Towards Production-Run Heisenbugs Reproduction on Commercial Hardware

Shiyou Huang, Bowen Cai, and Jeff Huang, ***Texas A&M University***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/huang

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Towards Production-Run Heisenbugs Reproduction on Commercial Hardware

Shiyou Huang
Texas A&M University
huangsy@tamu.edu

Bowen Cai
Texas A&M University
bowen.cai@tamu.edu

Jeff Huang
Texas A&M University
jeff@cse.tamu.edu

Abstract

We present a new technique, H3, for reproducing Heisenbugs in production runs on commercial hardware. H3 integrates the hardware control flow tracing capability provided in recent Intel processors with symbolic constraint analysis. Compared to a state-of-the-art solution, CLAP, this integration allows H3 to reproduce failures with much lower runtime overhead and much more compact trace. Moreover, it allows us to develop a highly effective core-based constraint reduction technique that significantly reduces the complexity of the generated symbolic constraints. H3 has been implemented for C/C++ and evaluated on both popular benchmarks and real-world applications. It reproduces real-world Heisenbugs with overhead ranging between 1.4%-23.4%, up to 8X more efficient than CLAP, and incurs only 4.9% runtime overhead on PARSEC benchmarks.

1 Introduction

The ability to reproduce software bugs is crucial for debugging, yet due to the often non-deterministic memory races among threads, it is notoriously difficult to reproduce concurrency bugs, *i.e.*, the so-called *Heisenbugs* [15]. Researchers have investigated significant efforts in *record & replay* (RnR) systems aiming to eliminate the non-determinism. However, it remains challenging to deploy an RnR system for production runs. Most existing solutions either are too slow due to the high runtime overhead incurred by tracing the shared memory dependencies, introduce the observer effect that makes the Heisenbugs disappear [17, 20, 31], or require special hardware that does not exist [16, 25, 26, 28, 33].

CLAP [18] introduces the idea of recording only thread-local information (*i.e.*, thread-local *control flow* paths) and then using offline constraint solving to reconstruct the shared memory dependencies. It is a promising solution for reproducing Heisenbugs because it does not record any cross-thread communication (data or synchronization); hence it requires no synchronizations during recording, which not only reduces the runtime overhead but also minimizes the observer effect.

To enable a production-run RnR solution, however, CLAP is still unsatisfactory due to two important challenges. First, although CLAP is much faster than conventional solutions, the runtime overhead incurred by CLAP using software path-recording is as large as 3X, which is unacceptable for most production environments. Second, the constraints generated by CLAP can be too complex to solve. In the worst case, the complexity of the constraints is exponential in the trace size. Despite that SMT solvers (e.g., Z3 [14]) are becoming increasingly powerful, in practice, the constraints can become too large to solve in a reasonable time.

In this paper, we present H3, a new RnR system to reproduce Heisenbugs by extending CLAP with commercial hardware features. Our key observation is that both of the aforementioned challenges can be effectively addressed by hardware-supported *control-flow* tracing. As also indicated in the CLAP paper [18], for path recording, hardware techniques [30] can achieve as low as 0.6% overhead. In reality, recent Intel processors (starting from the 5th generation) have provided a new feature called Processor Tracing (PT) to trace the program control flow with very small (less than 5%) runtime overhead [2]. PT uses highly-compacted packets (*i.e.*, only one bit for each conditional branch) to capture branch outcomes, often producing a compact trace requiring < 1 bit per retired assembly instruction. Moreover, hardware-supported tracing allows us to perform a significant reduction of the constraints generated by CLAP, because memory accesses executed on each core are ordered internally. We develop a core-based constraint reduction technique that reduces the complexity of the constraints from exponential in the trace size to only *exponential in the number of cores*.

As illustrated in Figure 1, H3 consists of two phases.

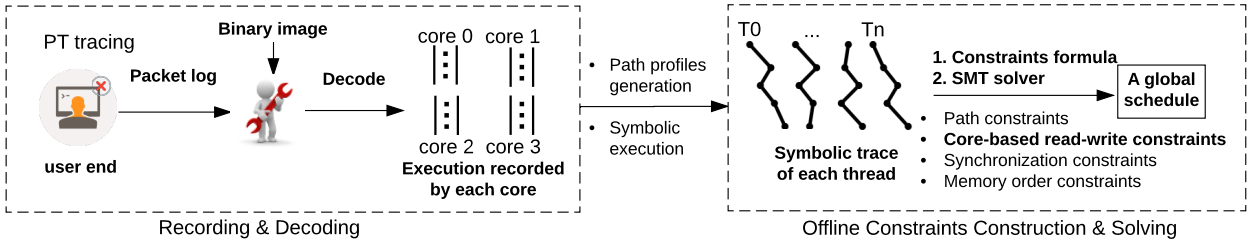

Figure 1: H3 Overview.

First, users run the target program on a COTS (commercial off-the-shelf) hardware with PT enabled. Once a failure occurs, the PT trace together with the thread context switch log are sent to the developer for reproducing the bug. From the PT trace and the binary image of the target program, H3 generates the instructions executed on each core. Second, H3 infers the instructions executed by each thread based on the thread context switch log and generates a symbolic trace for each thread. It then constructs symbolic constraints with the core-based constraint reduction, and computes a global failure reproducing schedule with an SMT solver.

Despite a clear design, realizing H3 faces two main additional technical challenges: 1) How to transform the low-level hardware trace to a high-level (source or IR) trace? 2) How to capture the data values (PT does not trace data values)? To solve the first challenge, we transform the hardware trace into a sequence of IR-level tuples, to identify what basic blocks are executed by each thread. This is done by matching the low-level assembly instructions in the per-thread local execution with that in the IR (*i.e.*, LLVM bitcode). For the second challenge, we symbolically execute the IR along the sequence of basic blocks for each thread. The unknown data values (including all the unknown read values and addresses) are encoded as symbolic variables, and are computed via constraint solving.

We implemented H3 for C/C++ programs based on PT, and evaluated it with a collection of popular performance benchmarks and real-world applications containing known Heisenbugs. Our experimental results show that H3 incurs only 1.4% to 23.4% runtime overhead for all the applications and only 4.9% for the PARSEC benchmarks on average, as much as 8X more efficient than CLAP. Moreover, H3 reduces the size of the constraints in CLAP by 28% to 99%, improving the speed of constraint solving by 2X-250X in most cases, and enabling H3 to reproduce more bugs than CLAP within a limited time budget.

This paper makes the following contributions:

- To our best knowledge, H3 is the first technique that integrates hardware control flow tracing with offline symbolic analysis for reproducing production-run Heisenbugs on commercial hardware.
- We develop a new core-based constraint reduction technique that significantly reduces the complexity of generated symbolic constraints from exponential in the trace size to exponential in the core counts.
- We implement and evaluate H3 on both popular benchmarks and real applications. Experiments show that H3 can reproduce real Heisenbugs in production runs with very small overhead.

2 Background

In this section, we first review the CLAP technique and elaborate its limitations. We then show how hardware control-flow tracing addresses these limitations.

2.1 CLAP

CLAP can not only reproduce Heisenbugs under sequential consistency (SC), but also a wide range of weak consistency memory models, including TSO (total store order) and PSO (partial store order) [9]. It has two key components: I) collecting per-thread control flow information via software path-recording (using an extended Ball-Larus path-recording algorithm [11]), and II) assembling a global schedule by solving symbolic constraints constructed over the thread local paths. To assemble a global schedule, CLAP has three steps:

1. Along the local path of each thread, it collects all the critical accesses (read, write or synchronization) to shared variables.
2. It introduces a fresh symbolic value for each read access, and collects the path constraints following the control flow for each thread via symbolic execution; it introduces an order variable for each critical access, and generates additional constraints according to synchronization, memory-consistency model, and potential inter-thread memory dependencies.
3. It uses an SMT solver to solve the constraints, to which the solutions correspond to global schedules that can reproduce the error. In other

words, the SMT solver computes what inter-thread memory dependencies would satisfy the memory-consistency model and enable the recorded local execution path.

CLAP contains several components to model a failing execution as constraints (*e.g.*, failure, path, synchronization, read-write, and memory model). We next use an example in Figure 2 to illustrate these constraints. Section 3.3 presents the constraint model in detail.

The program in Figure 2 contains a real Heisenbug that only manifests under the PSO memory model, which caused a $12 million financial loss in the real-world [7]. The root cause of the bug is that the write to z (line 5) can be reordered with the writes to x and y (lines 3-4) under PSO. The dashed arrow in the figure shows that the satisfaction of the *if* condition at line 7 depends on the write to z at line 5, which always happens after lines 3 and 4 under SC. However, under PSO, the write to z is allowed to happen before the write to y at line 4. As a result, when the *if* condition is satisfied, the value of $x+1$ and y may be unequal and hence triggering the error. The error can be triggered by the following PSO schedule: 1-2-3$_{R_x}$-3$_{W_x}$-4$_{R_y}$-5-7-8$_{R_x}$-8$_{R_y}$ (the subscripts are used to distinguish different operations from the same line).

The CLAP constraints for reproducing the buggy PSO schedule are shown in Figure 3. We use the order variable O_i denotes the order of the corresponding access at line i. The symbolic variable R_v^i denotes the value returned by the read access to the variable v at line i, and W_v^i the value written to v by the write at line i. To distinguish different operations at the same line, we add the type of the operation to the order variable. For example, $O_3^{R_x}$ and $O_3^{W_x}$ represent the orders of the read and write to x at line 3, respectively.

To manifest the error, CLAP enforces the assertion to be violated while satisfying the path constraints, i.e., $true \equiv (R_z^7 = 1 \wedge R_x^8 + 1 \neq R_y^8)$. A major part of the CLAP constraints is the read-write constraints, which are used to capture the potential inter-thread memory dependencies. Because the order of the memory accesses from different threads is unknown, the read-write constraints must encode a schedule for every potential read-write match, in which the read returns the value written by the write. For example, the read of z at line 7, R_z^7, may be matched with either the initial value 0, or the value written by line 2 or 5. If the former, the read R_z^7 should happen before all the writes to z; if the latter, R_z^7 should be matched with the corresponding write. For example, if R_z^7 returns the value by the write at line 2, the constraint $R_z^7 = W_z^2 \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)$ is generated.

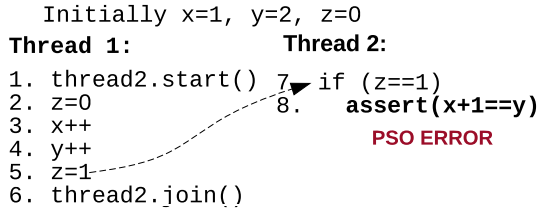

Figure 2: A real PSO bug in an electron microscope software [7], which caused a $12 million loss of equipment.

Read-Write Constraints

$$(R_z^7 = 0 \wedge O_7 < O_2) \vee (R_z^7 = W_z^2 \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)) \vee (R_z^7 = W_z^5 \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC	PSO
$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x} < O_5 < O_6$ $O_7 < O_8^{R_x} < O_8^{R_y}$	$O_1 < O_2 \quad O_5 < O_6$ $O_3^{R_x} < O_3^{W_x} \quad O_4^{R_x} < O_4^{W_x}$ $O_7 < O_8^{R_x} < O_8^{R_y}$

Path Constraints	Failure Constraints
$R_z^7 = 1$	$R_x^8 + 1 ! = R_y^8$

Figure 3: The CLAP constraints for reproducing the PSO error in Figure 2. To save space, we show the read-write constraints for z only. Those for x and y are similar.

CLAP Limitations

1. Exponential complexity of read-write constraints. The read-write constraints generated by CLAP are very complicated in practice because there may exist many writes that a read can be matched with. In the worst case, the complexity of the read-write constraints (*i.e.*, the space of scheduling choices) is exponential in the number of writes (which typically accounts for a large percentage of the events in the trace). This is a bottleneck in CLAP especially for programs with intensive inter-thread memory dependencies, because the SMT solver may fail to solve the constraints. We will present a detailed complexity analysis in Section 3.4.

2. Slowdown of software path-recording. CLAP uses a highly optimized algorithm (i.e., Ball-Larus [11]) to track the control flow information for each thread. Although it greatly reduces the runtime overhead incurred by many other RnR solutions, it still incurs 10%-3X performance slowdown on popular benchmarks [18]. For instance, for the example in Figure 2, when the code is executed in a loop for 10 million times, CLAP incurs 2.3X program slowdown.

3. Difficulty of code instrumentation. It is difficult to apply software path-recording in production runs because it requires code instrumentation. Real-world programs often rely on external libraries, proprietary code, and/or are composed from layers of frameworks and extended by third-party plugins. Tracing the whole pro-

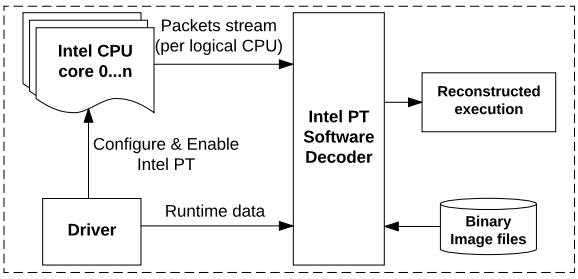

Figure 4: Components of Intel Processor Tracing (PT).

gram control flow by code instrumentation is difficult or impossible. For example, if a failure is caused by a bug in the uninstrumented external code, the constraints generated by CLAP may be incomplete and hence fail to reproduce the bug.

2.2 Hardware Control-Flow Tracing

Tracing control flow at the hardware level opens a door to apply CLAP in production runs by addressing the aforementioned limitations in three ways. First, hardware-supported control flow tracing is significantly more efficient than software-level path-recording. Compared to the 10%-3X overhead by software path-recording, PT achieves as low as 5% runtime overhead [2]. Second, hardware can track the full control flow of the code executed on each core. PT can not only trace the application code, but also the whole operating system kernel [2]. Third, tracing the control flow on each core enables a significant reduction of the complexity of the read-write constraints, because reads and writes from the same core are ordered already.

Next, we first review the basics of PT and then show its performance improvement over software path-recording on PARSEC 3.0 benchmarks [5].

Intel PT. As depicted in Figure 4, PT consists of two main components: tracing and decoding. For tracing, it only records the instructions that are related to the change of the program control flow and omits everything that can be deduced from the code (*e.g.*, unconditional direct jumps). For each conditional branch executed, PT generates a single bit (1/0) to indicate whether a conditional branch is taken or not taken. As such, PT tracks the control flow information, such as loops, conditional branches and function calls of the program, with minimal perturbation, and outputs a highly compact trace.

For decoding, PT provides a decoding library [1] to reconstruct the control flow from the recorded raw trace. It first synchronizes the packet streams with the synchronization packets generated during tracing, and then iterates over the instructions from the binary image to identify what instructions have been executed. Only when the decoder cannot decide the next instruction (e.g., when it encounters a branch), the raw trace is queried to guide the decoding process.

Table 1: Runtime and space overhead of PT on PARSEC.

Program	Native time (s)	PT time (s)	PT **OH(%)**	PT trace
bodytrack	0.557	0.573	**2.9%**	94M
x264	1.086	1.145	**5.4%**	88M
vips	1.431	1.642	**14.7%**	98M
blackscholes	1.51	1.56	**9.9%**	289M
ferret	1.699	1.769	**4.1%**	145M
swaptions	2.81	2.98	**6.0%**	897M
raytrace	3.818	4.036	**5.7%**	102M
facesim	5.048	5.145	**1.9%**	110M
fluidanimate	14.8	15.1	**1.4%**	1240M
freqmine	15.9	17.1	**7.5%**	2468M
Avg.	4.866	5.105	**4.9%**	553M

PT is configurable via a set of model-specific registers by the kernel driver. It provides a privilege-level filtering function for developers to decide what code to trace (*i.e.* kernel vs. user-space) and a CR3 filtering function to trace only a single application or process. PT on Intel Skylake processors also supports filtering by the instruction pointer (IP) addresses. This feature allows PT to selectively trace code that is only within a certain IP range, which can further reduce the tracing perturbation.

PT Performance. Table 1 reports the runtime and space overhead of PT on the PARSEC 3.0 benchmarks. We report the execution time of the programs without and with PT tracing (and the trace size), marked as native and PT respectively. Among the 10 benchmarks, PT incurs 1.4% to 14.7% runtime overhead (4.9% on average) and 88MB to 2.4GB space overhead (0.5GB on average).

3 H3

In this section, we present the technical details of H3. As we have described in Figure 1, H3 integrates hardware control-flow tracing with offline symbolic constraint analysis to reproduce Heisenbugs. Although the overall flow is easy to understand, there are three technical challenges in the integration:

1. **Absence of the thread information**. There is no thread information from the PT traces. It is unknown which instruction is executed by which thread, and hence difficult to construct the inter-thread synchronization and memory dependency constraints.

2. **Gap between low-level hardware traces and high-level symbolic traces**. The decoded execu-

tion from PT is in the low-level assembly form. However, to construct constraints and to reproduce bugs, we need a high-level symbolic trace containing shared variable accesses and branch conditions.

3. **No data values for memory accesses**. PT only traces control flow information but does not record any data values of memory accesses. To reconstruct the shared memory dependencies, we need a way to match reads with writes without using values.

We present our solutions to these challenges in the next three subsections. We also present a constraint reduction algorithm in Section 3.4 enabled by the partial order of writes per-core, which significantly reduces the complexity of the generated constraints.

3.1 Thread Local Execution Generation

We leverage the context-switch software events (generated by the Linux Perf tool) to distinguish instructions from different threads. Each context-switch event contains three attributes: TID, CPUID, and TIME (*i.e.*, the timestamp of the event). Because PT also generates frequent synchronization packets (including the timestamp information) into the packet stream, we can use the timestamp information to synchronize the context switch events with the PT packets from the same core (*i.e.*, CPUID). Because the timestamp clocks local to each core is precise, the inferred thread identity based on the timestamp information is also precise. Hence, we locate the context switch points in the PT packets on each core by comparing the timestamps, and determine the thread identity of each instruction as the TID attribute of the leading context-switch event.

3.2 Symbolic Trace Generation

In CLAP, the symbolic trace of each thread is generated by symbolic execution along the recorded path profile of each thread. The path profile for each thread is decoded (from the Ball-Larus path encoding [11]) as a sequence of basic block transitions at the LLVM IR level in the form of (*Tid, BasicBlockId*). In H3, we also rely on these high-level per-thread path profiles to collect the symbolic traces, and we extract the path profiles from the low-level PT trace as follows. We first instrument all basic blocks of the target program and assign each a unique identifier. Then we compare the generated assembly code from the instrumented program with the decoded instructions from the PT trace to identify which basic blocks are executed by each thread.

Algorithm 1 shows the process of generating the path profiles for each thread. The algorithm takes as input: (1) the executed instructions and their corresponding line number; and (2) the basic blocks of the control-flow of the program with the *BlockId* and the line number of the first instruction of this block. The algorithm first gets the executed instructions by each thread (line 3) and then matches the line number of the executed instructions with that contained in each basic block (line 4-7). To identify the path profile of a thread, the algorithm iterates over the instructions of each thread to check whether the instruction is the first one of the block by comparing the line number (line 5). If so, we add this block into the path profile as (*Tid, BasicBlockId*).

Algorithm 1 Path profiles generation

Input: $\mathbb{L}$:< *line, insn* > //execute instructions and #line
Input: $\mathbb{B}$: <*line, block_id*> //basic blocks of the paths
Output: $\mathbb{Q}$: <*tid, block_id*>//path profile of each thread

```
1: for each tid do    //traverse each thread
2:     //get the instructions of each thread
3:     ℓ = {𝕊 ⊆ 𝕃 | ∀insn ∈ 𝕊.insn, Tid(insn) = tid}
4:     for each item ∈ ℓ do
5:         if item.line ∈ 𝔹.line then
6:             block_id = 𝔹.get(item.line)
7:             ℚ.add(tid, block_id)
8: return ℚ
```

3.3 Matching Reads and Writes

To reconstruct the shared memory dependencies without data values, similar to CLAP, we construct a system of symbolic constraints over the per-thread symbolic traces. The basic idea is to introduce an order variable for each read/write denoting the unknown scheduling order, and a symbolic variable for each read/address denoting the unknown read value and address. We symbolically execute the program following the recorded per-thread control flow, and constructs constraints over the order and symbolic variables to determine the inter-thread orders and values of reads/addresses.

More specifically, we construct a system of SMT constraints formula, denoted by Φ_g, over the symbolic traces. The computed orders/values from solving Φ_g then correspond to one or more concrete global schedules that can reproduce the Heisenbugs. We note that the computed schedules may be different from that in the failure execution, but any one of them is sufficient to reproduce the Heisenbugs.

Φ_g can be decomposed into five parts:

$$\Phi_g = \Phi_{path} \wedge \Phi_{bug} \wedge \Phi_{sync} \wedge \Phi_{mo} \wedge \Phi_{rw}$$

where Φ_{path} denotes the path conditions by each thread; Φ_{bug} the condition for the bug manifestation; Φ_{sync} the interactions between inter-thread synchronizations; Φ_{rw}

the potential inter-thread memory dependencies; and Φ_{mo} the memory model constraints. The formula contains two types of variables: (1) *V - the symbolic value variables* denoting the values returned by reads; and (2) *O - the order variables* the order of each operation in the final global schedule.

Path Constraints (Φ_{path}). The path constraints are constructed by a conjunction of all the path conditions of each thread, with each path condition corresponds to a branch decision by that path. The path conditions are collected by recording the decision of each branch via symbolic execution.

Bug Constraints (Φ_{bug}). The bug constraints enforce the conditions for a bug to happen. A bug can be a crash segfault, an assert violation, a buffer overflow, or any program state-based property. To construct the bug constraints, an expression over the symbol values for satisfying the bug conditions is generated. For example, the violation of an assertion *exp* can be modeled as !*exp*.

Synchronization Constraints (Φ_{sync}). The synchronization constraints consist of two parts: partial order constraints and locking constraints. The partial order constraints model the order between different threads caused by synchronizations *fork/join/signal/wait*. For example, The *begin* event of a thread t should happen after the *fork* event that starts t. A *join* event for a thread t should happen after the last event of t. The locking constraints ensures that events guarded by the same lock are mutually exclusive. It is constructed over the ordering of the *lock* and *unlock* events. More specifically, for each lock, all the *lock/unlock* pairs of events are extracted, and the following constraints for each two pairs (l_1, u_1) and (l_2, u_2) are constructed: $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

Memory Order Constraints (Φ_{mo}). The memory order constraints enforce orders specified by the underlying memory models. H3 currently supports three memory models: SC, TSO and PSO. For SC, all the events by a single thread should happen in the program order. TSO allows a read to complete before an earlier write to a different memory location, but maintains a total order over writes and operations accessing the same memory location. PSO is similar to TSO, except that it allows re-ordering writes on different memory locations.

Read-Write Constraints (Φ_{rw}). Φ_{rw} matches reads and writes by encoding constraints to enforce the read to return the value written by the write. Consider a read r on a variable v and r is matched to a write w on the same variable; we must construct the following constraints: the order variables of all the other writes that r can be matched to are either less than O_w or greater than O_r.

As discussed in Section 2.1, Φ_{rw} can be complicated because there may exist many potential matches between reads and writes. The size of Φ_{rw} is cubic in the trace size and its complexity is exponential in the trace size. Nevertheless, in next subsection, we show that both the size and complexity of Φ_{rw} can be greatly reduced in H3.

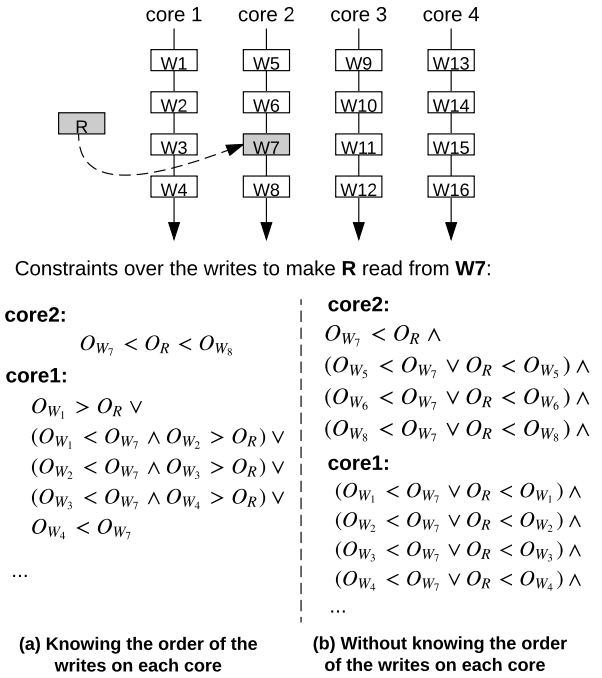

Figure 5: Core-based constraint reduction.

3.4 Core-based Constraints Reduction

Besides the low runtime overhead, another key innovation enabled by PT is that the order of executed events on each core (either by the same thread or by different threads) is determined, which can reduce the complexity of Φ_{rw} from exponential in the number of writes to exponential in the core counts.

The key observation of this reduction is that the executed memory accesses on each core decoded from PT trace are already ordered, following the program order. Once the order of a certain write in the global schedule is determined, all the writes that happen before or after this write, on the same core, should occur before or after this write in the schedule correspondingly. This eliminates a large number of otherwise necessary read-write constraints for capturing the potential inter-thread memory dependencies.

Consider an example in Figure 5, which has four cores with each executing four different writes. Suppose there is a read R that can be potentially matched with all of these writes, because each of them writes a different value to the same shared variable read by R. Without the partial order information of each core, we must include all writes and their orderings into the constraints.

For instance, if R reads the value from the write W_7 on Core 2, then R must happen after W_7 (*i.e.*, $O_R > O_{W_7}$), and all the other writes must either happen before W_7 or after R. Taking W_5 as an example; it must either happen before W_7 or after the read R, resulting in the constraint $(O_R < O_{W_5} \vee O_{W_5} < O_{W_7})$. In general, if there are N writes in the trace, the constraints can generate 2^N different ordering choices for these writes. As typically most accesses in the trace are reads and writes, this exponential search space can be a bottleneck for the technique to scale.

However, with the per-core partial order information, the execution order of the writes on each core is already determined. To prevent other writes from happening between the considered write and read, we only need to take the read-write as a whole and insert it to those sorted writes. Algorithm 2 presents our constraints reduction algorithm. Following this algorithm, to make R read from W_7, for all the other writes on Core 2, we only require $O_{W_7} < O_R < O_{W_8}$. Moreover, for the writes on the other cores, our new constraints encode fewer ordering choices. For example, for the four writes (W_1-W_4) on Core 1, the constraints are written as $O_R < O_{W_1} \vee (O_{W_1} < O_{W_7} \wedge O_R < O_{W_2}) \vee (O_{W_2} < O_{W_7} \wedge O_R < O_{W_3}) \vee (O_{W_3} < O_{W_7} \wedge O_R < O_{W_4}) \vee O_{W_4} < O_{W_7})$. There are only 5 ordering choices (compared to 16 in CLAP).

Algorithm 2 Core-based constraints reduction

Input: a matched read-write $< R, W >$
Output: Φ_{rw} to make R read from W
1: *Initial*: $\Phi_{rw} = \emptyset$
2: ***case 1***: *writes executed on the same core as W*
3: $\Phi_{rw} = \Phi_{rw} \wedge (O_W < O_R < O_{W'})$ //W' happens right after W on the same core
4: ***case 2***: *writes executed on other cores*
5: //for any two writes W_i and W_{i+1} on the same core
6: $\Phi_{rw} = \Phi_{rw} \wedge (O_R < O_{W_i} \vee (O_{W_i} < O_W \wedge O_R < O_{W_{i+1}})) \vee O_W > O_{W_{i+1}}$
7: *return* Φ_{rw}

We note that the core-based constraints apply to SC and TSO, but may not apply to those weak memory models that allow re-ordering of writes on the same core. The reason is that if writes are re-ordered, the partial order witnessed on each core may not reflect the actual buggy execution order.

Theorem 1 below states the soundness guarantee of the core-based reduction:

Theorem 1 *If a concurrent program runs on an SC or TSO platform with C cores and there are N writes executed, the number of the ordering choices of the read-write constraints is reduced from* 2^N *to* $(\frac{N}{C}+1)^C$.

Proof. Consider that a read R returns the value of a write W. When not knowing the partial order of the writes on each core, each write either happens before W or after R. Consequently, there are 2^N ordering choices in total. If the partial order of the writes on each core is known and each core contains $m_i = \frac{N}{C}$ writes, the ordering on each core has only $m_i + 1$ choices. Therefore, the total number of choices is reduced to $\Pi_{i=1}^{C}(m_i+1)$, which equals to $(\frac{N}{C}+1)^C$.

4 Implementation

We have implemented H3 for Pthreads-based C/C++ programs based on a number of tools, including CLAP [18], the Linux Perf Tools [3], the PT decoding library [1], and the Z3 SMT solver [14]. We use Perf to control Intel PT to collect the packet streams and the context switch events. We first insert the context switch events to the packet streams by comparing the timestamp information, and then use the PT decoding library to decode the packets information. As in CLAP, we use KLEE [12] as the symbolic execution engine to generate the symbolic traces for each thread, and construct an SMT constraint formula. We modified CLAP to implement the core-based constraint reduction algorithm, and we use Z3 to solve the constraints.

Shared Variable Identification. We first run a static thread sharing analysis based on the Locksmith [29] race detector and then manually mark each shared variable `x` as symbolic by `klee_make_symbolic(&x, sizeof(x), "x")`, like CLAP. One way to automate this step is to conservatively consider all variables in the program as potentially shared and marked them as symbolic. However, this would produce a large amount of unnecessary constraints. For external function calls that are not supported by KLEE, we also mark the input and return variables of the external function calls as symbolic.

Constraint Reduction. For the core-based constraint reduction, we first extract the writes on the same core from the PT trace and store these writes in a map (*core Id:* $w_1[line], w_2[line]...$). When constructing the read-write constraints, this map is used to determine which write belongs to which core by comparing the associated line number information. Because all writes on the same core occur in the order that they are executed, we construct a happens-before constraint over these writes. When matching a read r to a corresponding write w, we first constrain r to happen after w and happen before the write that occurs right after w on the same core, and we

Table 2: Benchmarks.

Program	LOC	#Threads	#SV	#insns (executed)	#branches (total)	#branches (app)	Ratio app/total	Symb. time
racey	192	4	3	1,229,632	78,117	77,994	99.8%	107s
pfscan	1026	3	13	1,287	237	43	18.1%	2.5s
aget-0.4.1	942	4	30	3,748	313	5	1.6%	117s
pbzip2-0.9.4	1942	5	18	1,844,445	272,453	5	0.0018%	8.7s
bbuf	371	5	11	1,235	257	3	1.2%	5.5s
sbuf	151	2	5	64,993	11,170	290	2.6%	1.6s
httpd-2.2.9	643K	10	22	366,665	63,653	12,916	20.3%	712s
httpd-2.0.48	643K	10	22	366,379	63,809	13,074	20.5%	698s
httpd-2.0.46	643K	10	22	366,271	63,794	12,874	20.2%	643s

then only need to disjunct the order constraints between w and those writes from a different core.

5 Evaluation

Our evaluation of H3 focuses on answering two sets of questions:

- How is the runtime performance of H3? How much runtime improvement is achieved by H3 compared to CLAP?
- How effective is H3 for reproducing real-world Heisenbugs? How effective is the core-based constraint reduction technique?

5.1 Methodology

We evaluated H3 with a variety of multithreaded C/C++ programs collected from previous studies [18, 35, 6], including nine popular real-world applications containing known Heisenbugs. Table 2 summarizes these benchmarks. *pfscan* is a parallel file scanner containing a known bug; *aget-0.4.1* is a parallel *ftp/http* downloading tool containing a deadlock; *pbzip2-0.9.4* is a multithreaded implementation of *bzip* with a known order violation; *bbuf* is shared bounded buffer and *sbuf* is a C++ implementation of the JDK1.4 *StringBuffer* class; *httpd-2.2.9*, *httpd-2.0.48*, *httpd-2.0.46* are from the Apache HTTP Server each containing a known concurrency bug; We also included *racey* [6], a special benchmark with intensive races that are designed for evaluating RnR systems. We use *Apache Bench* (*ab*) to test *httpd*, which is set to handle 100 requests with a maximum of 10 requests running concurrently.

We compared the runtime performance of H3 and CLAP by measuring the time and space overhead caused by PT tracing and software path-recording. We ran each benchmark five times and calculated the average. All experiments were performed on a 4 core 3.5GHz Intel i7 6700HQ Skylake CPU with 16 GB RAM running Ubuntu 14.04.

We evaluated the effectiveness of H3 for reproducing bugs by checking if H3 can generate a failure reproducing schedule and by measuring the time taken by offline constraint solving. We set one hour timeout for Z3 to solve the constraints.

For most benchmarks, the failures are difficult to manifest because the erroneous schedule for triggering the Heisenbugs is rare. Similar to CLAP, we inserted timing delays (*sleep* functions) at key places in each benchmark and executed it repeatedly until the failure is produced. We also added the corresponding assertion to denote the bug manifestation.

Benchmark Characteristics. Table 2 reports the execution characteristics of the benchmarks. Columns 3 and 4 report the number of threads and shared variables, respectively, contained in the execution. We also profiled the total number of the executed instructions and branches in the assembly code, and the branches from the LLVM IR code, as reported in Columns 5-7. Column 8 reports the ratio of the number of the branches in the instrumented application code versus the total number of branches (in both the application code and all the external libraries). For most benchmarks (except *racey*), the ratio is smaller than or around 20%. Column 9 reports the time for constructing the symbolic trace for the corresponding recorded execution of the benchmark.

5.2 Runtime Performance

Table 3 reports the performance comparison between H3 and CLAP. Column 2 reports the native execution time of the benchmarks. Columns 3-4 report the execution time with H3 and CLAP and their runtime overhead. Column 5 reports the speedup of H3 over CLAP. Column 6 reports the percentage of branch instructions in the execution. This number is proportional to the runtime

Table 3: Performance comparison between H3 and CLAP.

Program	Native time (s)	Time (s)			Branch insts%	Space overhead	
		CLAP (**Overhead**)	H3 (**Overhead**)	Speedup		CLAP	H3
racey	0.268	0.768(186.6%)	0.288(**7.5%**)	**65.2%**	6.4%	96M	2.68M
pfscan	0.094	0.104(11.0%)	0.116(**23.4%**)	-11.5%	18.4%	3.2K	30K
aget-0.4.1	0.139	0.156 (12.1%)	0.152(**9.4%**)	**2.6%**	17.9%	11K	41K
pbzip2-0.9.4	0.102	0.134(31.4%)	0.112(**9.8%**)	**16.4%**	14.8%	5.2K	677K
bbuf	0.232	0.696(200%)	0.264(**13.8%**)	**62.1%**	20.1%	3.9K	2.7M
sbuf	0.216	0.299(38.5%)	0.256(**18.5%**)	**14.4%**	17.2%	6.6K	4.5M
httpd-2.2.9	0.53	0.71(34.0%)	0.57(**7.5%**)	**19.7%**	17.4%	7.8M	10.43M
httpd-2.0.48	0.45	0.59(32.1%)	0.51(**13.3%**)	**13.6%**	17.4%	8.1M	11.79M
httpd-2.0.46	0.42	0.57(36.2%)	0.50(**19.0%**)	**12.3%**	17.4%	7.2M	10.62M
avg.	0.272	0.447(64.3%)	0.307(**12.9%**)	**31.3%**	16.3%	13.2M	4.8M

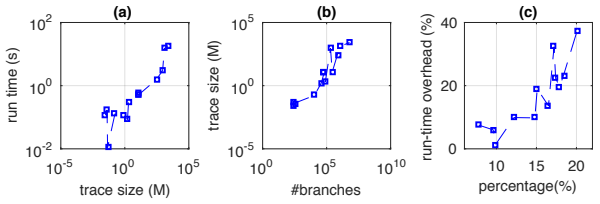

Figure 6: H3 performance analysis.

overhead of PT. Columns 7-8 report the space overhead of H3 and CLAP, respectively.

Overall, the runtime overhead of H3 on these benchmarks ranges between 7.5%-23.4% and 12.9% on average. Compared to CLAP (11.0%-2X overhead), H3 achieves as much as 8X performance improvement and reduces its overhead significantly by 2.6%-65.2% and 31.3% on average. The only exception is *pfscan*. However, this is just because *pfscan* contains significantly more external calls compared to the other benchmarks; while H3 records all external library calls, the implementation of CLAP does not (which sacrifices the correctness). In addition, the short execution time of *pfscan* can suffer from noise.

For space overhead, H3 produces 30KB-2.4GB traces on these benchmarks, whereas CLAP produces 2KB-2.1GB. Some numbers of CLAP are smaller than that of H3, because external library calls are not traced by CLAP.

H3 performance analysis. We note that the performance of H3 is dominated by PT for tracking the control flow events. The additional cost for H3 to track context switching events is almost negligible as compared to tracing the control flow. We have also evaluated the runtime performance of H3 on the PARSEC 3.0 benchmarks and found that H3 incurs only 1.4% to 14.7% runtime overhead (4.9% on average) and 0.5GB trace size, the same as that reported in Table 1 for PT.

We further conducted a performance study of H3 on PARSEC with respect to three impacting factors: the trace size, the number and percentage of branch instructions, as shown in Figure 6. Figure 6(a) shows the relation between the size of the recorded trace and the execution time of H3. Figure 6(b) shows the relation between the number of executed branches and the size of the recorded trace. Figure 6(c) shows that relation between the percentage of executed branch instructions and the runtime overhead of H3. The results indicate that the performance of H3 is proportional to the percentage of executed branch instructions in the execution. Recall Column 8 in Table 2 that the number of branches in the application code often accounts for a small percentage of the total number of branches. Hence, in practice, the performance of H3 can be further improved by tracing only the application code and omitting external library calls.

5.3 Effectiveness of Bug Reproduction

Table 4 reports the results of Heisenbug reproduction. We successfully evaluated five benchmarks[1] with a total number of seven Heisenbugs. *racey1*, *racey2* and *racey3* correspond to the *racey* benchmark with 500, 1000, and 1500 loop iterations.

Column 2 reports the number of unknown variables in the constraint formula, corresponding to the number of read/write/synchronization operations in the symbolic trace. Columns 3-6 report the results of CLAP, including the total size of the generated constraints (in terms of the number of constraint clauses), the size of read-write constraints, the constraint solving time by Z3 and whether Z3 returns a solution before timeout in one hour. Columns 7-10 report the corresponding results of H3.

Overall, H3 is more efficient and effective than CLAP in reproducing Heisenbugs. The key difference between H3 and CLAP is that with the core-based constraint reduction, H3 generates a much simpler and smaller con-

[1]We excluded *aget* and the *httpd* benchmarks because the KLEE symbolic execution failed on them.

Table 4: Results of Heisenbug reproduction. (-) means the solver runs timeout in one hour.

Program	#Var	CLAP #constraints		solve time	success?	H3 #constraints		solve time	success?
		#Total	#RW			#Total	#RW(**Reduction**)		
bbuf	79	14264	13902	98s	Y	10344	9982(**28.2%**)	52s	Y
sbuf	102	438	302	1s	Y	344	208(**31.1%**)	1s	Y
pfscan	25	199	60	1s	Y	179	40(**33.3%**)	1s	Y
pbzip2	113	5890	1270	2s	Y	5460	840(**33.9%**)	1s	Y
racey1	15040	540602	540388	-	N	50602	50388(**90.7%**)	267s	Y
racey2	30108	41612000	41607900	-	N	201202	200788(**99.5%**)	-	N
racey3	67850	1.3×10^8	1.3×10^8	-	N	451802	451188(**99.7%**)	-	N

straint formula than CLAP. H3 reduces the size of the CLAP constraints by 28%-99%, and is able to reproduce more bugs than CLAP. Both H3 and CLAP reproduce the bugs in the four benchmarks *bbuf*, *sbuf*, *pfscan* and *pbzip2*. H3 additionally reproduces the bug in *racey1*, while CLAP fails because the solver could not solve the constraints in time. In addition, for *bbuf*, although both H3 and CLAP can reproduce the bug, H3 is much faster (52s vs 98s) than CLAP. H3 fails on *racey2* and *racey3* because the constraints in these two cases are still too complex to solve.

6 Limitations and Future Work

Our experimental results show that H3 achieves a significant performance improvement over CLAP by integrating hardware control-flow tracing with constraint analysis. Nevertheless, we observe several factors that can be leveraged to further improve the performance of H3.

Large PT Trace Data. On our current platform, the size of the PT trace buffer per core is limited to 4MB. For tracing long running programs, the buffer can get full quickly (e.g., 0.01s for the PARSEC benchmarks). Currently, Perf actively monitors the trace buffer and flushes it to disk once the buffer is full. To avoid overwriting the buffered data, Perf also needs to disable PT when the buffer is full, and wakes it up when the data is copied out. This is a main bottleneck that limits the runtime performance of H3 because the program execution has to be suspended when PT is off, otherwise the control flow data may be lost when the buffer data is being copied out. We also experienced data loss with Perf when using PT to track long traces. This happens because the speed of copying data out is not fast enough, causing certain buffered data overwritten by the new data. We expect that a larger trace buffer or double buffering in the future generations of PT will help alleviate this problem.

Data Values. Another limitation of PT is that it only tracks the control flow of the program but not any data values or memory addresses. This is the main reason why symbolic execution is needed in H3 to construct symbolic traces. Although symbolic execution engines such as KLEE are becoming increasing powerful, scaling symbolic execution to long running programs remains a challenging problem. In addition, limited by KLEE, H3 currently can only reproduce concurrency failures that occur in the application code, but not external function calls (though it traces the control flow in all external libraries).

For future work, we plan to use hardware watchpoints (as also used in Gist [19]) to capture the value and address of variables along with the PT control flow tracing. With the value information, we can then skip the symbolic execution part but construct the constraints by matching the values of reads and writes directly. Moreover, this will further reduce the complexity of the generated constraints.

Constraint Solving for Long Traces. Although our constraint reduction is effective, the complexity of the generated constraints is still exponential in the number of cores. For long traces, the constraint size can still be large and solving them remains challenging. For example, H3 failed on *racey2* and *racey3* due to the solver timeout. For this problem, we plan to improve H3 in two ways. First, we can perform periodic checkpoints (e.g., using the snapshot mode of Perf) to save the current state of the program, such that when a failure occurs, H3 needs only to generate the constraints from the last checkpoint to the failure. Second, we can reduce the amount of the trace by not tracing the control flow in the external libraries (e.g., using the IP filtering featured supported by Skylake processors). As shown in our experimental results, the branches from the application code account for only a small percentage (7-20%) of the total trace, most of which are from the external libraries. Skipping tracing the external libraries will greatly reduce both the trace size and the runtime overhead.

Non-deterministic Program Inputs. Similar to CLAP, currently H3 does not record the program input but assumes that all program inputs are fixed. If the program input is non-deterministic or certain program inputs are missed, H3 may fail to reproduce the bug. This prob-

lem can be addressed by tracking the program input and enforcing the same input value during the symbolic trace construction and the bug reproduction. Mozilla RR [4] is a promising solution to track non-deterministic inputs in real-world systems, by tracing only system call results and signals with `ptrace`. We expect that by integrating H3 with RR, H3 will be able to reproduce failures resulted from both non-deterministic schedules and inputs.

7 Related Work

Researchers have proposed many different RnR systems, both at the software level [8, 13, 17, 18, 21, 22, 23, 27, 32, 34, 38] and hardware-level [16, 25, 26, 28, 33]. Most RnR systems are either order-based [13, 17, 23, 34, 38] that rely on faithfully recording the shared memory dependencies at runtime, or search-based [8, 18, 21, 22, 32] that record only partial information at runtime and rely on powerful search engines such as SMT solvers to reconstruct the memory dependencies.

A central goal of RnR systems is to reduce the runtime overhead such that they can be used in production runs. Hardware techniques [16, 25, 26, 28, 33] are often much more efficient than software-level implementation, but most previous RnR systems rely on special hardware that is not available. Intel PT is an exciting hardware feature that opens a door for RnR systems to be applied broadly in COTS platforms.

Gist [19] introduces a bug diagnosis technique that also leverages PT to identify root causes of a failure with low overhead. Different from H3, Gist assumes the failure can be reproduced in the first place, but it may fail to do so. In addition, Gist relies on statistical analysis to identify failure causes, but it has no guarantee, i.e., it may miss real causes or report false positives. Compared to Gist, H3 solves a different problem: reproducing failures before they can be diagnosed, and H3 is sound: it guarantees to reproduce the failure as long as the constraints can be solved by the solver.

Arulraj *et al.* [10] use hardware performance counters for failure diagnosis. This technique leverages the hardware to sample predicates from a large number of successful and failing runs and then use the sampled predicates to diagnose the failure via statistical analysis.

ReCBuLC [36] uses hardware clocks that are available on modern processors to help reproducing Heisenbugs. The recorded timestamps local to each thread together with a statistical analysis for calculating the time differences among local clocks across different cores, are used to determine the global schedule of shared-resource accesses. One limitation of this approach is that the statistical analysis may fail to infer a correct global schedule.

The idea of using offline constraint analysis to infer global failure schedules was pioneered by Lee *et al.* [21, 22]. The technique uses load-based checkpoints to search for a global schedule without recording any shared memory dependencies. However, compared to PT, the load-based checkpoints are not supported by the commodity architecture.

Similar to CLAP, both ODR [8] and Symbiosis [24] rely on symbolic constraint solving to figure out schedules that can satisfy certain conditions. ODR uses constraints to reproduce failures, and Symbiosis uses constraints for reducing the schedule complexity.

PRES [27] proposes a probabilistic replay technique that uses an intelligent feedback-based replayer to reproduce failures with lightweight recording. PRES may fail to reproduce the bug in the first attempt due to a recorded incomplete schedule. However, it can learn from the previous failing replays to rectify the schedule. Typically after a few attempts, PRES is able to find a correct schedule to reproduce the bug.

Both CoreDump [32] and ESD [37] rely on only the program coredumps to diagnose failures. CoreDump uses a technique called execution indexing to compare the differences between coredumps from failing and normal runs to identify the failing point. ESD uses static analysis and symbolic execution to synthesize both program inputs and schedule to reproduce failures. Using coredumps is promising for diagnosing real-world failures since coredumps are often available after the program crash. However, since there is no program control flow information, the technique may be difficult to reproduce failures that require complex paths and schedules to manifest.

8 Conclusion

We have presented H3, a novel technique that reproduces Heisenbugs by integrating hardware control flow tracing and symbolic constraint solving. With the efficient control flow tracing supported by PT, H3 enables for the first time the ability to efficiently reproduce Heisenbugs in production runs on commercial hardware. We have also presented an effective core-based constraint reduction technique that significantly reduces the size of the symbolic constraints and hence scales H3 to larger programs compared to the state-of-the-art solutions. Our evaluation on both popular benchmarks and real-world applications shows that H3 can effectively reproduce Heisenbugs in production runs with very small overhead, 4.9% on average on PARSEC.

Acknowledgement

We would like to thank our shepherd, Gilles Muller, and the anonymous reviewers for their valuable feedback. This work was supported by NSF award CCF-1552935.

References

[1] Intel processor trace decoder library. https://github.com/01org/processor-trace.

[2] Intel PT Micro Tutorial. https://sites.google.com/site/intelptmicrotutorial.

[3] Linux perf documentation. https://github.com/torvalds/linux/tree/master/tools/perf.

[4] Mozilla rr. https://github.com/mozilla/rr.

[5] The PARSEC benchmarks. http://parsec.cs.princeton.edu/.

[6] Racey: A stress test for deterministic execution. http://pages.cs.wisc.edu/~markhill/racey.html.

[7] A real-world bug caused by relaxed consistency. http://stackoverflow.com/questions/16159203.

[8] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[9] T. S. architecture manual. Version 9. *SPARC International, Inc.* 1994.

[10] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[11] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, 1996.

[12] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2008.

[13] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2013.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE Computer Society, 1986.

[16] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[17] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2010.

[18] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

[19] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, 2015.

[20] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2012.

[21] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline symbolic analysis to infer total store order. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 357–358. IEEE, 2011.

[22] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2009.

[23] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via tightly bounded recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2015.

[24] N. Machado, D. Quinta, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. *ACM Trans. Softw. Eng. Methodol.*, 25(2), Apr. 2016.

[25] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[26] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[27] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[28] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[29] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 2006.

[30] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO, 2005.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[32] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[33] M. Xu, R. Bodik, and M. D. Hill. A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 122–133. IEEE, 2003.

[34] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. Order: Object centric deterministic replay for java. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC, 2011.

[35] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[36] X. Yuan, C. Wu, Z. Wang, J. Li, P.-C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, and Y. Guan. Recbulc: Reproducing concurrency bugs using local clocks. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE, 2015.

[37] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, 2010.

[38] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, 2012.