Tolerating Concurrency Bugs Using Transactions as Lifeguards

Jie Yu and Satish Narayanasamy
University of Michigan, Ann Arbor
{*jieyu,nsatish*}*@umich.edu*

***Abstract*—Parallel programming is hard, because it is impractical to test all possible thread interleavings. One promising approach to improve a multi-threaded program's reliability is to constrain a production run's thread interleavings in such a way that untested interleavings are avoided as much as possible. Such an approach would avoid hard-to-test rare thread interleavings in production runs, and thereby improve correctness. However, a key challenge in realizing this goal is in determining thread interleaving constraints from the tested correct interleavings, and enforcing them efficiently in production runs.**

In this paper, we propose a new method to determine thread interleaving constraints from the tested interleavings in the form of lifeguard transactions (LifeTxes). An untested code region initially is contained in a single LifeTx. As the code region is tested over more thread interleavings, its original LifeTx is automatically split into multiple smaller LifeTxes so that the newly tested interleavings are permitted in production runs.

To efficiently enforce LifeTx constraints in production runs, we propose a hardware design similar to the eager conflict detection capability that exist in a conventional hardware transactional memory (TM) systems, but without the need for versioning, rollback and unbounded TM support. We show that 11 out of 14 real concurrency bugs in programs like Apache, MySQL and Mozilla could be avoided using the proposed approach for a negligible performance overhead.

***Keywords*-Parallel Programming, Concurrency Bugs, Software Reliability**

I. INTRODUCTION

Parallel programming is inherently harder than von Neumann style single-threaded programming. The number of possible states at a program statement exponentially increases with the number of threads executed, as the memory operations in a thread could interleave with the memory operations in the other threads in many different orders. Understanding, testing and verifying the correctness of all possible thread interleavings is impractical. Programmers tend to test only a small fraction of all possible legal thread interleavings in an application before shipping it to customers. The remaining untested interleavings could cause production runs to fail due to concurrency bugs.

Yu and Narayanasamy [38] observed that in a well tested program, a programmer is likely to have tested at least the interleavings that manifest frequently when the program is executed. But many rare interleavings are likely to remain untested. Such rare untested interleavings tend to be the common cause for a majority of concurrency bugs that manifest at the production site. Given this, a parallel runtime system that avoids untested interleavings by biasing the runtime thread schedule to select a tested interleaving, whenever possible, could potentially avoid a majority of concurrency bugs from manifesting at the customer site. The performance cost of disallowing rare untested interleavings would not be significant, because though they are many in number, they are only likely to manifest infrequently (if they do manifest frequently, then they are likely to have been tested).

The key challenge in realizing an interleaving constrained parallel runtime system is in devising the right interleaving constraint that can be learned from the test runs and communicated to the runtime system. The interleaving constraint should be such that it disallows untested interleavings at runtime and avoids a majority of concurrency bugs, but at the same time it does not unnecessarily restrict parallelism by disallowing correct thread interleavings in production runs. Predecessor Set (PSet) constraint [38] was recently proposed to serve this purpose. However, PSet interleaving constraints require a complex hardware support to enforce them efficiently in production runs. Also, PSet constraints cannot avoid concurrency bugs due to incorrect interleaving of memory accesses to different locations such as the multi-variable atomicity violations [16].

This paper presents a new interleaving constraint called Lifeguard Transaction (LifeTx). Lifeguard transactions are similar to the programmer specified transactions [12], [14] in that it instructs the runtime to execute them in a serializable order. The difference is that LifeTxes are automatically derived based on interleavings observed during testing. When enforced, they are likely to avoid concurrency bugs. But, the runtime may also choose not to enforce a LifeTx constraint either because the performance is being adversely affected or to ensure forward progress. We exploit this relaxed requirement of LifeTx constraints to significantly reduce the hardware support required to enforce them.

We describe a profiling algorithm to determine LifeTxes from all the tested correct executions. Before testing a program, a thread's main function is contained in a LifeTx. Of course, this is overly constrained as a thread's entire execution needs to be serializable with respect to all the other threads. But, when a programmer tests a new interleaving for which there is no serializable execution that satisfies the current set of LifeTxes learned till that test run, we split one of the existing LifeTx such that the resulting LifeTxes are serializable for the newly tested interleaving. Thus, the newly tested interleaving would be allowed in future production runs. A programmer would test as many interleavings in a program as practically possible. The more a programmers tests, smaller and less constrained the LifeTxes will be. Once

the testing is done, the LifeTx constraints are encoded in the program binary and shipped to the production site.

To enforce LifeTx constraints in production runs we propose a simple but efficient hardware support similar to the conflict detection mechanism in a hardware transactional memory (HTM) system [14]. Conflicts between concurrent LifeTxes are eagerly detected by tracking the cache blocks accessed by a LifeTx and monitoring the coherence messages. Since the runtime is not obligated to enforce the LifeTx constraint, we avoid the complexity of versioning, rollback and unbounded region support required in TM systems. Instead, we simply stall the coherence reply on detecting a conflict till one of the conflicting LifeTx finishes its execution or the stall time exceeds a predefined threshold. The threshold can be configured by the end user to make a trade-off between performance and reliability. We show that a majority of LifeTx constraints including those that encapsulate buggy code regions can be enforced by simply stalling coherence replies on detecting a conflict. The constraint violations detected during production runs and beta-testing could be logged and communicated back to the developer so that the programmer could test those interleavings and relax LifeTx constraints for future executions.

We implemented a Pin tool [22] to profile the test runs and determine LifeTxes. We studied a set of applications that includes Apache, MySQL, Parsec, and a few micro-kernels. We tested these applications as much as we can using the regression test suits and/or randomized input. Insufficient testing could result in unreasonably large LifeTxes. But, we observed that even with our less than industrial strength testing effort, LifeTxes are on the order of only a few hundred instructions in length. To study the performance impact and bug avoidance capability, we modeled hardware support for runtime conflict detection and conflict avoidance support using Simics [23]. Thus, the simulated environment is significantly different from the test environment. Yet, we find that, in the worst case, for one of the MySQL's execution, we detected only 178 constraint violations during an execution of about 2.1 billion instructions. Resolving those constraint violations by stalling coherence replies incurred only negligible performance overhead. We also analyzed 14 real bugs in our benchmark suite, out of which 12 were atomicity violations. Out of the 12 atomicity violation bugs, 11 bugs (two of them were multi-variable atomicity violation bugs) were successfully avoided by enforcing LifeTx constraints learned during testing.

LifeTx constraints are useful for tolerating concurrency bugs in programs written using traditional form of synchronization operations such as locks, barriers, etc. We believe that they will also be useful for programs written using programmer specified transactions as well. Because, even when programmers use transactions, they could still introduce atomicity bugs by not encapsulating all the instructions that need to be atomic in a single transaction. LifeTx constraints can avoid such atomicity bugs.

In addition to helping us avoid concurrency bugs in production systems, LifeTx constraints derived from tested

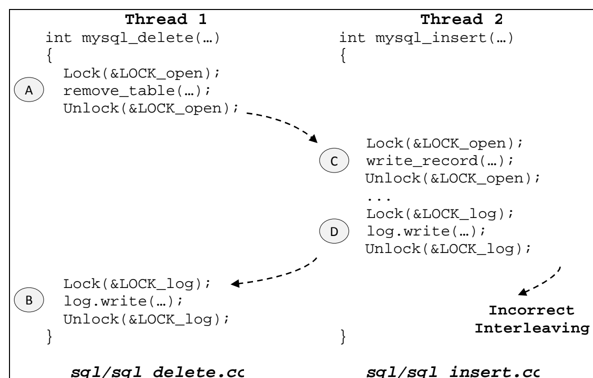

Figure 1. A multi-variable atomicity violation bug in MySQL.

executed could also help improve the testing process of multithreaded programs. One common practice is to blindly stress test as many rare interleavings as possible. Instead, programmers and automatic testing tools can prioritize their efforts on exposing more interleavings for code regions contained in the larger LifeTxes. Also, LifeTx constraint violations logged during production runs could help programmers prioritize their test efforts and also determine the root cause of a program crash.

II. MOTIVATION

Figure 1 shows a real multi-variable atomicity violation bug in MySQL. Instructions `A` and `B` need to execute atomically to produce the correct log record for the `mysql_delete` operation. However, the synchronization operations in the code are not sufficient to prevent another thread from incorrectly overwriting the log before `Thread 1` could write its log. This could causes an inconsistency in the log file, which could potentially lead to a crash when the database system tries to recovery using the log file. Notice that there does exist a happens-before relation between all the operations to the table and the log due to the synchronization operations. Thus a happens-before data-race detector [7] cannot detect and avoid this bug. Furthermore, the atomicity violation involves multiple variables in the program, making it impossible to be detected and avoided by many single-variable atomicity violation detectors [18], [21].

III. ALGORITHM FOR DETERMINING LIFETXES

In this section we define LifeTx interleaving constraints and describe an online profiling algorithm for automatically determining those constraints from correct test runs.

A. Lifeguard Transactions (LifeTxes) and Profiling Algorithm Overview

Our goal is to tolerate concurrency bugs in production system by constraining the runtime system to execute tested thread interleavings as much as possible instead of allowing it to execute any legal interleavings permissible by user specified synchronizations. To achieve this, we require techniques to determine interleaving constraints from test runs, encode them in a program binary and enforce them at

runtime. By enforcing these constraints during production runs, untested interleavings would be avoided.

We propose Lifeguard Transaction (LifeTx) interleaving constraint. Every instruction in a program is part of some LifeTx. A LifeTx constraint is similar to a programmer specified transaction [14] and is defined for a static code region (a code region is a consecutive sequence of instructions in the source program). An execution is said to satisfy the LifeTx constraints, if there exists an equivalent execution where the LifeTx protected code regions are executed in a serial total order.

We learn LifeTx constraints from correct test runs. We start from a conservative set of LifeTxes assuming that the entire execution of each thread is part of one LifeTx. Clearly, this initial constraint is too strict. As a programmer tests an execution for which there is no serializable execution of LifeTx protected code regions, we split the current set of LifeTxes. This is done by introducing what we call a *cutpoint* in the source program. During an execution, a cutpoint serves to terminate the previous LifeTx and then start a new LifeTx. Thus, a set of cutpoints represents the LifeTx constraints. They are encoded in the binary. As a programmer tests more and more interleavings, we progressively construct smaller and smaller LifeTxes such that the final set of LifeTxes are *conflict-serializable* for all the tested interleavings. The runtime, which we describe in Section IV, would then try to enforce a serializable order between the LifeTxes during a production run.

In fact, tested interleavings are encoded in the set of learnt LifeTx constraints. By successfully enforcing these LifeTx constraints during production runs, we can ensure that if the execution of a code region was atomic in all the test executions, then it will be atomic in production runs, preventing untested unserializable interleaving with respect to this code region from manifesting. Consider the atomicity violation bug shown in Figure 1. In any correct execution, code regions `A` and `B` would have executed atomically and therefore will be part of the same LifeTx. Even though, the programmer has not correctly synchronized the critical sections, our LifeTx constraints would automatically ensure that property and thereby avoid a potential concurrency bug in production runs.

The algorithm can be divided into two major parts, checking conflict-serializability for the current set of LifeTxes and splitting LifeTxes when a conflict is detected, which will be addressed in the subsequent sections.

B. Checking Conflict-Serializability for LifeTxes

To determine whether a test run satisfies the set of LifeTx constraints learnt until that test run, we check conflict-serializability for these LifeTxes. To check conflict-serializability, we construct a directed graph called *conflict-serializability graph*. Each node in the graph represents a LifeTx. When a new LifeTx begins its execution during testing, a new node is added to the graph. All the nodes executed by a thread are connected according to the execution order. A conflict edge is added between two LifeTxes if they executed *conflicting* memory operations. Two concurrent memory operations executed in different threads are said to conflict if they accessed the same memory location and at least one of them is a write. Duplicated edges are not allowed. A conflict-serializability violation is detected when we detect a cycle while adding a new edge to the conflict-serializability graph [11]. The cycle indicates that the tested execution is not *conflict-serializable* with respect to the LifeTxes learnt till that point.

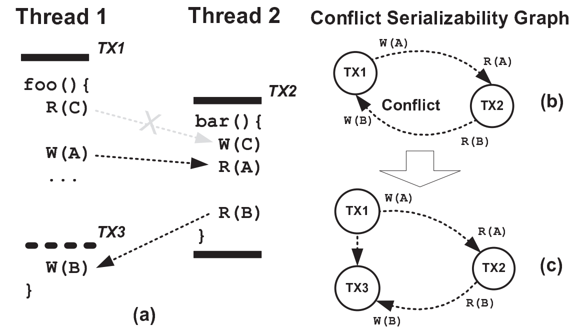

Figure 2. A Conflict-serializability violation.

Figure 2 shows a thread schedule and the corresponding conflict-serializability graph. `Thread1` and `Thread2` are currently executing LifeTxes `T1` and `T2` respectively. `W(X)` represents a write to memory location `X`, and `R(X)` represents a read to memory location `X`. An conflict-serializability violation is detected at the point of `W(B)` in `Thread1`, because a cycle is detected in the conflict-serializability graph.

Conflict-serializability violation checks only report a violation when there is an unserializable memory accesses. Whereas, other serializability checks such as strong strict two-phase locking do not guarantee this. We want our profiling algorithm to be conservative – do not split a LifeTx unless an real unserializable interleaving is observed during testing.

To detect conflicting memory operations, for each memory location, we maintain a data structure to store the latest write operation and the latest read operations for each thread to that location. Each time a conflict edge is added to the conflict-serializability graph, we check whether a cycle exists in the graph. Since the complexity of cycle detection is linear to the number of nodes in the graph, it is impractical to keep all the nodes in the graph. We employ an optimization [9] that removes those nodes that do not have incoming edges and the corresponding LifeTxes have terminated, because they cannot be part of any future cycle.

C. Splitting LifeTxes On a Conflict

On detecting a conflict-serializability violation for the current set of LifeTxes during a test run, our tool decides to split one of the conflicting LifeTx by introducing a cutpoint into that LifeTx. The LifeTx that executed the most recently conflicting memory operation is chosen as the victim for the split.

In order to decide the location of the cutpoint in the source program, we need information about the memory

accesses involved in the conflict. This is obtained by tracking conflicting memory access information on every conflict edge in the graph used for conflict-serializability check. It is possible that we may detect multiple conflicts between two LifeTxes during an execution. Since we do not allow duplicated edges, we choose to maintain the most recent conflict. This could help us pinpoint the location of the cutpoint more precisely when a conflict-serializability violation is detected. For example, in figure 2, conflict edge `R(C)` $\rightarrow$ `W(C)` is discarded when conflict `W(A)` $\rightarrow$ `R(A)` is detected.

The LifeTx chosen for the split will contain at least two memory operations that participate in the conflict-serializability violation. Otherwise, there cannot be a cycle in the graph. For example, in Figure 2, `Tx1` is the LifeTx that executed the most recent conflicting memory operation `W(B)`. It also contains another memory operation `W(B)` that participates in the conflict-serializability violation. The cutpoint to split `Tx1` can be introduced anywhere between these two memory accesses. We choose to always insert the cutpoint just before the last conflicting memory operation `W(B)`. After a cutpoint is inserted, we also split the corresponding node of the LifeTx in the conflict-serializability graph by terminating the current LifeTx and introducing a new LifeTx. Consider the example in figure 2, figure 2(b) shows the conflict-serializability graph before splitting, and figure 2(c) shows the graph after splitting.

Inserting a cutpoint in effect relaxes the atomicity constraints between all the memory operations that happen-before the cutpoint and all the memory operations that happen-after the cutpoint in a LifeTx. Ideally, we should relax the atomicity constraint only for memory operations involved in the conflict instead of inserting a cutpoint. However, that would require complex runtime hardware support for enforcing them. We leave such a design for future work.

Splitting LifeTxes Spanning Multiple Code Levels. The LifeTx chosen for splitting could span across multiple *semantic-segments*, which requires special handling while inserting a cutpoint. All the instructions executed in a function are considered to be part of a semantic-segment. Similarly, all the instructions of a loop are considered to part of another semantic-segment. Instructions of an iteration of a loop together constitute a different semantic-segment. Figure 3 shows a thread interleaving where the LifeTx `TX1` spans across two semantic-segments, functions `foo()` and `bar()`. As discussed before, a cutpoint could be inserted anywhere between conflicting accesses `R(A)` and `W(C)`. Our algorithm picks the outermost semantic-segment that contains the conflicting access, and inserts a cutpoint at the point in the source program where the next semantic-segment starts. In Figure 3, the cutpoint is inserted just before the function call `bar()`. Inserting a cutpoint inside inner semantic-segments such as `bar()` are more likely to allow interleavings that are not tested. For example, if we insert a cutpoint inside `bar` to resolve the conflict in our example, then when `bar()` is invoked from a different function, the LifeTx executed at that time would be

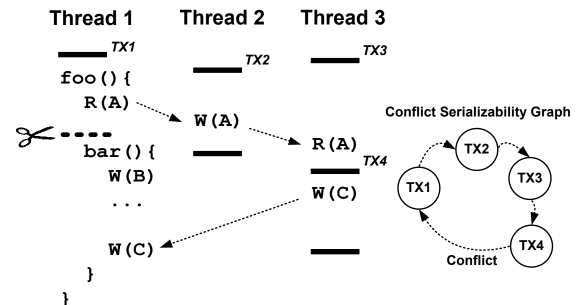

Figure 3. A conflict-serializability violation detected across multiple semantic-segments.

terminated. This could prevent us from avoiding concurrency bugs. In effect, our heuristic for inserting cutpoints biases against inserting context insensitive cutpoints. The algorithm for loops and loop-iteration semantic-segments is similar.

To track semantic-segments, we instrument the entries and exits for each semantic-segment in the program. For example, we instrument each function call and return. For loops and iterations, we statically identify loop entries, exits and back edges using goose tool [1], and then instrument them. To decide which semantic-segment is the outermost one that contains the conflicting accesses, we assign a thread-local counter (monotonically increasing) for each memory operation executed by the thread, and maintain a per-thread stack to track the value of the counter when the thread enters a semantic-segment. By comparing the counter values of the conflicting memory operations and the counter values stored in the stack, we can easily identify the outermost semantic-segment that contains conflicting memory operations.

D. Practical Issues

In this part, we discuss a few practical issues when applying our LifeTx inference algorithm to a real world multi-threaded program that is written using explicit synchronizations.

1) Relaxing Conflict Detection For Synchronization Functions: As we discussed in Section III-C, inserting a cutpoint in effect relaxes the serializability constraints between all memory accesses before and after the cutpoint. Ideally, we should relax the constraint only for the memory accesses that are involved in the conflict. Such an approximation will become problematic, especially when we detect conflicts for memory accesses that are inside synchronization functions. Figure 4 illustrates the problem. In the example, two threads are contending for the same lock. Two conflicts will be detected according to our LifeTx inference algorithm. When the same lock functions are called by a different function F, the cutpoint inserted would terminate and restart the transaction containing F as well. As a result, some concurrency bugs like the one discussed in Figure 1 may escape.

This problem actually exists for any function, not just for synchronization functions we showed. However, using a general way to solve the problem is very difficult. Instead, we choose to address this problem for code regions that matter the most – synchronization functions. This is because

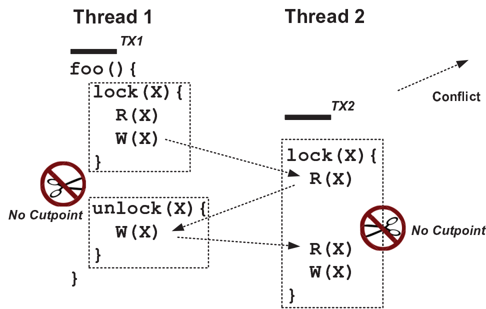

Figure 4. Conflicts due to memory operations executed in synchronization functions.

synchronization calls are interleaved heavily with other synchronization calls in remote threads.

We disable conflict detection for synchronization calls both during testing and also in production runs. This is similar to the escape actions described in [34]. Thus, we ignore the unserializable dependencies between shared-memory operations inside synchronization functions. However, we determine the happens-before relation specified by the synchronization calls, and treat a happens-before relation as a dependency between transactions during testing. That is, a conflict edge is added for each happens-before relation in the conflict-serializability graph that we construct during testing. This is necessary to allow tested interleavings between code regions synchronized using traditional synchronization operations during production runs.

2) Optimizations for Reducing Runtime Conflict Detection Complexity: To reduce the hardware support required for conflict detection in production runs we seek to limit the number of memory locations accessed by a LifeTx. For this, we employ a simple heuristic that profiles the loops that have a trip count greater than a threshold, and introduce a cutpoint before the back edge of those loops. The intuition here is that it is unlikely that a program would require atomicity property across a loop that iterates for a long time.

Speculating past certain system calls such as network I/O could be difficult for the runtime system. We introduce a cutpoint before such difficult to speculate system calls, so that the runtime system need not support speculation past those system calls. However, this optimization is not necessary for the runtime system which does not require speculation (Section IV-A).

3) Mapping Cutpoints to Source Code: As cutpoints are gathered for a program, we map them back to the statement in the source code. If a programmer makes a simple change to the program, then all the cutpoints gathered through testing will still be valid.

E. Discussion and Limitations

1) Testing Correctness: Programmers have to ensure that the test runs are correct. This could be done by verifying the program output or by checking the test runs using traditional dynamic bug detection tools.

2) Input dependency: We use very different input for testing and measuring performance to evaluate whether our approach is sensitive to program input. Our results (Section V) indicate that LifeTxes could be learnt in a few test runs and the overhead observed during a simulated production run is negligible. However, though rare, a production run could encounter frequent conflicts for some input. In such a scenario, we could turn off the LifeTx protection for performance, and the conflicting LifeTxes could be logged and communicated to the developer. These logs could assist programmers prioritize their testing effort. It is rare for an execution with incorrect interleaving to produce a correct answer. But if such a test input exist, then we may incorrectly relax constraint by splitting a LifeTx. We did not encounter this scenario in our experiments.

3) Context: LifeTx constraints are context insensitive. As a result, some concurrency bugs might not be avoided due to the lack of context information such as MySQL-4 in Table II. We could associate context information (e.g. calling stack) with LifeTxes to address this problem, but it will require significantly more test runs to learn and require complex runtime system support.

IV. Runtime Support for LifeTxes

We now discuss the runtime support for enforcing LifeTxes. The goal of the runtime system is to ensure that the execution of LifeTxes is conflict serializable with respect to each other during production runs. We implemented a runtime detection and avoidance support using Pin, but it slows down an execution by several orders of magnitude and therefore are unrealistic for use in a production system. To efficiently enforce LifeTxes at runtime, architectural support is a must. In this section, we first present LifeTx-Stall, an architectural design that considers hardware complexity as a first-order constraint (Section IV-A). It is simple and lightweight while still very effective in enforcing LifeTxes. We also discuss an ideal design, which we later use for comparison. In Section V we evaluate the performance, conflict detection and bug avoidance capability of the proposed architectural design using our Simics based simulation model.

A. LifeTx-Stall Design

LifeTx-Stall design consider hardware complexity as a first-order constraint. It is actually a simplified Hardware Transactional Memory (HTM) system *without* speculation and unbounded TM support. Each LifeTx is treated as a hardware transaction. On encountering a cutpoint, the processor commits the current LifeTx and starts a new LifeTx.

To enforce LifeTxes, we need to check whether the LifeTxes executed at runtime are conflict serializable. However, hardware support for checking conflict-serializability violations is fairly complex [30]. Therefore, we choose to detects conflicts for LifeTxes eagerly [37]. It tracks the cache blocks read and written by a LifeTx and detects a conflict by monitoring coherence requests. Although it could unnecessarily report a conflict between two LifeTxes while they are actually conflict serializable, it simplifies the design

a lot. Our results (Section V-E) also indicate that the number of extra conflicts are acceptable.

To resolve a conflict, instead of using the traditional recovery mechanism used in HTM designs, we propose to use a simpler scheme. The main idea is that a processor would delay the coherence reply when the requester has a conflict with the LifeTx currently running on it until the current LifeTx commits. To ensure forward progress, we assign a threshold for the number of cycles to wait. Such a scheme does not guarantee to enforce all the LifeTx constraints, therefore, it is a best effort scheme. We have the luxury to design such a simple system because not all LifeTx constraints need to be enforced (unlike HTM systems). Such a design avoids the need for speculation and rollback support. As a result, neither version management nor checkpointing is needed, which avoids issues in traditional HTM systems such as speculative I/O buffering. Even with such a simple design, our results (Section V-E) indicate that it is effective in avoiding most of the conflicts.

LifeTx-Stall assumes a snoop bus based MESI cache coherence protocol. There are two major functionalities that LifeTx-Stall needs to support. One is to detect runtime LifeTx conflicts. The other is to resolve the detected LifeTx conflicts so that the resultant execution is conflict serializable with respect to both LifeTxes. The following sections discusses these two functionalities in detail.

1) Detecting LifeTx Conflicts: **Maintaining Transactional Meta-data**. LifeTx-Stall needs to keep track of the memory locations accessed by a LifeTx so as to detect conflicts. Each private cache block is extended with two additional bits (TX-READ and TX-WRITE), which we call *transactional meta-data*, indicating whether the cache block is read and written by the current LifeTx. A processor clears all the meta-data in its private cache when it commits a LifeTx. Also, when a processor timeouts waiting for a conflict to resolve, all the processors clear the meta-data in their private caches. LifeTx-Stall maintains transactional meta-data in private caches. If the cache line is evicted from the private caches, the transactional information will be lost. To alleviate this problem, we assume a small victim cache along with each private cache, which similar to the one used in many processor implementations for improving performance by indirectly increasing the associativity of caches. The replacement policy for the victim cache is LRU based, but with the exception that it always prioritizes to hold cache blocks with transactional meta-data.

Monitoring Coherence Actions. LifeTx-Stall assumes a bus based design. LifeTx-Stall detects a read-write or write-write conflict by monitoring coherence requests broadcasted on the bus and by checking its private transactional meta-data. On detecting a conflict, a processor would set a dedicated wired-OR line (similar to the wired-OR line sued for detecting whether a shared copy exists or not). This is useful in efficiently resolving the conflict using the stall mechanism described in Section IV-A2.

Relaxed Memory Accesses. LifeTx-Stall does not update the transactional meta-data for synchronization accesses (discussed in Section III-D1) or memory accesses from OS. This is because we are not interested in the conflicts that are caused by the relaxed synchronization accesses or OS accesses. Those accesses are called relaxed memory accesses. Although LifeTx-Stall does not update transactional meta-data for relaxed memory accesses, it still needs to check conflicts for them. In other word, a relaxed memory access can cause a conflict and a resultant processor stall. This is because a cache block might be accessed by both regular and relaxed memory accesses. A normal memory access which indeed conflicts with the remote processors may not be able to trigger a bus transaction if the read or write permission of the cache block has already been obtained by a precedent relaxed memory access to the same cache block. The simple policy we employed may cause unnecessary stalling (false positives). However, our results indicate that the number of conflicts detected at runtime is still negligible. This problem can also be addressed if the compiler can separate the data that is accessed by normal memory accesses from the data that is accessed by synchronization accesses.

Granularity. Instead of detecting conflicts at the granularity of a cache block, LifeTx-Stall can be extended to support word level conflict detection, which would avoid false conflicts. To achieve this, we need to maintain transactional meta-data for each word in a block, and associate word offset information with coherence requests. However, a few issues arise in such a design. First, the transactional meta-data of a word in a cache block may get lost when the cache block is invalidated due to a remote write to a different word of the same cache block, which would affect the bug avoidance capability of our system. For example, a processor `P1` and a processor `P2` both have a shared copy of a cache block `B` initially. `P1` reads the first word `W1` in `B`, causing the TX-READ bit of `W1` be set. Then, `P2` writes to another word `W2` in `B`. Since the conflicts are detected at the granularity of word, no conflict would be detected. At this time, `P1` invalidates its copy of `B` to service the write request, and in the process loses the TX-READ bit for `W1`. Second, the permission of a cache block could migrate to a different processor causing a potential silent conflicting access. Consider the following example: a read to a word in a block does not generate a coherence request since the read permission is obtained by a previous read to the same block but not the same word. These problems will not occur if we detect conflicts at the granularity of block. Nevertheless, our results show that the bug avoidance capability of the word based implementation is not significant (Section V-D2). Furthermore, we observe that detecting conflicts at the granularity of word can significantly reduce the number of false conflicts at runtime (Section V-E1). Therefore, we choose to use word level conflict detection in our LifeTx-Stall design.

2) Resolving LifeTx Conflicts: A processor detects a conflict by monitoring the coherence requests and checking its transactional data. On detecting a conflict, a processor sets a dedicated wired-OR line that can be read by all the processors. The processor that initiated the coherence

request reads the wired-OR line and determines that it needs to stall and re-issue the same request after a specified time period has elapsed. The wired-OR line also helps other processors with a valid read copy to not invalidate their cache block in response to an invalidation request that got stalled. The stalled processor would re-issue the coherence request after a specified time period has elapsed. If after requests fail to get a response after a specified number of attempts, the processor issues a special coherence request that cannot be ignored. Thereby, we ensure forward progress.

To enhance the capability of resolving a conflict, we could add rollback support to the LifeTx-Stall design. However, adding rollback support will significantly increase the hardware complexity. In addition to supporting speculation, we need to keep track of the dependencies among transactions caused by the relaxed memory accesses. Each time a transaction is about to commit, it has to wait until all the dependent transactions have committed. This not only increases hardware complexity, but also could hurt performance. Since one of the goals of LifeTx-Stall design is to reduce hardware complexity, we choose not to support rollback in our LifeTx-Stall design.

B. LifeTx-CS Design

As we mentioned in Section IV-A, LifeTx-Stall is optimized for hardware complexity. We also studied how effective an ideal hardware conflict detection and resolution mechanism could be. This would require an ability to detect conflict-serializability violation similar to the design proposed in DATM [30], and an ability to rollback and re-execute a LifeTx, which could potentially contain system calls. We call this ideal design as LifeTx-CS.

V. Results

In this section, we evaluate our technique from several perspectives. We first access how much testing is required to learn LifeTxes (Section V-B). Then, we study the characteristics of resultant LifeTxes in terms of their footprints and lengths (Section V-C). After that, we discuss the bug avoidance capability of our technique using 14 documented concurrency bugs, and its trade-offs with runtime system designs (Section V-D). Finally, we evaluate the performance of two proposed runtime system designs (Section V-E).

A. Experimental Setup

We built our profiling tool using the PIN [22] binary instrumentation infrastructure. To study the runtime system, we modeled LifeTx-Stall design in Simics [23], a full system simulator. We also built a PIN based simulator to model LifeTx-CS design. All the experiments are conducted on a Quad-Core Dell T3400 workstation, with a 64-bit Redhat Enterprise Linux 5 on it.

1) Benchmarks: Two sets of benchmarks are used throughout our evaluation. The first set is called *Bug-Bench*, which used to study the bug avoidance capability of our technique. We choose 14 documented concurrency bugs from previous studies [10], [17], [21], [38] and the bug databases of a few open source applications. Among these 14 bugs, 8 of them are *Bug Kernels*, which are code snippets extracted from real buggy programs. Some program details might be omitted in *Bug Kernels*. The remaining 6 bugs are *Real Bugs*. For these real bugs, we use the original programs, and study their real executions. Table II lists the bug benchmarks we have studied. The *ID* column shows the bug identifiers in their corresponding bug databases.

The second set of benchmarks, called *Perf-Bench*, is used to evaluate LifeTx characteristics and the runtime performance. It consists of selected parallel benchmarks from Splash2 [35] ($\texttt{fft}_{perf}$, $\texttt{radix}_{perf}$, $\texttt{fmm}_{perf}$ and $\texttt{ocean}_{perf}$) and Parsec [4] ($\texttt{blackscholes}_{perf}$, $\texttt{canneal}_{perf}$), and several widely used multi-threaded applications ($\texttt{pbzip2}_{perf}$, $\texttt{pfscan}_{perf}$, $\texttt{mysql}_{perf}$, $\texttt{apache}_{perf}$). Notice that the versions of $\texttt{mysql}_{perf}$, $\texttt{apache}_{perf}$ and $\texttt{pbzip2}_{perf}$ used in *Perf-Bench* are the same as that of $\texttt{MySQL-1}_{bug}$, $\texttt{Apache}_{bug}$ and $\texttt{Pbzip2}_{bug}$ used in *Bug-Bench*. The corresponding testing methods for them are the same, saving us a little testing effort.

2) Testing Methodology: LifeTxes are learned from correct test runs. We build our profiling tool using PIN binary instrumentation tool. Our profiling tool does not require source code. We use *goose* [1], a PIN based tool, to automatically extract loop information from program binaries.

We perform testing for both *Bug-Bench* and *Perf-Bench*. For scientific programs in Splash2 and Parsec, randomly generated parameters are used (e.g. matrix size, number of threads, etc.) in each test run. For each MySQL benchmark ($\texttt{MYSQL-1}_{bug}$ to $\texttt{MYSQL-4}_{bug}$, $\texttt{mysql}_{perf}$), we use selected tests from the regression test suite that is shipped with MySQL source code. Also, we run OSDB [2] multi-user test in parallel with the regression test to create a parallel environment. Each version of MySQL is tested individually. For $\texttt{Apache}_{bug}$ and $\texttt{apache}_{perf}$, we use httperf tool to generate concurrent requests to a set of html files. For $\texttt{Pbzip2}_{bug}$ and $\texttt{pbzip2}_{perf}$, we compress a randomly chosen files using random number of threads in each test run. For $\texttt{pfscan}_{perf}$, we search a random string from a large collection of random files or directories in each test run. Finally, for all the bug kernels, we use random inputs (e.g. random loop count, random strings, etc.) and random number of threads.

3) Simulation Methodology: We use simulator to study the proposed runtime system behavior. We built two simulators, LifeTx-Stall and LifeTx-CS. We model LifeTx-Stall in Simics. We extend the *g-cache* timing model in Simics. The configurations of the baseline system is listed in Table I. The coherent caches are based on MESI protocol, and is implemented on a snoop bus. We model all the features we discussed in Section IV-A. Conflicts are detected at the granularity of word. We assume 1-cycle LifeTx commit latency and 100-cycle give up latency. The timeout threshold is set to 50K cycles.

We model another proposed design, LifeTx-CS, using PIN binary instrumentation infrastructure. The simulator models in-place memory update. The transactional undo logs are kept in the main memory as LogTM [37] does. The modeled system tests conflict serializability for LifeTxes, which is

Processor	4 cores, 2.0GHz, in-order
L1 Cache	Private, 64KB I-cache, 64KB D-cache, 4-way set associative, 32B block size, 3-cycle latency, write-back, 1KB fully associative victim cache
L2 Cache	Shared, 8MB, 8-way set associative, 128B block size, 15-cycle latency, write-back
Main Memory	2GB DRAM, 200-cycle access
Interconnect	Bus based, latency not modeled

Table I
BASELINE CONFIGURATION.

similar to MetaTM [30] does. Conflicts are detected at the granularity of word. The simulator also supports rollback and re-execution.

The inputs used in simulation are different from those used in testing. For benchmarks in Splash2 and Parsec, we use a set of input parameters that are not used during testing. For $\texttt{pbzip2}_{perf}$, we compress a new file. For $\texttt{pfscan}_{perf}$, we search a randomly generated string from a different directory. For all the MySQL benchmarks in *Bug-Bench*, we use their bug triggering inputs. For $\texttt{mysql}_{perf}$ in *Perf-Bench*, we use OSDB to generate concurrent requests to a newly created database. For $\texttt{Apache}_{bug}$ and $\texttt{apache}_{perf}$, we generate concurrent requests to a different set of html files (which could trigger the bug). Notice that the inputs used in the PIN based simulator are not identical to that used in the Simics based simulator.

B. Learning LifeTxes

To access the time required to learn LifeTxes, we perform testing for all the benchmarks using the methods described in Section V-A2. Figure 5 shows the testing results for all the *Perf-Bench* (*Bug-Bench* results are similar). Each point along the x-axis represents a unique test run [1], and the y-axis represents the cumulative total number of cutpoints learned after a particular test run. As the figure shows, our profiling algorithm reaches a stable state reasonably fast. That shows that the testing process during the normal software development should be adequate for our purpose.

C. Characteristics of LifeTxes

Once we have obtained LifeTxes from testing, the most important question to answer is how "big" each LifeTx is. Most hardware transactional memory systems have limits on the size of each transaction and even for unbounded transactional memory systems, supporting large transactions is not efficient.

D. Bug Avoidance Capability

There are several factors that contribute to the bug avoidance capability of our technique. We classify these factors into two major groups. One is coming from testing, and the other is from runtime systems.

[1]For $\texttt{mysql}_{perf}$, one test run is a 12min run of the workload we described before. For $\texttt{apache}_{perf}$, one test run means a session of concurrent requests generated by httperf.

1) Testing Impact: We infer LifeTxes from testing. For an atomicity violation bug, whether it can be avoided or not depends on whether the inferred LifeTxes enclose the *critical path* – the code execution path that need to be atomic but is not enforced by the program. In other words, to test the bug avoidance capability, we can check whether there exist cutpoints inside a critical path. If not, our runtime system will make best effort to enforce the atomicity of that critical path, avoiding the atomicity violation bug.

For each bug in *Bug-Bench*, we check each inferred cutpoint after testing is finished. We check to see whether there exists any cutpoint in the critical path. If not, the concurrency bug is under LifeTx protection. We compare LifeTx with data-race detectors and PSet [38] based systems. Table II shows the results. Among the 14 bugs we have analyzed, 12 of them are atomicity violation bugs. Out of these 12 atomicity violation bugs, LifeTx provides protection for 11 of them, including not only data-race free atomicity violations ($\texttt{BankAccount}_{bug}$, $\texttt{CircularList}_{bug}$ and $\texttt{StringBuffer}_{bug}$), but also multi-variable atomicity violation bugs ($\texttt{StringBuffer}_{bug}$ and $\texttt{MySQL-3}_{bug}$). Those bugs cannot be easily avoided by traditional data race and atomicity violation surviving techniques.

Our technique cannot avoid $\texttt{MySQL-4}_{bug}$. That is because a cutpoint is found in a function which is called in the critical path. During testing, this function is found to be not serializable under a different calling context. In order to solve this problem, we could associate context information with each cutpoint. However, that will significantly increase the runtime system complexity.

2) Runtime System Influences: Even if the critical path is under LifeTx protection, it is possible that the concurrency bug is not avoided since we assume a best-effort runtime system.

To avoid concurrency bugs, the runtime system must be able to detect them first. Usually, for atomicity violation bugs, a transactional conflict will be detected when the bug is triggered. However, under some circumstances, a runtime system might not be able to detect a bug. There are two major causes: 1) *Loss of transactional meta-data*, and 2) *Permission migration*. The problem has already been discussed in Section IV-A1. Table III shows the characteristics of the four critical LifeTxes (for the four real bugs that have LifeTx protection) that are collected from LifeTx-Stall simulation (critical LifeTxes are the LifeTxes that contain the critical sections). In the table, the second column shows how many times a particular critical LifeTx gets executed in the simulation workload, and all the other columns show the average number for each LifeTx instance. As can be observed from the table, most of the time, critical LifeTxes have moderate memory footprint and do not suffer meta-data loss or permission migration problems. The statistics shown in table III are for all executions of the critical LifeTx. Triggering a real concurrency bug in Simics is challenging. However, we successfully managed to trigger $\texttt{MYSQL-1}_{bug}$ in Simics by implementing CTrigger [25] algorithm, and the bug was successfully avoided by LifeTx-Stall design.

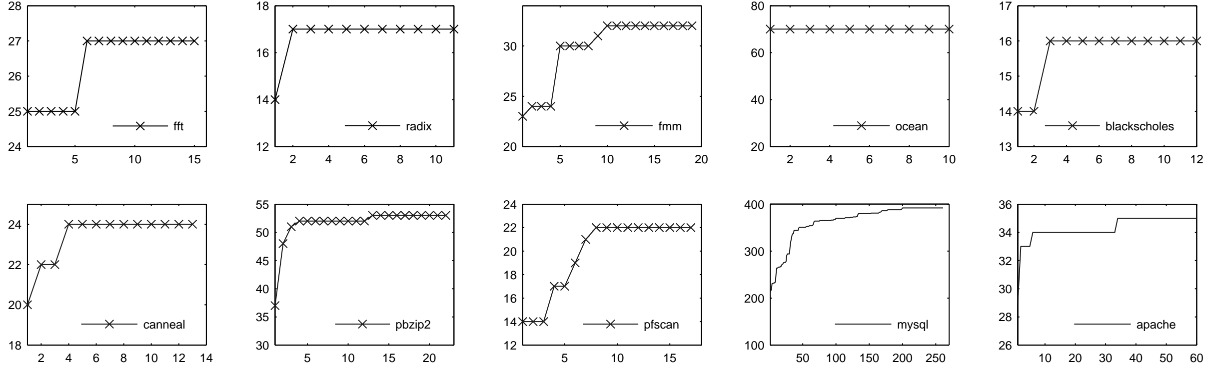

Figure 5. Number of test runs required for getting stable cutpoints.

	Bug #	Name	ID	App. Version	Type	LifeTX Protection	Data Race Detector	PSet Protection
Kernel	1	BankAccount	N/A	N/A	Single-A.V.	Yes	No	Yes
	2	CircularList	N/A	N/A	Single-A.V.	Yes	No	Yes
	3	StringBuffer	JDK1.4 [10]	N/A	Multi-A.V.	Yes	No	No
	4	LogProcSweep	N/A	N/A	Race, Single-A.V.	Yes	Yes	Yes
	5	Mozilla-1	342577	N/A	Race, Single-A.V.	Yes	Yes	Yes
	6	Mozilla-2	200119	N/A	Race, Single-A.V.	Yes	Yes	Yes
	7	Mozilla-3	52111	N/A	Race, Single-A.V.	Yes	Yes	Yes
	8	Mozilla-4	106009	N/A	Order Vio.	No	No	Yes
Real	9	Apache	25520	httpd-2.0.48	Race, Single-A.V.	Yes	Yes	Yes
	10	MySQL-1	791	mysql-4.0.12	Race, Single-A.V.	Yes	Yes	Yes
	11	MySQL-2	3596	mysql-4.0.19	Race, Single-A.V.	Yes	Yes	Yes
	12	MySQL-3	2011	mysql-4.0.16	Race, Multi-A.V.	Yes	Yes	No
	13	MySQL-4	169	mysql-3.23.56	Multi-A.V.	No	No	No
	14	Pbzip2	N/A	pbzip2-0.9.4	Race, Order Vio.	No	Yes	Yes

Table II
BUG AVOIDANCE CAPABILITY.

Name	# Instance	# Conflicts		Footprint (# Blks)	Inst. Cnt.	Meta Loss (# Blks)	Migration (# Blks)
		Resolved	Timeout				
Apache	362	0.0	0.0	126.1	4215.6	0.0	0.0
MySQL-1	2	0.0	0.0	141.5	2296.5	0.0	0.0
MySQL-2	264	1.0	0.0	158.7	17405.9	1.0	2.0
MySQL-3	3	0.0	0.0	62.0	6137.7	0.0	0.0

Table III
CHARACTERISTICS OF CRITICAL LIFETXES.

Even if a concurrency bug (the conflict) can be detected, it is possible that the runtime system cannot resolve it. For LifeTx-Stall design, since it does not have rollback support, some bugs may not be avoided. One example is two concurrent read-modify-writes race for a shared variable, stalling on either write will not resolve the bug. Even with rollback support (such as LifeTx-CS design), some bugs still may not be avoided. For example, two LifeTxes may have cyclic dependency, preventing each other from making forward progress. However, we argue that under such cases, the bug is less likely to be an atomicity violation bug since no valid execution can be found to be serializable with the code region that is intended to be atomic.

E. Performance Study

Finally, we study the performance of our runtime system. We evaluate both LifeTx-Stall and LifeTx-CS designs. We model LifeTx-Stall in Simics. For each benchmark in *PerfBench*, we start the simulation from the middle of the program execution, usually from the program point after which all worker threads have been created (e.g. after the first barrier). Table IV shows the simulation statistics. We find that most of the LifeTx conflicts can be resolved by just stalling one conflicting thread. In worst case ($\texttt{pfscan}_{perf}$), only 4 timeouts are reported. We also list the average waiting cycles for resolved conflicts. Except for the conflicts in $\texttt{pfscan}_{perf}$, most of the conflicts can be resolved in 2000 cycles. $\texttt{pfscan}_{perf}$ stalls longer than others because it spends relatively more time in OS handling I/O operations which take longer time to finish. The percentage of total waiting cycles (with respect to total simulated cycles) for each benchmark is negligible, indicating small runtime overhead. Figure 6 compare the performance between the baseline system and LifeTx-Stall system. The average overhead is less than 0.6%, which is negligible.

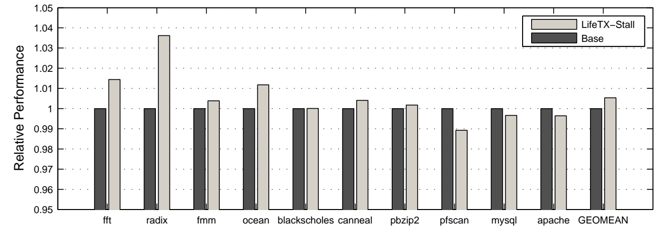

Figure 6. LifeTX-Stall performance overhead.

We also model LifeTx-CS runtime system design using a PIN based simulator. The runtime performance overhead of LifeTx-CS can be broken down into two parts. The first part is the cost in supporting a TM system (e.g. version management and conflict detection). However, post work in hardware TM [30] has shown that such cost can be

Programs	LifeTx-Stall					LifeTx-CS			
	# Conflicts		Avg. Resolved Wait Cycles	% Waiting Overhead	Inst. Cnt.	# Conflicts		% Re-exec Inst.	Inst. Cnt.
	Resolved	Timeout				Resolved	w/ Syscall		
fft	0	0	0	0.000%	543M	0	0	0.000%	130M
radix	0	0	0	0.000%	35M	0	3	0.013%	360M
fmm	178	0	631	0.125%	225M	1	2	0.000%	2270M
ocean	141	0	1116	0.563%	44M	0	0	0.000%	339M
blackscholes	0	0	0	0.000%	92M	0	0	0.000%	810M
canneal	0	0	0	0.000%	16M	32	2	0.007%	45M
pbzip2	0	0	0	0.000%	658M	0	40	0.003%	892M
pfscan	13	4	12288	1.407%	43M	1	2	0.027%	13M
mysql	178	3	470	0.002%	2.1B	23	4	0.019%	794M
apache	0	0	0	0.000%	444M	0	1	0.017%	374M

Table IV
RUNTIME STATISTICS FOR (A) LIFETX-STALL, (B) LIFETX-CS.

minimized by using hardware support. The second part of the overhead is determined by the runtime conflicts. Each runtime conflict would result in LifeTx abort and re-execute instructions. We measured the number of aborts and the number of additional instructions that need to be executed due to those aborts. Table IV shows the number of runtime conflicts (or false conflicts) and the number of aborted instructions. For example, for $\texttt{mysql}_{perf}$, we detect 27 constraint violations for a program that executed about 800 million instructions, which resulted in re-execution of 0.019% of instructions. And for the worst case ($\texttt{pfscan}_{perf}$), only 0.027% of the instructions are re-executed. Therefore, the runtime overhead due to transactional abort and re-execution is negligible. Notice that there are a few conflicts which we cannot resolve. This is because our current PIN based implementation of our simulator cannot undo the effect of certain system calls. However, this could be resolved by integrating operating system support [26].

1) Block vs. Word: To decide the conflict detection granularity for LifeTx-Stall, we implemented both versions. Table V shows the comparison results. For each design, we simulate the same number of instructions. As we can see, word level conflict detection reports far less conflicts than the block level counterpart. For example, for $\texttt{mysql}_{perf}$, block based implementation reports 41192 conflicts while the word based implementation only reports 181 conflicts. Another interesting finding is that block based implementation is more likely to cause an timeout than the word based implementation. This is reasonable since our profiling tool detects conflicts at the granularity of word. Detecting conflict on block at runtime introduces more dependencies than that seen during testing, leading to more timeouts. Therefore, we choose to detect conflicts at the granularity of word.

Programs	Block		Word	
	Conflict Resolved	Timeout	Conflict Resolved	Timeout
fft	0	0	0	0
radix	8	1	0	0
fmm	255	7	178	0
ocean	757	125	141	0
blackscholes	0	0	0	0
canneal	0	0	0	0
pbzip2	0	1	0	0
pfscan	19	13	13	4
mysql	22	41170	178	3
apache	0	0	0	0

Table V
GRANULARITY OF CONFLICT DETECTION: BLOCK VS WORD.

VI. RELATED WORK

Many research have been conducted to tackle concurrency bugs. However, most of them focus on bug detection under testing environment. There are three major types of concurrency bugs that these tools try to detect: data races, atomicity violations, and order violations. Data race detectors are widely studied [32] but they cannot detect all concurrency bugs. Atomicity violations is another major source of concurrency errors [18], [24], [36]. Many of these atomicity violation detection tools require annotations from programmers to explicitly specify which code regions need to be atomic [3], [8]. Whereas, we automatically infer atomic sections from tested executions. SVD [36], AVIO [18] and MUVI [16] use heuristics to automatically infer atomic regions. Recently, there has been efforts to detect order violations [5], [19], [39]. The invariants and constraints in these testing tools are tailored to reduce false positives as it would amount to wasted programmers effort. In contrast, our goal is to avoid concurrency bugs, where we can afford to have a higher false positive rate, provided performance cost of avoiding them is not high. This allows us to devise conservative constraints like LifeTx that tries to detect and avoid a wider range of concurrency bugs.

Bug avoidance and tolerating techniques have gained interests recently. Some techniques integrate dynamic bug detection tools with checkpoint and re-execution systems [27], [28]. In these tools, once a bug is detected, the program will be rolled back to a previous checkpoint and re-execute so as to bypass the buggy interleavings. Compared to our system, such systems have two major drawbacks. First, the bug detection tools used are optimized for reducing false positives, which could limit their capabilities on avoiding variety of concurrency bugs. Second, unlike our design, supporting checkpoint and re-execution support is heavyweight and complex.

ColorSafe [20] is a recently proposed system concurrent to our work. ColorSafe uses a data centric approach inspired from data centric synchronizations [6], [15], [33]. It identifies related data based on when they are allocated or by using programmer annotations. Then it groups them into colors

and uses hardware support to unserializable interleavings between accesses to these colors. This approach can detect certain multi-variable atomicity violations. One of the challenge in designing a data-centric system is in determining the beginning and end of the transaction. Past solution employs heuristics based on length, but it could either limit its bug avoidance capability or generates high volume of runtime conflicts. In contrast, our system does not have this problem since our system determines transaction boundaries based on observed tested interleavings. Furthermore, ColorSafe uses history buffers and color buffers to detect potential atomicity violations, which is fairly complex when compared to our design.

Interleaving constrained system [38] proposed by Yu and Narayanasamy also uses tested interleaving to constrain production run interleavings. They use PSet constraints which cannot avoid multi-variable atomicity violations. Further, their runtime system requires the capability for PSet violation detection, global checkpoint and re-execution support, which are all significantly complex when compared to our hardware proposal.

Atom-Aid [21] leverages chunk based processor design which dynamically constructs chunks and enforces a serial order between them. Atom-Aid checks potential single-variable serializability violations and dynamically constructs chunks based on that. Therefore, although Atom-Aid is able to avoid multi-variable atomicity violations, they are avoided purely by chance. In contrast, our technique constructs atomic regions by observing what unserializable interleavings have been tested, and only allow them at runtime. Therefore, our system can avoid multi-variable atomicity violation with a higher probability than Atom-Aid. Further, our runtime system is simpler and lightweight.

ISOLATOR [29] and ToleRace [31] detect and avoid one specific type of concurrency errors: asymmetric data races. An asymmetric data race occurs when one thread obeys the locking discipline correctly while some other threads do not. Both systems can efficiently detect and avoid asymmetric data races by maintaining a show memory copy for each critical section, and update it speculatively. When compared to our system, their schemes only handle one specific type of concurrency errors, while our technique tries to avoid untested interleavings, which is more general and could tolerate more types of concurrency errors, including multi-variable atomicity violations.

AVIO [18] is an architectural proposal for atomicity violation detection. Like our system, AVIO also learns constraints from correct test runs. It infers Access Interleaving (AI) invariants – whether two adjacent accesses to the same variable are unserializably interleaved with remote accesses – from correct test runs. During runtime, AVIO reports violations to these invariants by monitoring coherence messages. AVIO however cannot detect multi-variable atomicity violations, and also they did not focus on bug avoidance.

SVD [36] also automatically infers atomic regions based on a region hypothesis using control and data dependencies.

SVD's region hypothesis limits its ability to avoid complex atomicity violation bugs whose atomic regions do not obey that hypothesis (for example, $\texttt{MySQL-1}_{bug}$ in Table II). In contrast, our system does not assume any properties about atomic regions.

Transactions [13] can simplify parallel programming, but programmers could still make mistakes under this model. For efficiency reasons, a programmer might reduce the size of a transaction. Such optimization might lead to programming errors, when the size of a transaction is smaller than what it should be. Our technique does not have such issues, since LifeTxes are automatically derived from correct test runs. Therefore, our solution could be helpful even when developer program using the TM programming model, and it could serve as a complementary tool to the TM systems. Furthermore, LifeTx does not require precise enforcement of its transactional properties, which allowed us to reduce the complexity of runtime support by avoiding speculation and unbounded TM support.

VII. Conclusion

As we enter the multi-core era, providing support for developing reliable parallel programs is crucial. Most of the concurrency bugs manifest when a rare interleaving manifests in a production run. The traditional approach has been to test the program as much as possible and try to expose as many rare interleavings as possible. While testing for corner cases in single-threaded programs is absolutely necessary, it not so necessary for multi-threaded programs, especially given the fact that there are too many of those corner cases. Our approach is based on the insight that we can exploit the inherent non-determinism in parallel systems, and use it to our advantage. That is, bias the non-deterministic thread schedule to pick a tested interleaving as much as possible.

Sun has already incorporated hardware support for transactions. The proposed design could be simpler than a conventional HTM support. We proposed an algorithm for automatically deriving Lifeguard transactions from tested executions. We showed that the performance overhead of our stall based mechanism is negligible, and that we can avoid 11 out of 12 atomicity violations in programs like MySQL and Apache.

References

[1] The goose tool. http://systems.cs.colorado.edu/ moseleyt/goose/.

[2] The open source database benchmark. http://osdb.sourceforge.net/.

[3] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 167–178, New York, NY, USA, 2010. ACM.

[6] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 133–144, Washington, DC, USA, 2007. IEEE Computer Society.

[7] C. Flanagan and S. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[8] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, 2004.

[9] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, 2008.

[10] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[11] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Pub, 1993.

[12] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[13] M. Herlihy, V. Luchangco, M. Moir, and W.N.Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second ACM SIGACT-SIGOPS Symposium on Princicples of Distributed Computing*, 2003.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM, 1993.

[15] M. Isard and A. Birrell. Automatic mutual exclusion. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[16] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'08)*, 2008.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, 2006.

[19] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 553–563, New York, NY, USA, 2009. ACM.

[20] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA '10: Proceedings of the 37nd Annual International Symposium on Computer Architecture*, Saint-Malo, France, 2010.

[21] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[22] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[23] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[24] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, New York, NY, USA, 2008. ACM.

[25] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36, New York, NY, USA, 2009. ACM.

[26] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176, New York, NY, USA, 2009.

[27] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[28] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM.

[29] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[30] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[31] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–184, New York, NY, USA, 2009. ACM.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[33] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345, New York, NY, USA, 2006. ACM.

[34] H. Volos, N. Goyal, and M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT'08: 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[36] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[37] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007.

[38] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 325–336, New York, NY, USA, 2009.

[39] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 179–192, New York, NY, USA, 2010. ACM.