Lazy Diagnosis of In-Production Concurrency Bugs

Baris Kasikci
University of Michigan and Microsoft Research
barisk@umich.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Xinyang Ge
Microsoft Research
xing@microsoft.com

Ben Niu
Microsoft Research
beniu@microsoft.com

ABSTRACT

Diagnosing concurrency bugs—the process of understanding the *root causes* of concurrency failures—is hard. Developers depend on reproducing concurrency bugs to diagnose them. Traditionally, systems that attempt to reproduce concurrency bugs record fine-grained thread schedules of events (e.g., shared memory accesses) that lead to failures. Recording schedules incurs high runtime performance overhead and scales poorly, making existing techniques unsuitable in production.

In this paper, we formulate the *coarse interleaving hypothesis*, which states that the events leading to many concurrency bugs are coarsely interleaved. Therefore, a fine-grained and expensive recording is unnecessary for diagnosing such concurrency bugs. We test the coarse interleaving hypothesis by studying 54 bugs in 13 systems and find that it holds in all cases. In particular, the time elapsed between events leading to concurrency bugs is on average 5 orders of magnitude greater than what is used today in fine-grained recording.

Using the coarse interleaving hypothesis, we develop Lazy Diagnosis, a hybrid dynamic-static interprocedural pointer and type analysis to diagnose the root causes of concurrency bugs. Our Lazy Diagnosis prototype, Snorlax, relies on commodity hardware to track thread interleavings at a coarse granularity. Snorlax does not require any source code changes and can diagnose complex concurrency bugs in real large-scale systems (MySQL, httpd, memcached, etc.) with full accuracy and an average runtime performance overhead of below 1%. Broadly, we believe that our findings can be used to build more efficient in-production bug detection and record/replay techniques.

SOSP '17, Shanghai, China

DOI: 10.1145/3132747.3132767

CCS CONCEPTS

•Software and its engineering → Multithreading; Concurrency control; Software testing and debugging;

ACM Reference format:
Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 17 pages.
DOI: 10.1145/3132747.3132767

1 INTRODUCTION

Bug diagnosis, the activity of identifying the root cause of a failure, is expensive. According to a recent report [80], testing and debugging with the purpose of bug diagnosis amounts to 35% of all software development costs, a figure expected to rise up to 41-50% by the end of 2018. Diagnosing and fixing bugs is also time-consuming, taking up to 50% of developers' time [61].

Concurrency bugs, which happen due to synchronization problems in multithreaded software, are notorious for being harder and more time-consuming to fix than other types of bugs [56]. Concurrency bugs can have catastrophic consequences [50, 88], and can compromise system security [24, 94]. To diagnose concurrency bugs, developers need to determine the exact interleavings of memory accesses and/or synchronization operations leading to failures. According to practitioners in industry [32] and in the open-source community [96], diagnosing and fixing concurrency bugs can take weeks, or even up to a month.

To make matters worse, certain concurrency bugs only occur in production and cannot be reproduced in house due to their non-deterministic nature. This substantially complicates debugging [42], because traditionally, developers rely on reproducing bugs to fix them. In particular, a study by Google revealed that developers lacked the means to diagnose and fix hard-to-reproduce bugs [81].

A notable method to diagnose concurrency bugs that are difficult to reproduce is record/replay systems [62, 64, 71, 77]. Record/replay systems record the execution of a

program (inputs, system call return values, thread schedules, etc.), which enable developers to reproduce failing executions and diagnose bugs. Despite improvements in the past decade, multiprocessor record/replay of arbitrary multithreaded programs incurs large runtime performance overheads (200% in the worst case for the state of the art [92]), making record/replay unsuitable for in-production usage. Furthermore, expensive record/replay systems may perturb executions and hide concurrency bugs [42]. For instance, rr [64], Mozilla's record/replay system, serializes concurrent executions, which may mask bugs that may only occur during parallel executions.

Another promising approach to tackle hard-to-reproduce in-production bugs is failure diagnosis systems [39, 42, 53]. These systems identify failure root causes using instrumentation and/or non-commodity hardware support to record events such as memory accesses, code execution paths, and thread schedules. Although promising, these systems assume that in-production code can be instrumented for sampling purposes [51, 52], or use unscalable mechanisms to track executions, or rely on custom hardware support for bug diagnosis. None of these assumptions hold for real-world software. *Consequently, there is no deployed in-production system for concurrency bug diagnosis.*

State-of-the-practice in-production bug diagnosis systems (e.g., the ones used in industry) have stringent efficiency requirements. Therefore, they rely on post-mortem analysis of crash dumps [2, 13, 31, 33]. These systems diagnose bugs by mainly analyzing call stacks in a crash dump using a mixture of custom heuristics and function white-listing. The most advanced in-production bug diagnosis system that we are aware of is RETracer [23] from Microsoft, which performs a backward data-flow analysis from a corrupt pointer to identify the provenance of the corruption.

In this paper, we introduce the *coarse interleaving hypothesis*, which states that the events leading to many concurrency bugs in real systems are coarsely interleaved. We test the coarse interleaving hypothesis using 54 concurrency bugs (order violations, atomicity violations, and deadlocks) in 13 real systems. We find that the time elapsed between events leading to concurrency bugs is at least 91 microseconds, which is 5 orders of magnitude greater than what fine-grained recording provides. Consequently, a more efficient coarse-grained time tracking mechanism is sufficient to track thread interleavings in practice.

Using the coarse interleaving hypothesis, we develop *Lazy Diagnosis*, a hybrid dynamic-static analysis technique for accurate and efficient concurrency bug diagnosis. Lazy Diagnosis relies on control flow tracing with timing information in modern hardware (e.g., Intel Processor Trace [35] or ARM Embedded Trace Macrocell [14]). The key idea behind Lazy Diagnosis is to *lazily* bind dynamic control and timing information to an otherwise expensive static interprocedural pointer and type analysis. The control flow trace allows static analysis to only focus on executed code, and the coarse timing information enables the analysis to be partially flow sensitive, resulting in accurate and efficient root cause diagnosis.

As we will explain in §7, the coarse interleaving hypothesis may not hold for concurrency bugs in programs with a high degree of parallelism and fine-grained concurrency. If a concurrency bug does not meet the coarse interleaving hypothesis, Lazy Diagnosis will not produce misleading results. Lazy Diagnosis will point out the events (e.g., memory accesses, lock operations, etc.) that are likely involved in the concurrency bug without providing the ordering information between these events, which we believe is still useful.

Overall, we make the following contributions:

- *Coarse interleaving hypothesis*, which claims that a coarse-grained timing information is sufficient to infer thread interleavings leading to many concurrency bugs. We validate the hypothesis for 54 bugs in real systems, and discuss implications of this hypothesis to testing and debugging multithreaded programs.
- *Lazy Diagnosis, a novel hybrid dynamic-static program analysis technique* that leverages the coarse interleaving hypothesis to accurately and efficiently diagnose concurrency bugs. Lazy Diagnosis can diagnose bugs in unmodified programs running on commodity hardware. Lazy Diagnosis does not rely on sampling, which leads to low root cause diagnosis latency (i.e., it can quickly diagnose bugs).

We implemented Lazy Diagnosis in our prototype system Snorlax [1]. We tested Snorlax using real systems such as MySQL, Memcached, Apache httpd, SQLite, Transmission. Snorlax diagnosed concurrency bugs with 100% accuracy with a runtime performance overhead of 0.97% on average. Snorlax can diagnose concurrency bugs after a single failure because it does not rely on sampling. Although Snorlax is geared towards the Intel platform, Lazy Diagnosis can be implemented on other platforms with control flow tracing capability (e.g., ARM). Therefore, we believe that our techniques are broadly applicable, and Snorlax is suitable for real-world adoption.

In the rest of this paper, we first discuss the challenges of developing concurrency bug diagnosis techniques (§2), followed by a discussion of the coarse interleaving hypothesis (§3). We then describe the design of Lazy Diagnosis (§4) followed by the implementation (§5) and evaluation (§6) of Snorlax. After discussing limitations (§7) and the related work (§8), we finally conclude the paper (§9).

[1] Snorlax is a lazy, but powerful Pokémon

2 CHALLENGES OF CONCURRENCY BUG DIAGNOSIS

We now explain the key challenges of diagnosing concurrency bugs which apply to testing and debugging concurrency bugs as well. We also explain how challenges of concurrency bug diagnosis are interrelated.

2.1 Overhead Challenge

A fundamental challenge in building effective concurrency bug diagnosis techniques is the runtime performance overhead incurred on the programs monitored for diagnosis purposes. Low runtime performance overhead is especially important for diagnosing in-production bugs.

To reduce performance overhead, existing techniques employ a variety of methods. The first technique is to use custom hardware support [16, 76, 77], which hurts applicability. The second major technique is to use sampling [15, 17, 42, 44, 51, 52]. Alas, sampling techniques can significantly reduce the probability of diagnosis. For instance, according to our evaluation (§6.3), the diagnosis probability can be reduced at least by a factor of 3.7× and up to 2523×. Furthermore, certain sampling techniques modify the source code or instrument in-production code on-the-fly, which may not be practical or even possible. Moreover, modification or instrumentation of deployed programs could perturb program behavior and mask bugs [42].

Certain techniques perform heavyweight in-house analyses [43] to avoid any runtime performance overhead. These techniques may take a long time to analyze executions (e.g., up to 5000× longer than real executions).

Lazy Diagnosis solves the overhead challenge with a hybrid dynamic-static program analysis that combines non-intrusive and low-overhead hardware control flow tracing, coarse-grained timing information, and powerful interprocedural static program analysis (§4). Control flow traces allow static analysis to reduce its scope to executed code (§4.2), and timing information enables partial flow sensitivity (§4.4), resulting in accurate diagnosis without sacrificing performance.

2.2 Accuracy Challenge

Bug diagnosis techniques need to be accurate (i.e., provide correct results) to be adopted and used by developers in the real world. For instance, if the results of a data race detector contain too many false positives, developers may end up losing a lot of time weeding out the false positives, or worse, they will simply not use the detector.

It is difficult to correctly identify the program state that leads to a failure in a complex large-scale software. An accurate concurrency bug diagnosis system needs to correctly track the statements leading to failures and their execution order. The accuracy challenge is also related to the overhead challenge. Tracking execution information is necessary for accurate diagnosis, but this can be expensive, especially for in-production code.

Existing techniques rely on either non-commodity hardware support [76, 77] or heavyweight analyses that are not suitable for in-production usage [43], whereas Lazy Diagnosis achieves high accuracy thanks to its hybrid dynamic-static interprocedural program analysis (§4).

2.3 Bug Diagnosis Latency Challenge

Existing bug diagnosis techniques are prone to long diagnosis latencies or low diagnosis probabilities due to sampling. There are two predominant sampling strategies. The first strategy is sampling in time [17, 40, 51, 52], i.e., turning monitoring on and off at certain time intervals, and the second is sampling in space [42, 44] which turns on monitoring for certain portions of a program. Prior work has shown that sampling in time can dramatically increase the latency of bug diagnosis (up to 100× according to [16]), and we show in our evaluation that sampling in space can cause an even more drastic increase (up to 2523× §6.3).

Reducing the bug diagnosis latency of hard-to-reproduce bugs requires monitoring programs continuously. To our knowledge, there is no in-production system that continuously monitors entire program executions for bug diagnosis purposes. The only related work to do so uses custom hardware [39] that is not readily available.

Lazy Diagnosis relies on efficient control flow tracing in modern commodity hardware (Intel PT [35] or ARM ETM [14]) to continuously track executions, thereby reducing bug diagnosis latency.

3 COARSE INTERLEAVING HYPOTHESIS

In this section, we evaluate the coarse interleaving hypothesis, which claims that coarse-grained timing information is sufficient to determine the thread interleaving of events (shared memory accesses and synchronization operations) leading to concurrency bugs. In the rest of the text, we refer to these events as *target events* or *target instructions*.

We defined diagnosis of a bug as the identification of the failure's root cause (§1), where the root cause of a failure is intuitively *the real reason* behind the failure. More precisely, we define the execution order of target events across threads as the *root cause* of concurrency bugs. Although we do not attempt a formal proof, we borrow this notion of root cause from the notion of causation defined by Halpern *et al.* [34]. In particular, the execution order of events such as memory accesses are the only events pertinent to the occurrence of concurrency bugs we have encountered (deadlocks, order violations, and atomicity violations), and any other event is immaterial to reproducing these bugs.

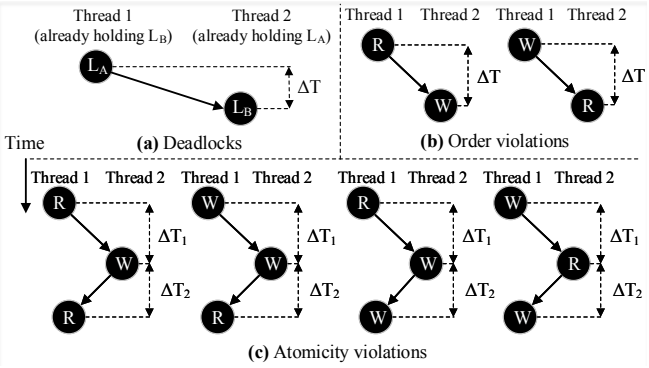

Figure 1: Patterns of concurrency bugs and target events

3.1 Concurrency Bug Patterns

Target event patterns vary based on the type of the concurrency bug. In this paper, we consider deadlocks, order violations, and atomicity violations. Order violations and atomicity violations are in many cases caused by one or more data races, which are unsynchronized accesses to shared variables from multiple threads.

We show the bug types and associated target event patterns in Figure 1. L_A denotes an attempt to call a lock acquisition function to acquire the lock A. R and W denote read and write instructions to the same memory location respectively. The arrow sign between events denotes an ordering in time. It is not a formal happens-before relationship [48] established by synchronization operations except in the case of the deadlock pattern.

Deadlocks happen when a group of threads are each waiting for one another to release a lock to progress with their execution. A deadlock pattern between two threads is in Figure 1.(a), where Thread 1 which is already holding the lock L_B attempts to acquire L_A. L_A is held by Thread 2, which in turn is attempting to acquire L_B. We denote the time elapsed between the two lock acquisition attempts as ΔT. Lazy Diagnosis is not limited to deadlocks with two threads, but we only show an example with two threads for brevity.

Order violations happen when two threads access the same memory location, at least one of the accesses is a write, and the sequence of accesses violates the program's correctness. An order violation pattern is in Figure 1.(b), where we denote the time elapsed between the two accesses in an order violation as ΔT.

Atomicity violations happen when developers do not properly identify sections of code that need to execute atomically and fail to enclose them in critical sections [57]. We show the patterns of single-variable atomicity violations in Figure 1.(c), namely RWR, WWR, RWW, WRW, since they are the most common ones [56]. Atomicity violations may involve order violations, however their key characteristic is the violated atomicity of a code section. We focus on single variable atomicity violations, because as we discuss in §7, it is challenging for Lazy Diagnosis to diagnose multi-variable atomicity violations without tracing the data flow of a program. We refer to the time elapsed between the first and the second access as ΔT_1 and the time elapsed between the second and the third access as ΔT_2.

It is possible to determine the order of target events if the time elapsed between the events can be measured. A developer can then fix the associated concurrency bugs e.g., by ensuring that threads do not acquire locks in the order that leads to a deadlock or by using proper synchronization to eliminate the order or the atomicity violation.

3.2 Evaluating the Coarse Interleaving Hypothesis

We now briefly discuss the systems we used to test the coarse interleaving hypothesis. Our benchmarks are widely-used real-world systems. MySQL [66] is a popular relational database system used by companies such as Google, Facebook, and Twitter; Apache httpd [7] is the most widely used web server in the world, powering around 50% of all the websites [69]; memcached is a distributed object cache used by companies such as Amazon, Youtube, and Facebook [28]; SQLite [86] is an embedded database used in Chrome, Firefox, and iOS; Transmission is a cross-platform BitTorrent client [78]; Pbzip2 is a parallel compression and decompression tool [30]; aget is a parallel wget utility [26]; the Java Development Kit (JDK) [72] is the implementation of the Java Platform; Apache Derby is a relational database implemented entirely in Java [9]; Apache Groovy is a dynamic programming language [10] used by services such as Eucalyptus [27] and Jenkins [38]; DBCP is Apache's connection pooling infrastructure for relational database connections in Java [8]; Apache Log4j is a Java-based logging utility [11]; Apache Lucene is an indexing and search library [12].

To measure the time elapsed between target events, we first identify the target instructions leading to each concurrency bug. We then instrument C/C++ programs using `clock_gettime()` and Java programs using `System.nanoTime()`, which also relies on `clock_gettime()` [29]. The instrumentation is solely for the purposes of evaluating the coarse interleaving hypothesis. Lazy Diagnosis and Snorlax do not rely on any instrumentation or source code modification. The instrumentation calls to time measurement functions are injected as immediate predecessors [3] of target instructions.

Linux uses the nanosecond-granularity time stamp counter (TSC) in Intel processors to implement `clock_gettime()` [55]. The reliability of our results depends on the accuracy of the TSC, and in particular, whether it is properly synchronized

	SQLite-1672	DBCP-65	DBCP-270	Derby-4129	Derby-5560	Derby-764	Derby-5447	Groovy-4736	JDK-6492872	JDK-6582568	JDK-6588239	JDK-6648001	JDK-6927486	JDK-6934356	JDK-7122142	JDK-8010939	Log4j-41214	Log4j-38137	Log4j-50463	Lucene-2783	Lucene-1544	Pool-146	Pool-149	Pool-162
ΔT	334	239	207	101	3505	265	782	567	892	245	216	198	1451	414	392	229	173	1780	990	234	249	780	621	614
$\sigma_{\Delta T}$	171	28	12	14	40	45	232	134	270	32	38	56	415	67	106	37	11	260	118	28	32	74	90	171

Table 1: Average time elapsed for deadlocks (ΔT in Figure 1.a) and standard deviation ($\sigma_{\Delta T}$) in microseconds

	DBCP-369	Groovy-3495	GROOVY-6068	GROOVY-4292	JDK-4243978	JDK-4779253	JDK-7100996	JDK-7045594	JDK-7132378	JDK-8023541	Log4j-44032	Log4j-54325	Pool-120	Pbzip2-N/A	Transmission-N/A
ΔT	451	296	780	154	985	619	442	399	283	516	341	424	303	673	569
$\sigma_{\Delta T}$	38	71	227	45	244	110	69	45	51	104	43	117	78	101	45

Table 2: Average time elapsed for order violations (ΔT in Figure 1.b) and standard deviation ($\sigma_{\Delta T}$) in microseconds

	Derby-5561	Groovy-4292	Groovy-5198	GROOVY-6456	JDK-4813150	JDK-6436220	JDK-7132889	Aget-N/A	apache-21285	apache-21287	apache-25520	Memcached-127	mysql-12228	mysql-3596	mysql-644
ΔT_1	451	174	2811	275	991	860	394	624	301	291	541	370	730	644	519
$\sigma_{\Delta T1}$	125	39	293	87	126	96	30	77	31	27	30	28	43	74	62
ΔT_2	670	367	451	256	515	506	710	619	412	195	531	213	1456	302	400
$\sigma_{\Delta T2}$	47	53	75	34	67	68	57	98	44	21	68	57	91	54	62

Table 3: Average times elapsed for atomicity violations (ΔT_1, ΔT_2 in Figure 1.c) and standard deviations ($\sigma_{\Delta T1}$, $\sigma_{\Delta T2}$) in microseconds

across multiple cores in our test machine which has an Intel Skylake i7-6500U CPU. Fortunately, Intel CPUs since Nehalem have invariant TSC [36], meaning that the TSC is synchronized across the CPUs [95]. Furthermore, the TSC is not affected by CPU frequency changes.

We reproduced all the 54 concurrency bugs in 13 systems that we could reproduce, while measuring time elapsed between target events. We chose these systems and bugs because they were previously used for evaluating concurrency bug diagnosis and debugging tools [16, 42, 97], and there are publicly available frameworks to reproduce these bugs reliably [54, 83]. To correctly evaluate the coarse interleaving hypothesis, we did not instrument these programs (e.g., by inserting delays) to increase the bug reproduction probability. Consequently, in some cases, we had to run programs a few thousand times (less than 5000 in any case) to reproduce the bugs.

We show the results of our study in Tables 1–3. The elapsed times represent averages over 10 runs, and we also provide standard deviations. The first row in each table shows the system name as well as the bug tracker id (N/A if the bug ID is not available). As a summary, the average time elapsed between target events is between 154 and 3505 microseconds. The shortest time elapsed is 91 microseconds. We discard the overhead of calling time measurement functions, which we measured to be always less than 1 microsecond in total for a given execution.

3.3 Implications of The Coarse Interleaving Hypothesis

Our evaluation results of the coarse interleaving hypothesis are encouraging. In particular, our results show that it is possible to diagnose all the concurrency bugs we studied using an efficient time tracking mechanism that is roughly 5 orders of magnitude coarser than what a fine-grained record/replay system would provide.

We compute the magnitude of the granularity difference as follows: Our study in the previous section revealed that the

shortest time elapsed between target events was 91 microseconds. On the other hand, multi-processor record/replay should be able to provide nanosecond granularity. Consider an L1 cache hit, which takes around one nanosecond (4 cycles in Skylake [84]). A record/replay system should be able to track the *exact* order in which multiple L1 cache hits from different cores execute, which requires nanosecond recording granularity. We then compute the 5 orders of magnitude ratio as $\frac{91us}{\sim 1ns} = 91000 \sim 10^5$.

As we show for our Lazy Diagnosis prototype implementation, Snorlax, modern hardware is able to provide such granularity with an average performance overhead of 1%.

We posit that the coarse interleaving hypothesis holds for real large-scale systems because of the inherent complexity of such systems (e.g., many operations, context switches, network communication, etc.). We acknowledge that the coarse interleaving hypothesis does not hold for operations in highly concurrent software such as a concurrent linked list, where a formal verification of correctness may be possible [18]. However, based on our study, we conclude that the coarse interleaving hypothesis is useful, and as we show in the next section, it can be used to efficiently diagnose concurrency bugs in practice.

We also believe that the coarse interleaving hypothesis can be useful for designers of bug detection tools and techniques such as record/replay systems and symbolic analysis tools. Recent work has shown that it is possible to build an efficient record/replay system if a multithreaded program has no data races [60]. Although this is true, record/replay systems are especially valuable for debugging hard-to-reproduce failures due to concurrency bugs such as data races. In many cases, the coarse interleaving hypothesis can be used to efficiently record the order of racing accesses, thereby enabling the design of efficient record/replay engines that can work in the presence of data races. Similarly, symbolic analysis engines that attempt to synthesize executions with data races [43, 99, 100] can leverage the coarse interleaving hypothesis to efficiently record the order of racing accesses rather than attempting to synthesize thread schedules with data races.

4 DESIGN OF LAZY DIAGNOSIS

As discussed in §2, the key challenge of building accurate concurrency bug diagnosis tools is achieving low overhead. The key reason why existing diagnosis tools incur high overhead is because the mechanisms they use to accurately track the interleaving of shared memory accesses and synchronization operations across multiple threads are expensive [92]. The coarse interleaving hypothesis suggests that it is possible to use a relatively coarse-grained and inexpensive time tracking mechanism to determine the execution order of target events as long as the granularity is sufficient.

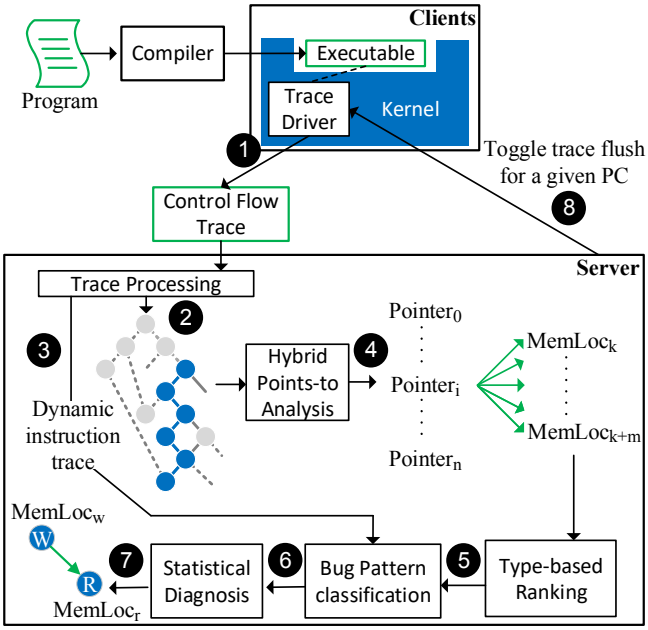

Figure 2: Design of Lazy Diagnosis

In this section, we present the design of Lazy Diagnosis, which builds on the coarse interleaving hypothesis. Lazy Diagnosis relies on efficient control flow tracing with timing information that is present in modern processors (Intel PT or ARM ETM) to *continuously* monitor the execution of programs. For instance, Intel PT generates a per-thread trace that contains both control flow events (e.g., taken branches) and timing events (e.g., TSC) that are synchronized across all CPUs. Lazy Diagnosis uses control flow traces with timing information to perform an interprocedural dynamic-static analysis for concurrency bug diagnosis. The analysis is interprocedural, because concurrency bugs can span multiple functions.

Figure 2 shows the high-level design of Lazy Diagnosis. Lazy Diagnosis operates in a client-server model, where the program is run on client machines in production, while continuously collecting control flow traces, and the server performs the core of Lazy Diagnosis' analysis. Note that a client in our model can be either a desktop or a server machine.

The control flow trace with timing information is generated upon a failure such as a crash or a deadlock, or on demand (step ①). Lazy Diagnosis performs steps ② to ⑦ after the server receives the first control flow trace upon a failure to identify the most likely root cause of the failure. To increase the statistical significance and hence the accuracy of bug diagnosis in step ⑦, Lazy Diagnosis gathers further control flow traces from successful executions, generated at the location where the failure occurred previously (step ⑧).

We now explain how the key components of Lazy Diagnosis work to solve the efficiency, accuracy, and root cause diagnosis latency challenges we outlined in §2. In particular, we discuss trace processing (§4.1), hybrid points-to analysis (§4.2), type-based ranking (§4.3), bug pattern computation (§4.4), and statistical diagnosis (§4.5).

$$\frac{p = \&l}{MemLoc_l \in p}\ (1) \qquad \frac{p = q}{p \supseteq q}\ (2) \qquad \frac{*p = q}{*p \supseteq q}\ (3) \qquad \frac{p = *q}{p \supseteq *q}\ (4)$$

Figure 3: Constraint inference rules for inclusion-based points-to analysis [4]

4.1 Trace Processing

First, trace processing uses control flow traces from clients to identify the executed instructions (step ②, Figure 2). In this context, when we say instructions, we refer to instructions used by Lazy Diagnosis, where the actual format is implementation-dependent (e.g., LLVM IR). This step does not take into account the timing information in the traces or the dynamic sequence of instructions (e.g., an instruction executed multiple times counts as having executed once).

We illustrate the result of step ② on the hypothetical control flow graph of the program, where blue nodes correspond to executed code as per the trace, and other nodes are grayed-out. The nodes on the graph represent basic blocks, which are contiguous instruction sequences without a branch. The edges represent branches, whose executions are tracked by control flow tracing. Hybrid points-to analysis uses the results of this to reduce the scope of the analysis (§4.2).

Second, trace processing uses the control flow trace with the timing information to generate a *dynamic instruction trace* of executed instructions that are *partially-ordered* in time, i.e., a subset of the executed instructions are ordered with respect to each other (step ③, Figure 2). The instructions in this trace are partially-ordered, because in practice, the granularity of the timing information in an efficient control flow tracing mechanism (e.g., Intel PT) is too coarse to infer a total order of all instructions. Nevertheless, as per the coarse interleaving hypothesis, we show that a partial order is sufficient to diagnose real concurrency bugs (§6.1). The results of this step are used by bug pattern computation (§4.4).

Trace processing happens whenever the server receives a control flow trace from a failing execution, or from a successful execution at the request of the server. The server requests traces to be generated for successful executions (step ⑧) to increase the statistical accuracy of root cause diagnosis (step ⑦). However, we note that Lazy Diagnosis does not employ statistical *sampling* like prior root cause diagnosis techniques [39, 42, 51–53].

Lazy Diagnosis instructs clients to generate traces from successful executions at the same program counter where a failure previously occurred. It may not be always possible to generate a trace at a previous failure location (e.g., if the failure is in error handling code), in which case the server will instruct control flow traces to be generated at predecessor basic block(s) of the block where the failure occurred previously. Lazy Diagnosis clients iterate over predecessor blocks until they reach a block where a trace can be generated.

4.2 Hybrid Points-to Analysis

Lazy Diagnosis uses trace processing results from step ③ to perform the hybrid interprocedural points-to analysis (step ④, Figure 2) that establishes a mapping between pointers and memory locations (e.g., $\texttt{Pointer}_i$ can point to locations $\texttt{MemLoc}_k$ to $\texttt{MemLoc}_{k+m}$). The hybrid points-to analysis is *lazy* in that it computes the points-to set only whenever the server receives a new control flow trace from a client, as opposed to offline, in an eager manner.

The key insight behind the hybrid points-to analysis is that we can relatively quickly perform an otherwise slow and unscalable interprocedural program analysis using the control flow information gathered from clients.

Hybrid points-to analysis is an inclusion-based points-to analysis [5]. In short, such an analysis iterates over all the instructions in a program while generating constraints, which are then fed into a constraint solver to determine points-to sets. In Figure 3, we provide four inference rules assuming C/C++ as the source language (the rules are similar for other languages), where $\texttt{MemLoc}_l$ is a memory location representing the address of object `l`. Pointers are represented by `p` and `q`. Rule (1) states that the assignment `p = &l` creates a constraint stating that the memory location of `l` is in the set of locations pointed to by `p`. Rule (2) states that the assignment `p = q` generates a constraint stating that the set of locations pointed to by `p` are a superset of the locations pointed to by `q` (i.e., the points-to set of `p` includes the points-to set of `q`, hence the name inclusion-based).

Inclusion-based points-to analysis is more accurate [25] than the other major class of interprocedural analysis, namely unification-based points-to analysis [87]. Although inclusion-based points-to analysis is more expensive than unification-based points-to analysis, hybrid points-to analysis employs *scope restriction* to improve the efficiency of inclusion-based analysis, while leveraging its higher accuracy.

Scope restriction Lazy Diagnosis restricts the scope of interprocedural analysis to the instructions that executed. This reduces the amount of code that Lazy Diagnosis needs to analyze by 9× on average (§6.1). Scope restriction enables Lazy Diagnosis to operate on code that actually executed, but

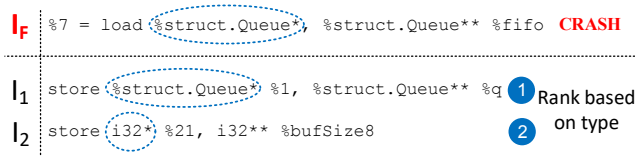

Figure 4: Type-based ranking example. The failure in this example is a crash. The ranking favors the `Queue*` type over the `i32*` type, because the instruction where the failure occurs operates on the type `Queue*`.

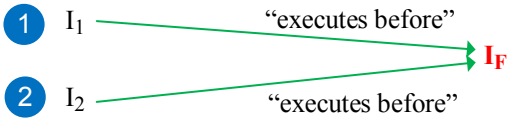

Figure 5: Partial flow sensitivity across instructions involved in a concurrency bug (I_1, I_2 and I_f from Figure 4). The "executes before" relationship is based on the timing information in the control flow trace, and it is not a "happens before" [48] relationship.

it comes at the cost of potentially missing points-to relations because, in practice, control flow traces are limited in size. For instance, we capture the last 6764 branches on average with a 64KB ring buffer in our evaluation. However, we will show that scope restriction does not impact root cause diagnosis accuracy in our evaluation.

Finally, hybrid points-to analysis is flow insensitive, i.e., it discards the execution order of program instructions. Hybrid points-to analysis leverages the conservativeness of flow insensitivity when analyzing a multi-threaded program. The execution order of instructions in a multithreaded program cannot be inferred based on the order of instructions in the code. In a multithreaded program, instructions from different threads can be arbitrarily interleaved to affect the points-to information, and flow insensitivity models this behavior. However, Lazy Diagnosis introduces flow sensitivity among target instructions during bug pattern computation (§4.4). This approach allows us to use the timing information in a conservative way.

4.3 Type-Based Ranking

Type-based ranking takes as input the points-to set of the operand of a failing instruction. The operand depends on the type of the failure: for a deadlock, the operand is a pointer to a lock object, and for a crash, the operand is an invalid pointer (e.g., due to memory corruption). Type-based ranking then ranks the instructions accessing the memory locations in the points-to set of the operand, based on the likelihood with which these instructions could be involved in a concurrency bug (Figure 2, step ⑤). We discuss the specifics of retrieving the operand from the instruction where the failure occurred in the implementation section (§5).

Type-based ranking highly ranks the instructions that operate on types that exactly match the type of the operand involved in the failure. Type-based ranking operates on static instances of instructions. In the bug pattern computation section (§4.4), we explain how Lazy Diagnosis takes into account the dynamic instances of the instructions.

Figure 4 shows an example of type-based ranking using the LLVM IR (intermediate representation) [49]. We assume that the failure in this example is a crash. I_f is the instruction where the failure occurred, and based on the hybrid points-to analysis, I_f, I_1, and I_2's operands (circled on the figure) might point to the same memory location. Type-based ranking ranks I_1, the store instruction operating on type `Queue*`, with rank 1, because I_f also operates on the same type. However type-based ranking ranks I_2, the store instruction operating on the 32-bit integer pointer type `i32*`, with rank 2.

Lazy Diagnosis does not discard any instruction based on its rank, because instructions operating on different types can be involved in the same concurrency bug. A type mismatch occurs due to type casts, where for instance, an `i32*` could be referring to a `Queue*`. Type-based ranking merely *prioritizes* the remaining stages of the analysis to improve bug diagnosis latency. In our evaluation (§6), type-based ranking decreases root cause diagnosis latency by 4.6×.

4.4 Bug Pattern Computation

Bug pattern computation takes as input the output of type-based ranking and the partially-ordered dynamic instruction trace generated by trace processing, and identifies patterns that may have caused concurrency bugs (step ⑥, Figure 2). Lazy Diagnosis is able to identify the patterns of deadlocks, order violations and atomicity violations from Figure 1.

As discussed in §3, concurrency bug diagnosis requires knowing the execution order of target instructions involved in the bug. Lazy Diagnosis determines the execution order of target instructions using *partial flow sensitivity*.

***Partial flow sensitivity*:** It uses the partially-ordered dynamic instruction trace (step ③, Figure 2) to add the ordering information between the *dynamic instances* of instructions that were previously ranked based on their types. Figure 5 shows how the instructions I_1, I_2, and I_F from Figure 4 are augmented with "executes before" relations. Partial flow sensitivity establishes executes-before relations across dynamic instances of instructions, as opposed to type-based ranking that operates on static instances of instructions. However, for brevity purposes, in Figure 5, we assume that each static instance of I_1, I_2 and I_f from Figure 4 only has one dynamic instance in the trace. Therefore, we refer to the dynamic and static instances of the instructions with the same name.

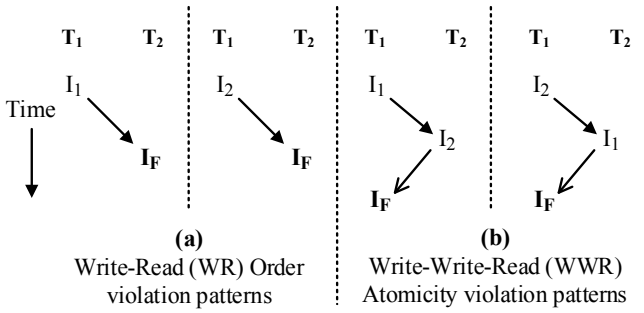

Figure 6: Patterns of potential concurrency bugs for the instructions in the example from Figure 4. T_i stands for thread i. I_1 and I_2 are writes (`store`); I_f is a read (`load`).

Next, bug pattern computation uses the partially-ordered instructions to generate potential deadlock, order violation, and atomicity violation patterns.

***Deadlocks*:** Major operating systems like Windows [31] and Ubuntu Linux [91] and execution environments such as the JVM can identify [73] that a failure occurred due to a deadlock. If the failure was due to a deadlock, Lazy Diagnosis generates potential deadlock patterns using the type-ranked and partially flow-sensitive instructions. Lazy Diagnosis determines which of the generated patterns is the root cause of the deadlock with great certainty in the final stage, namely, statistical diagnosis (§4.5).

***Order violations and atomicity violations*:** It is not easy to know whether a failure occurred due to an order violation or an atomicity violation. Therefore, if the failure is a crash, Lazy Diagnosis generates potential patterns of both order and atomicity violations that may have caused the crash.

Figure 6 shows potential order and atomicity violation patterns that Lazy Diagnosis generates using the instructions from the example in Figure 4, which fails with a crash. Since the failure instruction (I_f) is a read (`load`), and the other instructions (I_1, I_2) are write (`store`) instructions, the likely patterns are a WR order violation, and a WWR atomicity violation. Just as for deadlocks, Lazy Diagnosis determines the root cause of a failure with great certainty in the statistical diagnosis stage (§4.5).

Bug pattern computation uses additional out-of-band information such as thread IDs (e.g., T_1, T_2 in Figure 6) when computing concurrency bug patterns. For instance, bug pattern computation requires instructions involved in an order violation to be executed by different threads. It is possible to use thread IDs, because in practice, the control flow traces are per thread (with the proper kernel driver support).

4.5 Statistical Diagnosis

Statistical diagnosis [39, 51–53] determines the likelihood that the patterns computed by bug pattern computation are the root causes of concurrency bugs (step ⑦, Figure 2).

Statistical diagnosis computes the F_1 *score* [79] for each pattern, which is the harmonic mean of precision (P) and recall (R) ($F_1 = 2\frac{P.R}{P+R}$). In the context of Lazy Diagnosis, precision indicates the number of executions that fail among those executions that were predicted to fail based on the presence of a pattern. Recall indicates the number of executions that were predicted to fail based on the presence of a pattern among the executions that failed.

A high F_1 score for a concurrency bug pattern indicates that the presence of the pattern is positively correlated with the failure. A high F_1 score is also a strong indicator that the pattern is the root cause of a concurrency bug (as we show in our evaluation). The effectiveness of statistical diagnosis is due to the nature concurrency bugs we discussed in §3. In particular, we defined the patterns in Figure 1 as the root causes of concurrency bugs. Consequently, a high F_1 score (i.e., the presence of a pattern) is an indicator that the pattern in question is likely the root cause.

We note that execution traces from successful executions are necessary for increasing the accuracy of statistical diagnosis. In our evaluation, Snorlax was able to accurately diagnose failure root causes by gathering information from 10× more successful executions than failing executions. In practice, traces from successful executions are abundant and easy to obtain for most in-production software.

If there are multiple patterns with the same F_1 score, developers will need to manually identify the root cause from these patterns. We have not seen this in our evaluation (§6).

5 IMPLEMENTATION OF SNORLAX

Snorlax, our current prototype of Lazy Diagnosis, is built to analyze C/C++ programs compiled using clang [22]. We rely on clang to generate an LLVM bitcode file that is used by the server-side analysis. However a different implementation with another program analysis framework is possible [68, 85]. Snorlax also relies on Intel PT for generating control flow traces with timing information. This step too can be implemented on other platforms [14, 58]. Snorlax does not require the program for which it is diagnosing concurrency bugs to be modified in production.

On the client side, Snorlax relies on a custom 3773 LOC Intel PT driver for Linux that exposes an `ioctl` interface for configuring the driver to save the trace when the program executes a specific instruction, or whenever a fail-stop event such as a crash occurs. The driver uses hardware breakpoints (i.e., watchpoints) to detect that the execution reached a specific program counter.

We did not modify the Linux kernel itself, because our driver is a loadable module. We configured Intel PT to hold 64 KB of control trace per thread (configurable up to 128 MB) in memory in a ring-buffer. The ring-buffer mode overrides the control flow trace for each thread once the trace size reaches the buffer size. The ring-buffer mode keeps all the trace in memory until a failure occurs or the trace is saved on-demand, thereby avoiding any I/O overhead of saving the trace to persistent storage during operation.

We configured our Intel PT driver to insert timing packets (MTC and CYC packets [36]) into Intel PT trace at the highest possible frequency. This configuration does not set a specific frequency, but instructs Intel PT to inject as many timing packets as possible. Timing packets occupied on average 49% of the trace buffer size. This does not hurt Snorlax's accuracy as we discuss in the next section. In fact, timing packets allow Snorlax to determine target instruction orderings, thereby increasing root cause diagnosis accuracy.

On the server side, to decode Intel PT traces, we rely on the stock decoder from Intel [37]. We rely on the LLVM compiler toolchain [49] to implement the hybrid points-to analysis as well as the type-based ranking (total of 2618 LOC). For efficiency purposes, programs running in production do not contain the debug information. To mimic this behavior, we strip the debug information from generated binaries and use it on the server side to map the program counter of the failure to the LLVM intermediate representation.

Snorlax limits the maximum number of traces gathered from successful executions for the purposes of statistical diagnosis to 10× of the number of failing executions—an upper limit we empirically determined to be sufficient for full root cause diagnosis accuracy.

The client-server communication code as well as the bug pattern classification and statistical diagnosis modules are written in a total of 1486 LOC of Python. Snorlax clients retrieve the failure code (e.g., crash or hang) from Ubuntu's built-in ErrorTracker [91].

6 EVALUATION OF SNORLAX

In this section, we aim to answer the following questions regarding Snorlax: Can Snorlax accurately diagnose concurrency bugs (§6.1)? How efficient Snorlax is in diagnosing concurrency bugs (§6.2)? How does Snorlax compare to a state of the art bug diagnosis system (§6.3)?

To answer these questions, we evaluate Snorlax using concurrency bugs in real C/C++ systems such as MySQL (650 KLOC), Apache httpd (223 KLOC), memcached (9 KLOC), SQLite (100 KLOC), Transmission (60 KLOC), pbzip2 (2 KLOC), and aget (842 LOC). We use these benchmarks, because they have been used to evaluate previous concurrency bug detection and diagnosis tools [54, 97], and we compare Snorlax to one such tool, namely Gist [42].

On the client side, we ran our experiments with a 2 core Intel Skylake i7-6500U CPU with Intel PT support and 8 GB of RAM running Ubuntu 16.04 with kernel version 4.8.0-41. On the server side, we used an 8 core Intel Xeon E5-1620 CPU with 16 GB of RAM and the same Linux setup.

We used existing test cases to trigger the bugs [97]. For each buggy execution trace, we gathered an additional 10 traces from successful executions at the failure location where each bug manifests itself. On average, Snorlax clients gathered 6764 control events (e.g., branches, calls) and 6695 timing packets per thread.

6.1 Accuracy

In this section, we first evaluate the accuracy of Snorlax by comparing Snorlax's diagnosis results with the bug fix patches of the applications we evaluated. We also compare Snorlax's diagnosis results to those of a state-of-the art concurrency bug diagnosis tool, Gist. We then quantify the contribution of techniques used in Lazy Diagnosis to Snorlax's diagnosis accuracy.

Snorlax was able to accurately diagnose the root cause of all the concurrency bugs we evaluated. We manually analyzed the bug fixes of all the 11 concurrency bugs and found that developers eliminated the patterns that Snorlax diagnosed as the root causes of all these bugs. We also compared Snorlax's diagnosis results with Gist's diagnosis results and confirmed that the root causes diagnosed by Gist and Snorlax are the same. However, in the next section, we show that Gist's root cause diagnosis latency can be orders of magnitude larger than Snorlax's, making Gist less practical for in-production usage.

More formally, we adopt an accuracy metric from prior work [42] called ordering accuracy, namely A_O. A_O defines to what extent the order of target instructions diagnosed by a tool differs from the *ground truth*, i.e., the manually diagnosed and verified order of instructions leading to the failure. Ordering accuracy uses the normalized Kendall tau distance [46], τ, which measures the pairwise disagreements between ordered lists. For instance, for ordered lists of instructions $[I_1, I_2, I_3]$ and $[I_1, I_3, I_2]$, $\tau = 1$, because the pairs $[I_1, I_2]$ and $[I_1, I_3]$ have the same ordering, whereas the pair $[I_2, I_3]$ has different orderings in the two lists. If the ordered list of instructions that Snorlax computes is O_S and the manually computed (ground truth) list of ordered instructions is O_M, then the ordering accuracy is defined as $A_O = 100 * (1 - \frac{\tau(O_S, O_M)}{\#\ of\ pairs\ in O_S \cup O_M})$. We computed the ordering accuracy for all the concurrency bugs Snorlax diagnosed in our evaluation, and obtained 100% for each one.

The accuracy is 100% thanks to the timing packets injected in the control flow trace by Intel PT. For all the bugs we reproduced, timing packets' granularity was fine enough to accurately determine the order of target events leading

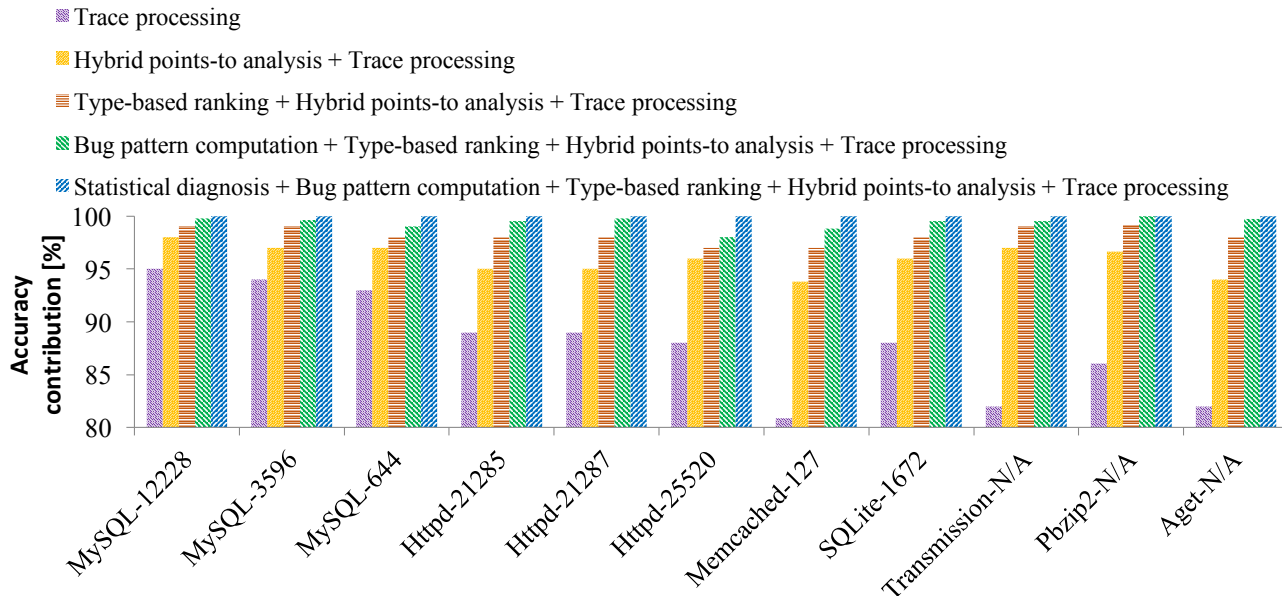

Figure 7: Contribution of each stage of Lazy Diagnosis to Snorlax's accuracy. On the x-axis, we show system names and the bug ids (N/A if bug id is not available). We start the y-axis at 80% to make the contribution of the last four stages clearer.

to concurrency bugs. More specifically, the longest time elapsed between the timing packets in our experiment was 65 microseconds, whereas the shortest time elapsed between target events was 91 microseconds. Our results show that Snorlax was able to leverage the coarse interleaving hypothesis to accurately diagnose concurrency bugs.

Finally, in Figure 7, we show to what extent each stage of Lazy Diagnosis (steps 2–7 in Figure 2) improves Snorlax's accuracy. To quantify this contribution, we compute how much every stage of Lazy Diagnosis reduces the number of instructions to be analyzed. In comparison to a purely static analysis, trace processing reduces the number of instructions to be analyzed by a geometric mean of 9×, contributing 87.9% towards full accuracy. The next big contribution comes from type-based ranking, which further reduces the number of instructions to be analyzed by a geometric mean of 4.6×, contributing an additional 9.7% towards full accuracy. Overall, for all the bugs, each of the five stages is necessary to achieve 100% accuracy.

6.2 Efficiency

In this section, we evaluate Snorlax's efficiency by first measuring the performance overhead of control flow tracing on client executions in production. We then measure how long it takes Snorlax to perform its analysis once a new control flow trace is obtained from a client and how much faster Snorlax is compared to the same static analysis but without the control flow trace (i.e., a whole-program analysis).

To evaluate the performance overhead of control flow tracing using Intel PT, we use tests written by other researchers [97], SQLite's performance tests, Apache's benchmarking tool ab [6], and MySQL's benchmarking tool [65]. To measure performance overhead for MySQL, httpd, and SQLite, we measure the drop in throughput and for all the other benchmarks, we measure the increase in runtime.

We show performance overhead results in Figure 8. Across all programs, control flow tracing incurs a runtime performance overhead of 0.97% on average. The highest average overhead as well as the peak overhead we recorded during our experiments is for pbzip2 at 1.78% and 1.91%, respectively. We conclude that Snorlax incurs low runtime performance overhead in client executions, and therefore it is suitable for in-production usage.

Table 4 shows the average server-side analysis time of Snorlax (steps 2–7 in Figure 2) and the average speedup of the analysis compared to a purely static analysis. After the first full-program static analysis, Snorlax takes on average 2.5 seconds to perform its server-side analysis whenever a new trace is received. Snorlax's hybrid points-to analysis is a function of the trace size and not the program size (aside from constant factors due to pre-processing the program binary), enabling Snorlax to diagnose bugs for large-scale software. The geometric mean of Snorlax's speedup over

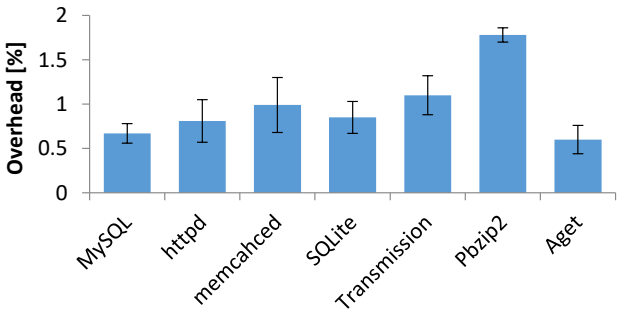

Figure 8: Runtime performance overhead of control flow tracing with Intel PT

	MySQL-12228	MySQL-3596	MySQL-644	Httpd-21285	Httpd-21287	Httpd-25520	Memcached-127	SQLite-1672	Transmission-N/A	Pbzip2-N/A	Aget-N/A
Analysis time	3.4	3.1	3	2.4	2.4	2.7	2.2	2.5	1.9	1.8	1.6
Speed-up vs. static analysis	104x	87x	91x	73x	75x	64x	5x	65x	4x	2x	2x

Table 4: Snorlax's average analysis times (in seconds) and speedups (as a factor) versus the static analysis without the control flow trace. Standard deviations are all below 3%

pure static analysis is 24×. The speedup is greater for larger programs, because the dynamic trace corresponds to a relatively small portion of the program.

6.3 Comparison to the State of the Art

In this section, we compare Snorlax to a state of the art open source concurrency bug diagnosis tool, Gist [42, 83]. First, we list the key differences between Snorlax and Gist in terms of assumptions, practicality, and static analysis. We then argue that Snorlax's concurrency bug diagnosis latency is substantially lower than Gist's. After it, we compare the scalability of Gist and Snorlax as the number of application threads increases, and show that Snorlax scales better. Finally, we compare Gist and Snorlax in terms of their generality.

Intrusiveness: Gist [42] repeatedly modifies the source code to instrument programs in production to gather information for root cause diagnosis, which may not be possible or desirable in practice. Snorlax does not modify or instrument programs running in production, because Intel PT tracks the control flow transparently. This makes Snorlax non-invasive and therefore more practically applicable.

Bug Recurrence Requirement: Gist employs sampling in space by turning monitoring on for a single bug it is trying to diagnose per execution. Consequently, if the number of bugs Gist is trying to diagnose is large, the probability that Gist is monitoring the events for the right bug decreases. For instance, at the time of this writing, Chromium has 684 open race condition bugs [89], which would reduce the root cause diagnosis latency of Gist by a factor of 684 compared to an always on monitoring for all the bugs. Snorlax on the other hand, does not employ sampling and has always-on monitoring, where it captures a control flow trace whenever a failure occurs. Therefore, Snorlax is not impacted by the number of bugs it is trying to diagnose at a given time.

Static Analysis: Snorlax's and Gist's static analysis have different designs and purposes. Gist's static analysis computes a static backward slice which includes all the program instructions that could effect the failing instruction. Gist then *refines* the static slice after every recurrence of the failure to improve bug diagnosis accuracy. Snorlax performs static points-to analysis to identify potential target events. Snorlax then uses control flow traces to determine the execution order of target events.

Diagnosis Latency: Gist requires on average 3.7 recurrences of a failure before it can diagnose a bug [42]. This is because, every time a failure recurs, Gist's analysis iteratively broadens its scope to reduce overhead and increase accuracy. Snorlax is always on and tracing the entire control flow that fits into the Intel PT buffer. Therefore, it requires a failure to occur once to be able to diagnose the bug. In summary, Snorlax has on average at least 3.7× lower bug diagnosis latency compared to Gist. In practice, Snorlax's latency is lower than Gist's by *at least* a further factor equal to the number of bugs being diagnosed. For instance, in the case of Chromium, Snorlax would have on average 2523× lower latency than Gist. In fact, this estimate is conservative, and in practice, Gist's bug diagnosis latency is unbounded. This is because Gist does not know in which executions a previously-observed bug will recur, and a bug for which Gist is trying to perform diagnosis may occur at an arbitrary time in the future.

Scalability: Finally, we measure the scalability of Snorlax and Gist when monitoring client executions in production. For this, we double the application thread count from 2 until we reach 32, while measuring the runtime performance overhead. For each thread count, we represent the average overhead across all applications (i.e., we conflate the overhead) to determine the scalability differences between the two tools for a range of programs.

We show the scalability results in Figure 9. The average overhead of Snorlax increases from 0.87% to 1.98%, because the Intel PT driver has to manage a separate buffer for each thread. Gist has low overhead for low thread counts.

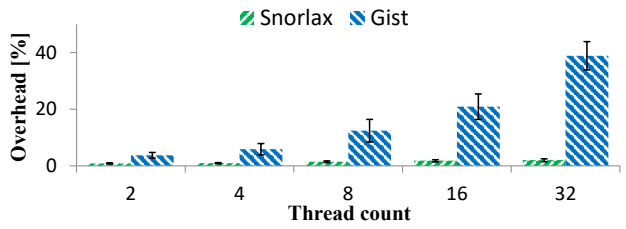

Figure 9: Scalability of Snorlax and Gist with the number of application threads

However, its overhead increases with an increasing thread count. In particular, Gist's overhead increases from 3.14% for 2 threads all the way to 38.9% for 32 threads. The poor scalability of Gist is due to blocking synchronization it employs to track the order of shared memory accesses. Snorlax leverages the coarse interleaving hypothesis to avoid such synchronization and the associated high overhead. We conclude that Snorlax scales better than Gist with the increasing number of application threads.

Generality: Gist does not rely on the coarse interleaving hypothesis to diagnose bugs. Therefore, in theory, it can perform diagnosis for a broader class of bugs than Snorlax. However, Gist's generality comes at the expense of poor scalability and increased overhead, making it less practically applicable.

7 DISCUSSION

In this section, we discuss open questions and current limitations of our prototype Snorlax.

***Applicability of coarse interleaving hypothesis*:** The coarse interleaving hypothesis does not hold for all classes of programs (e.g., a highly contended concurrent data structure, or a highly concurrent program running on a many-core machine). As a result, the coarse interleaving hypothesis cannot be leveraged to build tools that can diagnose concurrency bugs if the events involved in the bug are interleaved at a granularity that cannot be tracked efficiently. However, our evaluation with 54 bugs suggests that the coarse interleaving hypothesis holds for a broad range of bugs in real systems, thereby making it useful. Moreover, even if the coarse interleaving hypothesis does not hold, Snorlax is able to determine target events leading to concurrency bugs without the ordering information, which is still useful in practice.

***Presence of Intel PT*:** Snorlax relies on Intel PT, which is only available on Intel processors since the Broadwell microarchitecture (i.e., after 2014). Lazy Diagnosis is not limited to Intel PT, and it can be implemented using other current technologies such as ARM ETM.

***Limited control flow trace*:** In our experiments, Snorlax uses a 64 KB control flow trace ring buffer, which was sufficient to diagnose all the concurrency bugs with the low overhead numbers we reported. This finding corroborates the short-distance hypothesis from prior work [103], which states that a concurrency bug propagates through a short data/control dependency chain. For bugs where the short-distance hypothesis does not hold, a 64 KB, or even a 128 MB (the largest buffer that we currently support) buffer may not be sufficient. The solution to this limitation presents a performance challenge. We can record an entire Intel PT trace by saving the in-memory trace buffer to persistent storage every time the buffer is full. This solution will increase the runtime performance overhead as well as the storage overhead.

***Type-based ranking heuristic*:** This heuristic may not always reduce the root cause diagnosis latency, especially if the target events involved in a concurrency bug consist of accesses to data via generic pointers (e.g., a pointer to an integer). In our evaluation (§6), the type-based ranking heuristic reduced root cause diagnosis latency by a factor of 4.6×.

***Failing instruction not in the bug pattern*:** Snorlax assumes that the failing instruction is part of the bug pattern. If this assumption is not correct, Snorlax may not be able to diagnose failure root causes. Although recent work [23] reported that 85% of 140 real-world bugs at Microsoft had the failing instruction as part of the bug pattern, we will still explore handling other cases in future work. One possibility is to perform additional static analysis to find instructions with control/data dependencies to the failing instruction and include them in Snorlax's analysis.

***Non fail-stop failures*:** Snorlax is an automated root cause diagnosis system. Therefore it requires the failures to be either fail-stop, or it requires developers to define custom modes of failure (e.g., using assertions) allowing Snorlax to automatically determine that failure has occurred. Snorlax cannot diagnose latent bugs that corrupt a system's state without any externally-observable effect.

***Multi-variable atomicity violations*:** These violations do occur in real systems [56]. Lazy Diagnosis focuses on detecting single-variable atomicity violations. We leave the handling of multi-variable cases to future work.

***Privacy implications*:** Snorlax does not track any data values in the programs it analyzes, but it tracks control flow, which can potentially leak private information. One way to alleviate this problem is to use techniques such as symbolic analysis to anonymize control flow traces [20].

8 RELATED WORK

Lazy Diagnosis' design is influenced by Exterminator [70], which is the first system to suggest collaborative bug fixing

by merging patches generated by multiple users. Similarly, Clearview [75] automatically generates patches to fix vulnerabilities in production. Lazy Diagnosis can assist these techniques in the generation of patches for concurrency bugs.

Lazy Diagnosis' design is also influenced by Windows Error Reporting (WER) [23, 31], which is the first bug diagnosis technique that is widely deployed in production. Neither WER nor any other widely deployed bug diagnosis technique can diagnose concurrency bugs effectively. We believe Lazy Diagnosis can complement WER and other bug diagnosis systems and enable them to diagnose concurrency bugs efficiently and effectively.

SherLog [98] used dynamic and static analysis to automatically generate control and data flow information to diagnose bugs in sequential programs. Conseq [103] computes static program slices and perturbs recorded executions to detect bugs. Lazy Diagnosis can allow SherLog to work for concurrency bugs and lower Conseq's performance overhead.

PRES [74] and HOLMES [21] record execution information such as function calls and use this information for bug diagnosis. PRES uses recorded profiles to perform state space exploration to reproduce failures, and HOLMES performs bug diagnosis. Lazy Diagnosis can be used to improve the efficiency and effectiveness of both techniques.

ProRace [102] detects data races in production by using performance counters, control flow traces and program analysis. Snorlax can do root cause analysis for concurrency bugs other than data races, namely atomicity violations and deadlocks. Furthermore, Snorlax only relies on control flow tracing to perform its offline analysis and performs a multi-stage program analysis that differs from ProRace's analysis.

Delta debugging [101] isolates program inputs and the control flow leading to a failure by repeatedly reproducing a bug. Ochiai [1] and Tarantula [41] record failing and successful executions and replay them to isolate root causes. Lazy Diagnosis does not rely on expensive record/replay techniques nor does it assume bugs can be reproduced.

Castor [60] is a recent record/replay system that relies on commodity hardware support as well as instrumentation to enable low-overhead recording. Castro uses hardware-synchronized time stamp counters to order events without incurring any contention. We believe that Castor constitutes another example of how the coarse interleaving hypothesis can be used to improve the efficiency of an existing analysis (i.e., record/replay in the case of Castor).

As we discussed previously, certain techniques rely on non-commodity hardware extensions for root cause diagnosis [39] and record/replay [63, 67]. Lazy Diagnosis relies on modern commodity hardware.

We borrow statistical analysis of Lazy Diagnosis from prior work on cooperative bug isolation [15, 39, 53]. However, unlike prior work on cooperative bug isolation, Lazy Diagnosis does not rely on statistical sampling for bug diagnosis, which results in low bug diagnosis latency.

Symbiosis [59] and Portend [43, 45] perform symbolic program analysis to synthesize executions. Symbiosis uses synthesized executions for concurrency bug diagnosis, whereas Portend uses such executions to classify data races. Lazy Diagnosis can be used to help such tools synthesize executions faster in the absence of test cases that can reproduce failures.

Dynamic program slicing [82, 90, 93] relies on reproducing failures and tracking the control and data flow information, which is then used for bug diagnosis. Dynamic slicing is expensive and is not suitable for in-production usage. Lazy Diagnosis can provide failing thread schedules to dynamic slicing techniques and improve their efficiency.

Many prior systems have focused on testing and bug detection in concurrent programs [19, 47, 103]. Snorlax is complementary to all these systems, as it can diagnose the root causes of bugs detected by such systems.

9 CONCLUSION

In this paper, we introduced the coarse interleaving hypothesis which states that the events leading to many concurrency bugs are coarsely interleaved. We showed that the coarse interleaving hypothesis holds in a number of real large-scale systems, and the time elapsed between events leading to concurrency bugs is on average five orders of magnitude greater than what fine-grained recording provides. We leverage the coarse interleaving hypothesis to design Lazy Diagnosis, a hybrid dynamic-static program analysis technique for diagnosing concurrency bugs in real-world software. Our Lazy Diagnosis prototype, Snorlax, can diagnose concurrency bugs in large-scale software with full accuracy. Lazy Diagnosis has lower overhead, better scalability, and lower bug diagnosis latency than a state-of-the-art bug diagnosis system. Unlike prior work, Lazy Diagnosis does not require any modifications to the source code at any time. We believe that Lazy Diagnosis is suited for diagnosing in-production concurrency bugs.

ACKNOWLEDGEMENTS

We are indebted to our shepherd, Shan Lu, and the anonymous reviewers for their insightful feedback. We are also thankful to Ioan Stefanovici and Miguel Castro for their generous help in improving this paper. This work was supported in part by the Electrical Engineering and Computer Science Department of the University of Michigan.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Pacific Rim Intl. Symp. on Dependable Computing*.

[2] Adobe Systems Inc. 2017. Adobe Crash Reporter. https://helpx.adobe.com/creative-suite/kb/changing-settings-crash-reporter.html.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: principles, techniques, and tools*.

[4] Johnathan Aldrich. 2017. Lecture Notes: Pointer Analysis. https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/pointer.pdf.

[5] Lars O. Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. University of Copenhagen.

[6] Apache Software Foundation. 2010. Apache Benchmark (ab). http://httpd.apache.org/docs/2.0/programs/ab.html.

[7] Apache Software Foundation. 2013. Apache httpd. http://httpd.apache.org.

[8] Apache Software Foundation. 2017. Apache Commons DBCP. https://commons.apache.org/proper/commons-dbcp/.

[9] Apache Software Foundation. 2017. Apache Derby. https://db.apache.org/derby/.

[10] Apache Software Foundation. 2017. Apache Groovy. http://groovy-lang.org/.

[11] Apache Software Foundation. 2017. Apache Log4j. https://logging.apache.org/log4j/2.x/.

[12] Apache Software Foundation. 2017. Apache Lucene. https://lucene.apache.org/.

[13] Apple Inc. 2017. MacOSX CrashReporter. https://developer.apple.com/library/content/technotes/tn2004/tn2123.html.

[14] armetm 2017. ARM Embedded Trace Macrocell (ETM). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[15] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[16] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[17] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional detection of data races. In *Intl. Conf. on Programming Language Design and Implem.*

[18] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *Intl. Conf. on Programming Language Design and Implem.*

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX Conference on Operating Systems Design and Implementation*.

[20] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. 2008. Better Bug Reporting with Better Privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[21] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Intl. Conf. on Software Engineering*.

[22] Clang 2017. The Clang compiler. http://clang.llvm.org/.

[23] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *International Conference on Software Engineering*.

[24] CVE-2016-5195. 2017. Dirty Cow Vulnerability. https://dirtycow.ninja/.

[25] Rayside Derek. 2005. Points-to analysis. *Technical report, MIT CSAIL*.

[26] EnderUnix. 2017. Aget. http://www.enderunix.org/aget/.

[27] Eucalyptus Open Source Project. 2017. Eucalyptus. https://github.com/eucalyptus.

[28] Brad Fitzpatrick. 2013. Memcached. http://memcached.org.

[29] Free Software Foundation Inc. 2017. java_lang_VMSystem.c. https://tinyurl.com/nx28nym.

[30] Jeff Gilchrist. 2017. Parallel BZIP2. http://compression.ca/pbzip2.

[31] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *ACM Symp. on Operating Systems Principles*.

[32] Patrice Godefroid and Nachiappan Nagappan. 2008. Concurrency at Microsoft – An Exploratory Survey. In *Intl. Conf. on Computer Aided Verification*.

[33] Google Inc. 2017. Chrome Error and Crash Reporting. https://support.google.com/chrome/answer/96817?hl=enl.

[34] Joseph Y. Halpern and Judea Pearl. 2005. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Philosophy of Science*.

[35] Intel Corporation. 2013. Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[36] Intel Corporation. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual.

[37] Intel Corporation. 2017. Intel Processor Trace Decoder. https://github.com/01org/processor-trace.

[38] Jenkins Open Source Project. 2017. Jenkins. https://jenkins.io/.

[39] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *International Conference on Object Oriented Programming Systems Languages and Applications*.

[40] Sebastian Burckhardt John Erickson, Madanlal Musuvathi and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Symp. on Operating Sys. Design and Implem.*

[41] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *IEEE/ACM International Conference on Automated Software Engineering*.

[42] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *ACM Symp. on Operating Systems Principles*.

[43] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[44] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *ACM Symp. on Operating Systems Principles*.

[45] Baris Kasikci, Cristian Zamfir, and George Candea. 2015. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *ACM Trans. Program. Lang. Syst.*.

[46] M. G. Kendall. 1938. A New Measure of Rank Correlation. *Biometrika*.

[47] Ali Kheradmand, Baris Kasikci, and Arjan George Candea. 2014. Lockout: Efficient Testing for Deadlock Bugs. In *Workshop on Determinism and Correctness in Parallel Programming*.

[48] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. 21, 7.

[49] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Intl. Symp. on Code Generation and Optimization*.

[50] Nancy G. Leveson and Clark S. Turner. 1993. An Investigation of the Therac-25 Accidents. *IEEE Computer*.

[51] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. In *Intl. Conf. on Programming Language Design and Implem.*

[52] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Sampling User Executions for Bug Isolation. In *The Workshop on Remote Analysis and Measurement of Software Systems*.

[53] Benjamin Robert Liblit. 2004. *Cooperative Bug Isolation.* Ph.D. Dissertation. University of California, Berkeley.

[54] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In *IEEE/ACM International Conference on Automated Software Engineering*.

[55] Linux. 2017. Linux man pages, clock_gettime. https://tinyurl.com/jo2odgl.

[56] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[57] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Intl. Symp. on Computer Architecture*.

[58] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*

[59] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. In *Intl. Conf. on Programming Language Design and Implem.*

[60] Ali Mashtizadeh, Tal Garfinkel, David Terei, David Mazieŕes, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[61] Steve McConnell. 2004. *Code Complete.* Microsoft Press.

[62] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Intl. Symp. on Computer Architecture*.

[63] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. 2009. Capo: A Software-hardware Interface for Practical Deterministic Multiprocessor Replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[64] Mozilla Corporation. 2017. Mozilla rr. http://rr-project.org/.

[65] MySQL 2010. MySQL Benchmark Tool. https://dev.mysql.com/downloads/benchmarks.html.

[66] MySQL 2010. http://www.mysql.com/.

[67] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Intl. Symp. on Computer Architecture*.

[68] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Intl. Conf. on Compiler Construction*.

[69] Netcraft Survey 2013. Netcraft Web Server Survey. http://httpd.apache.org.

[70] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Exterminator: Automatically Correcting Memory Errors with High Probability. *Commun. ACM*.

[71] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*.

[72] Oracle. 2017. Java Development Kit. http://openjdk.java.net/.

[73] Oracle Corp. 2017. Diagnose a Hung Process. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/hangloop002.html.

[74] Soyeon Park, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, Shan Lu, and Yuanyuan Zhou. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *ACM Symp. on Operating Systems Principles*.

[75] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2010. Automatically Patching Errors in Deployed Software. In *Symp. on Operating Sys. Design and Implem.*

[76] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. 2013. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *Intl. Symp. on Computer Architecture*.

[77] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. 2011. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *IEEE/ACM International Symposium on Microarchitecture*.

[78] Transmission Project. 2017. Transmission. https://transmissionbt.com/.

[79] C. J. Van Rijsbergen. 1979. *Information Retrieval*.

[80] Capegmini S.A. 2015. Capgemini World Quality Report 2015-2016. https://www.uk.capgemini.com/thought-leadership/world-quality-report-2016-17.

[81] Caitlin Sadowski and Jaeheon Yi. 2014. How Developers Use Data Race Detection Tools. In *Workshop on Evaluation and Usability of Programming Languages and Tools*.

[82] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[83] Benjamin Schubert and Baris Kasikci. 2017. Bugbase. https://github.com/dslab-epfl/bugbase.

[84] skylake-cache 2013. Skylake specifications. http://www.7-cpu.com/cpu/Skylake.html.

[85] Soot 2017. Soot - A Java optimization framework. https://sable.github.io/soot/.

[86] SQLite 2013. SQLite. http://www.sqlite.org/.

[87] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Intl. Conf. on Programming Language Design and Implem.*

[88] The Associated Press. 2004. General Electric Acknowledges Northeastern Blackout Bug. http://www.securityfocus.com/news/8032.

[89] The Chromium Project. 2017. Chromium Issues. https://bugs.chromium.org.

[90] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: diagnosing production run failures at the user's site. In *ACM Symp. on Operating Systems Principles*.

[91] Ubuntu. 2017. Ubuntu Error. https://wiki.ubuntu.com/ErrorTracker.

[92] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[93] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic Replay

Based Cyclic Debugging with Dynamic Slicing. In *Intl. Symp. on Code Generation and Optimization*.

[94] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *The Fourth USENIX Workshop on Hot Topics in Parallelism*.

[95] Oliver Yang. 2017. Pitfalls of TSC usage. http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/.

[96] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *ACM SIGSOFT European Conference on Foundations of Software Engineering*.

[97] Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *Intl. Symp. on Computer Architecture*.

[98] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[99] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Debugging. In *ACM EuroSys European Conf. on Computer Systems*.

[100] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. 2013. Automated Debugging for Arbitrarily Long Executions. In *Workshop on Hot Topics in Operating Systems*.

[101] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*.

[102] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[103] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.