, PIKE analyzes the actual output of the application as well as a high-level digest of the state, potentially uncovering bugs that are not triggered by low-level data races and also facilitating the process of inspecting the results.

In order to reduce the number of false positives in data race finding tools and thus reduce the burden on testers, researchers have developed heuristics. By using heuristics some systems attempt to identify scenarios that frequently lead to false positives. DataCollider [Erickson 2010], for example, tries to detect benign data races caused by counters and accesses to different bits of the same variable. One approach is to use heuristics that rely on looking at the instructions at or near the problematic accesses or on manually whitelisting variables. The disadvantage of this approach is that it also increases the risk of missing erroneous data races. Another interesting approach to distinguish erroneous data races from benign data races relies on replaying the execution [Narayanasamy 2007]. It relies on trying to trigger the opposite outcome of the data race and then comparing the low-level results obtained with both data race outcomes. This approach, however, still aims at finding low-level data races.

There have been prior approaches for checking the linearizability of code to improve robustness [Burckhardt

2010a, Vafeiadis 2010, Vechev 2009, Xu 2005]. We differ from these approaches in two ways. First, they typically ignore the internal state of the application, which is important for the detection of latent bugs. Second, they check for the atomicity of smaller sections of code such as code blocks or library calls, which poses fewer challenges than testing the linearizability of large server applications.

AVIO [Lu 2006] detects atomicity violations at the level of individual memory accesses. AVIO achieves this by learning from a large set of runs (which are assumed to be correct) the valid memory access patterns (e.g., when are two consecutive accesses from a thread allowed to be interleaved by an access from another thread). AVIO shares our goal of attempting to find concurrency bugs without relying on finding data races, but in contrast AVIO works at a low-level and relies on training.

Finally, an entirely different approach for dealing with concurrency bugs is by using tools that prevent or make it less likely for programmers to make mistakes. One such approach is to use special programming languages [Vaziri 2006], while another is to use special hardware or frameworks such as transactional memory [Herlihy 1993, Shavit 1995].

7. Conclusion

This paper presented PIKE, a tool for testing concurrent applications. PIKE is able to find two particularly challenging types of bugs: semantic bugs and latent bugs. Semantic bugs generate subtle deviations from the expected behavior of the application, while latent bugs silently corrupt internal data structures, and manifest themselves to clients possibly long after the requests that triggered the bug are executed. PIKE detects these two types of bugs by testing if the application obeys linearizable semantics, both in terms of its outputs and its internal state. Our experience in applying PIKE to find concurrency bugs in MySQL was a positive one. We found that it was simple to write the necessary annotations to capture an abstract view of the service state, and that it was easy to make the number of false positives tractable by writing simple filtering rules for common violations of linearizability at the level of the application state. More importantly, we were able to find several semantic and latent concurrency bugs in a stable version of MySQL. Currently, we are applying PIKE to the current development release, and we are analyzing the potential crash and deadlock bugs we have found with our current test suite.

Acknowledgments

We are grateful for the feedback provided by the anonymous reviewers and for the help provided by our shepherd, Leonid Ryzhyk. Pedro Fonseca was supported by a grant provided by FCT.

References

[Artho 2003] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[Ben-Asher 2006] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *Proc. of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 37–40, 2006.

[Bonomi 2006] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. *Lecture Notes in Computer Science*, 4168:684–695, 2006.

[Boyapati 2002] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, 2002.

[Burckhardt 2010a] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. *SIGPLAN Not.*, 45(6):330–340, 2010. ISSN 0362-1340.

[Burckhardt 2010b] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Comput. Archit. News*, 38(1):167–178, 2010. ISSN 0163-5964.

[Burnim 2009] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 3–12, 2009.

[Cadar 2008a] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating System Design and Implementation (OSDI)*, pages 209–224, 2008.

[Cadar 2008b] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2): 1–38, 2008. ISSN 1094-9224.

[Engler 2003] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, 37(5):237–252, 2003. ISSN 0163-5980.

[Erickson 2010] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proc. of Operating System Design and Implementation (OSDI)*, pages 1–16, 2010.

[Eytani 2007] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Conc. & Comp.: Practice & Experience*, pages 267–279, 2007.

[Flanagan 2009] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, 2009. ISSN 0362-1340.

[Fonseca 2010] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*, pages 221–230, 2010.

[Godefroid 2005] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005. ISSN 0362-1340.

[Herlihy 1993] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[Herlihy 1990] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925.

[Lu 2008] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008. ISSN 0163-5964.

[Lu 2006] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2006.

[Marino 2009] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. of Programming Languages Design and Implementation (PLDI)*, pages 134–143, 2009.

[Microsoft] Microsoft. Generating test data for databases by using data generators. `http://msdn.microsoft.com/en-us/library/dd193262.aspx`.

[Mishra 2008] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 499–510, 2008.

[Musuvathi 2004] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. of Networked Systems Design and Implementation (NSDI)*, pages 155–168, 2004.

[Musuvathi 2008] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of Operating System Design and Implementation (OSDI)*, pages 267–280, 2008.

[Naik 2006] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proc. of Programming Languages Design and Implementation (PLDI)*, pages 308–319, 2006.

[Naik 2009] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 386–396, 2009.

[Narayanasamy 2007] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. of Programming Languages Design and Implementation (PLDI)*, pages 22–31, 2007.

[Oracle a] Oracle. MySQL :: Market share. `http://www.mysql.com/why-mysql/marketshare/`.

[Oracle b] Oracle. Storage engine poll. `http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html`.

[Oracle c] Oracle. The MyISAM storage engine. `http://dev.mysql.com/doc/refman/5.0/en/myisam-storage-engine.html`.

[Sahoo 2009] Swarup K Sahoo, John Criswell, and Vikram S. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. Tech. Report 2142/13697, University of Illinois, 2009.

[Savage 1997] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, 1997. ISSN 0163-5980.

[Shavit 1995] Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. of Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[Vafeiadis 2010] Viktor Vafeiadis. Automatically proving linearizability. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 450–464, 2010.

[Vaziri 2006] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proc. on Principles of Programming Languages (POPL)*, pages 334–345, 2006.

[Vechev 2009] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Proc. on SPIN Workshop on Model Checking Software (SPIN)*, pages 261–278, 2009.

[Xiong 2010] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proc. of Operating System Design and Implementation (OSDI)*, pages 1–8, 2010.

[Xu 2005] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005. ISSN 0362-1340.