Infrastructure-Free Logging and Replay of Concurrent Execution on Multiple Cores

Kyu Hyung Lee, Dohyeong Kim, and Xiangyu Zhang

Department of Computer Science, Purdue University, West Lafayette, IN, 47907, USA
{kyuhlee,kim1051,xyzhang}@cs.purdue.edu

Abstract. We develop a logging and replay technique for real concurrent execution on multiple cores. Our technique directly works on binaries and does not require any hardware or complex software infrastructure support. We focus on minimizing logging overhead as it only logs a subset of system calls and thread spawns. Replay is on a single core. During replay, our technique first tries to follow only the event order in the log. However, due to schedule differences, replay may fail. An exploration process is then triggered to search for a schedule that allows the replay to make progress. Exploration is performed within a window preceding the point of replay failure. During exploration, our technique first tries to reorder synchronized blocks. If that does not lead to progress, it further reorders shared variable accesses. The exploration is facilitated by a sophisticated caching mechanism. Our experiments on real world programs and real workload show that the proposed technique has very low logging overhead (2.6% on average) and fast schedule reconstruction.

Keywords: Software reliability, Debugging, Recording and Replay

1 Introduction

Logging and replay of concurrent execution in multi-core environment is very meaningful for debugging runtime failures and also very challenging. Much of the complexity stems from non-determinism that arises from the true parallel evaluation; the non-deterministic fine-grained interleavings are often difficult to precisely reproduce when replaying an erroneous execution. The challenge is exacerbated in the context of non-trivial production runs, in which a program may run for a while before a non-deterministic failure occurs and complex hardware/software infrastructure support for logging and replay is often not available.

Even though there have been a lot of recent efforts in testing, reproducing, diagnosing, and repairing concurrency bugs, existing techniques fall short in logging and replay of real concurrent production execution. Concurrency testing techniques [20, 24, 28] perform various guided searches of possible thread interleavings. They often assume that the failure inducing inputs are provided so that they can repetitively execute the program on these inputs. They do not log or replay the I/O interactions of the original failing execution. However, for production runs, inputs are often very complex, involving network packets, signals, and relying on specific file system state, which requires logging.

Another line of work is to record the order of instructions that access shared state, when they are executed in parallel on different cores. However the entailed instruction-level monitoring [5, 8, 31, 21, 19, 33, 11] is expensive and often requires hardware or complex software infrastructure support, limiting its applicability.

PRES [25] is a technique that uses dynamic binary instrumentation framework called PIN [18] to log different levels of runtime information of a failing run, such as system calls, synchronizations, and even basic blocks. It then tries to reproduce the failure on top of PIN using such information. If it fails, it switches to performing bounded search of shared memory access schedule, supported by the log. However, the need of infrastructure support such as PIN makes it difficult to be used for production runs and causes high logging overhead. We have also found that the bounded search of shared memory access schedule could be very expensive for long runs due to the large search space.

In this paper, we aim to develop a logging and replay technique for execution on multiple cores, serving both software users and developers. It does not require any extended hardware or complex software infrastructure, but rather operates directly on compiled binaries. It features a very low logging overhead as it does not try to log the precise non-deterministic access level interleavings. Replay is a cost-effective search process that produces a deterministic schedule leading to the failure. The produced schedule is for a single core, to allow easy application of follow-up heavy-weight analysis (e.g. slicing [1]) to the failing execution, as most such analysis are for single core execution. *Users* can easily apply our logging component to production runs of deployed software. Logs can be submitted to *developers* for remote reproduction, saving the trouble of manually crafting the failure inducing inputs. *Users* can also choose to reproduce the schedule on their side before submitting a bug report, which would substantially lower the burden of developers. It is very helpful during software development as well since it can be used for in-house testing due to its low system requirement and low overhead.

In our technique, we log minimal information to replay an execution such as non-deterministic system calls, signals and thread spawns in the multi-core logging phase. In the replay phase, we combine I/O replay with schedule exploration to replay concurrency failures on a single core. We leverage the observation by PRES [25] that a lot of non-determinism in a concurrent execution is intentional and thus harmless. It is hence not necessary to faithfully reproduce such non-determinism. Instead,we use the I/O replay log as the validation of an *acceptable* schedule that may be different from the original schedule and yet induces the same failure. The intuition is that if the schedule becomes so different from the original schedule, the program state would differ as well so that variables may have different values and different control flows may be taken. As a result, the replay log becomes invalid, e.g., an event is expected by the replay but not present in the log or an event has different arguments from those recorded in the log. If the replay fails to make progress, we start a process that explores different sub-schedules *within a window* close to the point where the replay fails. We have two layers of exploration, one at the synchronized block level and the other at the fine-

grained memory access level. Any new sub-schedule leading to some progress in replay is admitted to the final schedule. If both explorations cannot find a valid schedule in the current window, we continue to explore preceding windows until we make progress in the replay. We also observe that for long production runs, replay often fails to make progress at similar situations. We hence use caching to speedup exploration. The process is iterative and terminates when the whole log, including the original failure, is successfully replayed.
Our contributions are highlighted as follows.

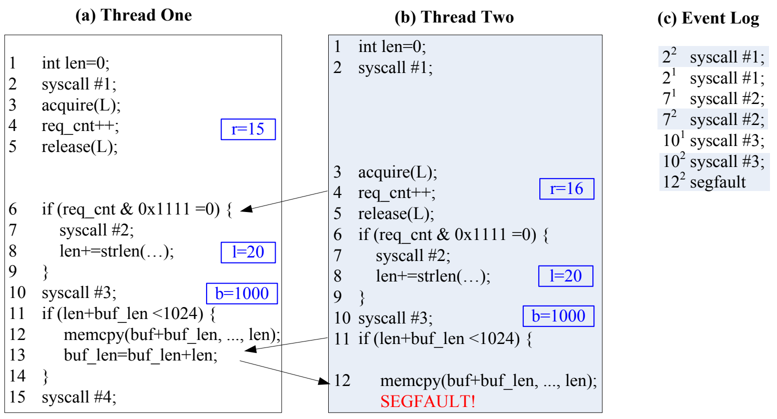
- We develop a logging and replay technique that does not require infrastructure/kernel/compiler support. This makes it more applicable than existing techniques. We also precisely formulate the technique.
- Our logging techinique focuses on minimizing overhead. We only log a subset of system calls, signals and thread spawns. They constitute the minimal necessary set of events to replay an execution. The logging overhead is negligible, 2.6% on average and 3.84% on the worst case.
- We study the characteristics of replaying real concurrent executions of two large subjects with different levels of thread contention and reveal insights about the various reasons why replay fails, which provide critical guidance for our design.
- We propose the notion of window based on the happens-before relation of events. When replay fails to make progress, we perform two layers of schedule exploration only within the window. This strategy allows us substantially reduce the search space.
- We have developed a caching mechanism that can avoid redundant schedule exploration, which is very common in practice.
- We perform thorough evaluation of the technique on a set of real world benchmark programs. The results show that our schedule reconstruction algorithm is very effective and efficient. It is 10.55 times faster than the PRES replay algorithm. We have also demonstrated scalability using a 7-days long real workload.

2 Motivation

In this section, we present the overview of our technique through an example and observations from replaying two large scale multi-threaded applications.

2.1 Motivating Example

Consider the example in Fig. 1. The code snippet is executed by two threads. The example simulates a real bug in the logging module of the *Apache* webserver. Apache logs remote requests for administration purpose. We extend the buggy logic to better explain our technique. Upon a request, the program increases the global request count at line 4. The access is protected by a lock. For every 16 requests (as suggested by line 6), an administrative message is generated and supposed to be put in the log eventually. The message is first stored in a thread local buffer and later copied to a global log buffer; the length of the local buffer *len* is hence updated at line 8. Lines 11-14 copy the local buffer to the global buffer. In particular, it first tests if appending the local buffer



```
1   int len=0;
2   syscall #1;
3   acquire(L);
4   req_cnt++;
5   release(L);

6   if (req_cnt & 0x1111 =0) {
7       syscall #2;
8       len+=strlen(…);
9   }
10  syscall #3;
11  if (len+buf_len <1024) {
12      memcpy(buf+buf_len, ..., len);
13      buf_len=buf_len+len;
14  }
15  syscall #4;
```

```
1   int len=0;
2   syscall #1;

3   acquire(L);
4   req_cnt++;
5   release(L);
6   if (req_cnt & 0x1111 =0) {
7       syscall #2;
8       len+=strlen(…);
9   }
10  syscall #3;
11  if (len+buf_len <1024) {

12      memcpy(buf+buf_len, ..., len);
        SEGFAULT!
```

2^2	syscall #1;
2^1	syscall #1;
7^1	syscall #2;
7^2	syscall #2;
10^1	syscall #3;
10^2	syscall #3;
12^2	segfault

Fig. 1. A segfault caused by concurrent execution on two cores. Different background colors denote different threads. Important variable values are shown on the right of the threads with r, l, b denoting *req_cnt*, *len*, and *buf_len* respectively. Symbol 3^2 denotes line 3 in thread 2.

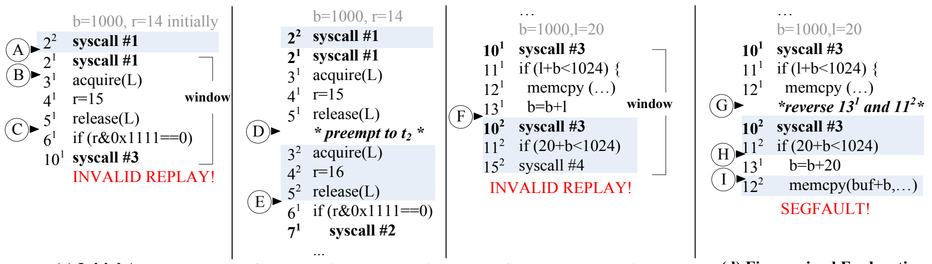


Fig. 2. The different phases of our single core replay scheme.

would overflow the global buffer (with size 1024) at line 11. Variable *buf_len* is the current length of the global buffer. If not, it copies the message and increases the global counter.

Fig. 1 shows a concurrent execution on two cores. The vertical direction is the time line. Observe that statements may be executed at the same time to simulate real concurrency. A few important happens-before are explicitly noted by arrows. We also show the important variable values on the right. Note that both threads observe *req_cnt* to be 16 at line 6. Hence, both threads have a local message of size 20 generated. The local buffer size *len* is 20 in both threads. Because the current global buffer size is 1000, the test at line 11 passes in both threads, allowing copying the messages to the global buffer. However, the global buffer size is increased in thread one before the memory copy in thread two, resulting in copying 20 bytes at the location of 1020 and thus a segfault.

Observe that the failure cannot be easily replayed as it requires two data races, one is about variable *req_cnt* at lines 4 and 6 and the other is about *buf_len* at 11 and 13. If we simply re-execute the program on a single core and assumes thread one executes before thread 2, the message is not even generated in thread one as *req_cnt=15*. As a result, the execution terminates normally. Two preemptions are needed to mutate it to the failing run. However, in production runs, a program usually operates for a long time before a failure. Performing a 2-preemption schedule exploration using techniques like CHESS [20] is prohibitively expensive. Furthermore, although logging happens-before relations between shared variable accesses may allow easy reproduction, it induces very high runtime overhead on production runs.

Our technique only logs system calls, signals and thread spawns in the original execution. This is the minimal set of information we need to replay a concurrent execution. Fig. 1 (c) shows the generated log. A global order of these events is also recorded.

Initial Replay Attempt. Initially, our algorithm tries to replay only based on the global order in the log. It keeps executing a thread. If a synchronization is encountered, e.g. before a lock acquisition or after a release, or a system call is about to execute, the algorithm checks to see if the next event in the log is for a different thread. If so, it context switches to that thread. Fig. 2 (a) shows the initial replay of the log in Fig. 1 (c). According to the log, it starts by executing thread 2. At point Ⓐ, when thread 2 is about to execute the acquisition at line 3, it identifies that the next event in the log belongs to thread 1. Hence, it context switches to thread 1. Points Ⓑ and Ⓒ are also synchronization points, but no switches are needed. At the end, the replay encounters *syscall #3* in thread one while the log has *syscall #2* as the next event. The root cause is that schedule differences cause a different control flow path. The inconsistency indicates that we should revise our schedule.

Coarse-grained Exploration. Our algorithm then explores a different schedule within a window. Intuitively, the execution in the current thread from the last consistent event of the thread to the inconsistent event very likely has undesirable state differences. Such differences could be caused by concurrent execution from other threads. Hence, the window includes all such concurrent execution. The window for the previous inconsistency is shown in Fig. 2 (a). The first phase is to reorder synchronized blocks in the window. Particularly, we try to context-switch to a thread different from that specified by the log. We explore in a backward order, starting from the inconsistent event.

Going backward from *syscall #3* in Fig. 2 (a), the first attempt would be a preemption at point Ⓒ. It results in an execution shown in (b). Note that although the execution is preempted to thread 2 at Ⓓ, it goes back to thread 1 at Ⓔ to respect the event order. As a result, *syscall #2* is correctly encountered. Hence, the preemption is admitted as part of the final schedule.

However, the replay later fails another validity check at the segfault event, as shown by Fig. 2 (c). That is, thread 2 is about to execute *syscall #4* at line 15 but the log indicates a *segfault* event. The root cause is that following the default replay strategy, thread 1 is able to execute lines 10-13 without being interleaved. As a result, the second data race critical to the failure does not occur such that line 11 in thread 2 takes the false branch. The window is determined as shown in the figure (more details about window identification will be disclosed in Section 4). Observe that there are no synchronizations in the window. We hence resort to the fine-grained access level schedule exploration.

Fine-grained Exploration. In this phase, we detect all data races only within the window, and try to reverse the order of the two accesses in a race. The search is also backward. In the window in (c), the race closest to the inconsistent event is the write of *buf_len* at 13^1 (i.e. line 13 in thread 1) and the read at 11^2. Hence, our schedule is enhanced to reverse the order of these two accesses, leading to the execution in (d). Observe that right before the write, at Ⓖ, the algorithm switches to thread 2. Right after the read, at Ⓗ, it switches back to thread 1. At Ⓘ, when thread 1 is about to execute *syscall #4*, it observes that the next event is in thread 2. It switches to thread 2. The segfault occurs at the memory copy statement. The highlighted events and the preemptions in (b) and (d) constitute the final schedule that allows a valid replay and generates the original failure. Note that if both coarse-grained and fine-grained explorations cannot find a schedule to make progress in the current window, we continue to explore preceding windows until we find a valid schedule.

2.2 Observations

Applications	Observed replay failures	Root Cause	CPU contention			Need	Within	Distance (root cause→fail)	
			None	Low	High	Fine-grained?	Window?	# of instructions	# of calls
Apache-1	Unmatch(write,poll)	Control flow	2	3	9	No	Yes	4884	185
Apache-2	Unmatch(poll,write)	Control flow	8	6	17	No	Yes	94	8
Apache-3	Unmatch(argument)	Value	0	1	2	Yes	Yes	1044	15
Apache-4	Unmatch(gettimeofday, read)	Control flow	4	5	14	No	Yes	720	8
Apache-5	Unmatch(read,gettimeofday)	Control flow	1	8	6	No	Yes	834	10
Apache-6	Deadlock	User lock	1	1	2	No	Yes	21	3
Apache-7	Deadlock	Sync order	2	7	11	No	Yes	106	6
Apache-8	Unmatch(segfault,write)	Value	0	0	1	Yes	Yes	631	8
MySQL-1	Unmatch(sigtimedwait,alarm)	Control flow	21	39	47	No	Yes	2032	52
MySQL-2	Unmatch(sigtimedwait,time)	Control flow	2	7	11	No	Yes	33	3
MySQL-3	Unmatch(time,sigtimedwait)	Control flow	12	28	34	No	Yes	84	2
MySQL-4	Unmatch(time,open)	Control flow	4	10	9	No	Yes	952	31
MySQL-5	Unmatch(open,time)	Control flow	9	36	24	No	Yes	15	4
MySQL-6	Unmatch(select,time)	Control flow	13	17	15	No	Yes	412	19
MySQL-7	Deadlock	Control flow	3	3	3	No	Yes	102	11
MySQL-8	Deadlock	User lock	31	45	42	No	Yes	39	3
MySQL-9	Deadlock	Sync order	4	3	5	No	Yes	56	2
MySQL-10	Deadlock	Control flow	0	0	4	No	Yes	36	3

Table 1. Replay failures

In order to motivate the idea, we perform a study on two large scale multi-threaded applications, namely `Apache` and `MySQL`, to understand the characteristics of replaying real concurrency. We execute them on a quad core machine and log the system calls. For `MySQL`, we use the input generated by the work-load emulation client, `mysqlslap`, which is provided with the program. For `Apache`, we use `httperf` to generate 1,000 concurrent requests. We create 4 worker threads for both subjects. Although these are benign executions, replaying them only with the system call logs is nonetheless challenging. Each time when replay fails to make progress due to deadlocks or unmatched events, called a replay failure, we manually study its root cause, leveraging our implementation of CHESS [20] in an interactive way. In particular, the implementation allows us to search backward from the execution point where replay fails to look for a number of preemptions at synchronizations or shared memory accesses that allow us to get through the failure point. For each replay failure, we manually try different configurations of the search (e.g. the distance to search backward and the number of preemptions) until we succeed. We also simulate different levels of thread contention by executing a configurable CPU-intensive threaded program in the background. We have studied three setups: (1) no contention – the subject program owns 100% of the CPU; (2) low – 66%; and (3) high – 50%.

Table 1 presents our observations. Column 2 presents the unique replay failures we have observed and column 3 shows the root cause of each failure. Columns 4-6 present the number of occurrences of each replay failure at different contention levels. Column 7 shows if we need fine-grained exploration to get through the failure. Column 8 shows if we could find a correct schedule in the exploration window (defined in Section 4). Columns 9 and 10 present the distance from the root cause (i.e. the farthest preemption needed) to the replay failure point, measured by the number of instructions and function invocations.

First of all, we observe much fewer replay failures than expected, even with the highest contention level. It seems to indicate that the non-determinism caused by real concurrency does not substantially affect system level behavior. We observe two kinds of replay failure symptoms: *unmatch* and *deadlock*. The former means that replay can not make progress because the event in the log does not match the expectation. In the table, we also present the mismatched events observed. These symptoms are caused by five possible reasons as demonstrated by the samples in Fig. 3. Circled numbers show the order of execution. The replay order is presented on the bottom of each example. Note that the replay is guided by the system call log in these examples. Upon each pthread synchronization or system call, the replay tries to switch to the thread indicated by the next event in the log. In (A), the replay fails at syscall#2 due to control flow difference. This is the most common type. In (B), the system call arguments do not match between the log and the replayed execution at ④. In (C), T1 is waiting for a conditional variable *cond_wait*, which did not happen in the original run. The replay then switches to T2, which cannot replay syscall#3 at ④ without T1 replaying syscall#2, and hence deadlock. In (D), the *spin_lock* function is a program specific lock invisible to our analysis. When T1 is about to execute ⑥, the replay context switches to T2 to respect the log order, but T2 cannot acquire the lock at ②, and hence deadlock. In (E), T1 is waiting for the conditional variable at ①, and then the replay context switches to T3 (hinted by

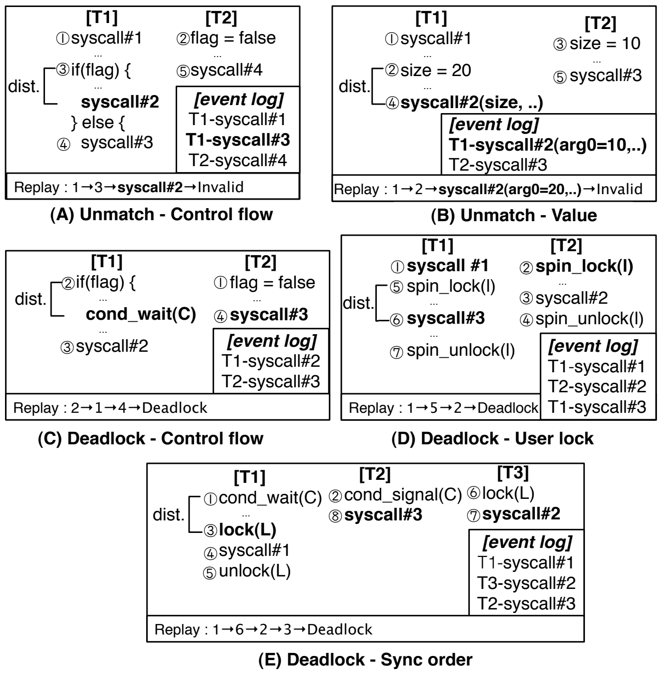


Fig. 3. The root causes of the observed replay failures.

the log order). T3 acquires lock L at ⑥ then it context switches to T2 before ⑦ because T2 is the only available thread at this point, again instructed by the log. Now T2 sends signal to T1 and it context switches to T1 at ⑧. However the mutex is already held by T3 and thus deadlock.

We also observe the following:

- Replay often fails during *normal* execution before it reaches the faulty point.
- Replay fails more often with higher contention.
- Replay tends to fail at the same failure repeatedly.
- Fine-grained exploration is rarely needed.
- Searching within the exploration window is sufficient for all cases we have seen.
- The distance between the root cause of replay failure and the symptom tends to be short.
- We have further observed that the repetition of replay failure is not caused by the re-occurrences of the same input. They are due to nondeterminism of low level shared data structures (e.g. *table* structures in MySQL, *buffered_log* in Apache) that have little to do with input values. In other words, we believe the repetitive behavior will always manifest, regardless of the input. This is supported by our experiment in Section 7.

3 Language and Semantics

To facilitate discussion, we introduce a kernel language. The syntax of the language is presented in Fig. 4. A method can be spawned as a thread. We model devices and I/O

KERNEL-LANGUAGE $\mathcal{L}$

$$
\begin{array}{lll}
\textit{Program } P & ::= & \overline{m()\{s\};} \\
\textit{Dev} \quad d & ::= & \textbf{stdin} \mid \textbf{stdout} \mid f \\
\textit{Expr} \quad e & ::= & x^{\ell} \mid c \mid e_1 \textbf{ binop } e_2 \mid \textbf{read}^{\ell}(d) \\
\textit{Stmt} \quad s & ::= & x :=^{\ell} e \mid \textbf{write}^{\ell}(d,e) \mid s_1 ; s_2 \mid \textbf{spawn}^{\ell}\, m() \mid \\
 & & \textbf{acquire}^{\ell}(k) \mid \textbf{release}^{\ell}(k) \mid \textbf{skip} \mid \textbf{assert}^{\ell}(e) \mid \textbf{fail}
\end{array}
$$

$$\textit{Method } m, \textit{ Var } x, \textit{ File } f, \textit{ Lock } k \in \textit{Identifier} \quad \textit{Constant } c \in Z$$

Fig. 4. Language Syntax

$$
\begin{array}{lll}
\textit{Store} & \sigma : & \textit{Var} \to Z \\
\textit{IOStore} & \iota : & \textit{Dev} \to \overline{Z} \\
\textit{LockState} & \mathcal{K} : & \textit{Lock} \to Z^{+} \cup \{\bot\} \\
\textit{Log} & \mathcal{L} ::= & \overline{\alpha} \\
\textit{LogEntry} & \alpha ::= & \textit{READ}\langle i,t,d,\ell,c\rangle \mid \textit{WRITE}\langle i,t,d,\ell,c\rangle \mid \\
 & & \textit{SPAWN}\langle i,t,\ell\rangle \mid \textit{FAIL}\langle i,t,\ell\rangle
\end{array}
$$

$$\textit{LogEntryId} \;\; i \in Z^{+} \qquad \textit{ThreadId} \;\; t \in Z^{+}$$

Fig. 5. Definitions

with **read**() and **write**(). Failures are modeled as assertion violations. Variables may be accessed by multiple threads.

3.1 Logging Semantics

Compared to other execution artifacts, logging I/O interactions with the environment is necessary as they cannot be constructed by post-mortem analysis. Hence, we log system calls with global timestamps.

Fig. 5 presents definitions for the logging semantics. The device store ι denotes the state of device, which is a mapping from a device to a sequence of constant values. The lock state $\mathcal{K}$ is a mapping from a lock to a thread id or a special value $\bot$, denoting the owner of the lock, or its availability for acquisition, respectively. The evaluation generates a log $\mathcal{L}$, which is a sequence of events. In the semantics, we model the read, write, thread spawn, and assertion failure events. In our real implementation, we log most system calls, thread spawns, and all exceptions such as segfaults. Note that synchronizations or shared variable accesses are not logged in order to achieve the lowest possible logging overhead.

Each log entry consists of a global id i serving as a timestamp, the thread id t, and a label ℓ indicating the program point at which the event happened. For reads and writes, the value being read or written is also logged. Logging read values is to avoid accessing the device during replay. Logging write values is to validate a replay.

The logging semantics are presented in Fig. 6. Expression evaluation is of the form $\boxed{\sigma,\iota,\mathcal{L} : e \xrightarrow{e}_{t} \iota',\mathcal{L}',e'}$, with σ the store, ι the device store, $\mathcal{L}$ the log, and e the expression. The evaluation is carried out in thread t. Devices are modeled as streams. In particular, one value is read at a time from the head of a stream; and a value can be written to the

$E ::= E;s \mid [\cdot]_s \mid x := [\cdot]_e \mid \textbf{write}(d,[\cdot]_e) \mid \textbf{assert}([\cdot]_e) \mid [\cdot]_e \textbf{ binop } e \mid c \textbf{ binop } [\cdot]_e$

Expression Rules $\boxed{\sigma,\iota,\mathcal{L} : e \xrightarrow{e}_t \iota',\mathcal{L}',e'}$ parameterized on the current thread id t

$$\sigma,\iota,\mathcal{L} : x \xrightarrow{e}_t \iota,\mathcal{L},\sigma(x) \qquad \sigma,\iota,\mathcal{L} : c_1 \textbf{ binop } c_2 \xrightarrow{e}_t \iota,\mathcal{L},c_3 \quad \textbf{where } c_3 = c_1 \textbf{ binop } c_2$$

$$\sigma,\iota,\mathcal{L} : \textbf{read}^{\ell}(d) \xrightarrow{e}_t \iota[d \mapsto \chi],\ \mathcal{L}\cdot READ\langle|\mathcal{L}|,t,d,\ell,c\rangle,\ c \quad \textbf{where } \iota(d) = c\cdot\chi$$

Local Statement Rules $\boxed{\sigma,\iota,\mathcal{K},\mathcal{L} :\ s \xrightarrow{s}_t \sigma',\iota',\mathcal{K}',\mathcal{L}',s'}$ parameterized on thread id t.

$$\begin{aligned}
\sigma,\iota,\mathcal{K},\mathcal{L} :\ x :=^{\ell} c &\xrightarrow{s}_t \sigma[x \mapsto c],\ \iota,\ \mathcal{K},\mathcal{L},\ \textbf{skip}\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{write}^{\ell}(d,c) &\xrightarrow{s}_t \sigma,\ \iota[d \mapsto \iota(d)\cdot c],\ \mathcal{K},\ \mathcal{L}\cdot WRITE\langle|\mathcal{L}|,t,d,\ell,c\rangle,\ \textbf{skip}\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{skip};s &\xrightarrow{s}_t \sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ s\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{acquire}^{\ell}(k) &\xrightarrow{s}_t \sigma,\ \iota,\ \mathcal{K}[k \mapsto t],\ \mathcal{L},\ \textbf{skip} \quad \textbf{if } \mathcal{K}(k) = \bot \vee \mathcal{K}(k) = t\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{release}^{\ell}(k) &\xrightarrow{s}_t \sigma,\ \iota,\ \mathcal{K}[k \mapsto \bot],\ \mathcal{L},\ \textbf{skip}\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{assert}^{\ell}(c) &\xrightarrow{s}_t \sigma,\ \iota,\ \mathcal{K},\ \mathcal{L}\cdot FAIL\langle|\mathcal{L}|,t,\ell\rangle,\ \textbf{fail} \quad \textbf{if } c = 0\\
\sigma,\iota,\mathcal{K},\mathcal{L} :\ \textbf{assert}^{\ell}(c) &\xrightarrow{s}_t \sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ \textbf{skip} \quad \textbf{if } c \neq 0
\end{aligned}$$

Global Rules $\boxed{\sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ (s_1,\ldots,s_n) \longrightarrow \sigma',\ \iota',\ \mathcal{K}',\ \mathcal{L}',\ (s'_1,\ldots,s'_n)}$

$$\frac{\sigma,\iota,\mathcal{K},\mathcal{L} :\ s \xrightarrow{s}_t \sigma',\ \iota',\ \mathcal{K}',\ \mathcal{L}',\ s'}{\sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ (s_1,\ldots,s_{(t-1)},\ E[s]_s,\ s_{(t+1)},\ldots,s_n) \longrightarrow \sigma',\ \iota',\ \mathcal{K}',\ \mathcal{L}',\ (s_1,\ldots,s_{(t-1)},\ E[s']_s,\ s_{(t+1)},\ldots,s_n)}$$

[STMT-ANY-THRD]

$$\frac{\mathcal{L}' = \mathcal{L}\cdot SPAWN\langle|\mathcal{L}|,t,\ell\rangle \qquad m()\{s_\Delta\} \in P}{\sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ (s_1,\ldots,s_{(t-1)},\ E[\textbf{spawn}^{\ell}\ m()]_s,\ s_{(t+1)},\ldots,s_n) \longrightarrow \sigma,\ \iota,\ \mathcal{K},\ \mathcal{L}',\ (s_1,\ldots,s_{(t-1)},\ E[\textbf{skip}]_s,\ s_{(t+1)},\ldots,s_n,s_\Delta)}$$

[SPAWN]

$$\frac{\sigma,\iota,\mathcal{L} :\ e \xrightarrow{e}_t \iota',\mathcal{L}',c}{\sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ (s_1,\ldots,s_{(t-1)},\ E[e]_e,\ s_{(t+1)},\ldots,s_n) \longrightarrow \sigma,\ \iota',\ \mathcal{K},\ \mathcal{L}',\ (s_1,\ldots,s_{(t-1)},\ E[c]_e,\ s_{(t+1)},\ldots,s_n)}$$

[EXPR-ANY-THRD]

Fig. 6. Logging Semantics

tail of the stream. More I/O complexity is omitted to simplify the formal discussion. Our implementation supports most system calls and signals. In the evaluation of a read expression, a constant value c is removed from the head of the stream; a read event is appended to the log. Local statement evaluation evaluates program statements in a thread, with the form $\boxed{\sigma,\iota,\mathcal{K},\mathcal{L} :\ s \xrightarrow{s}_t \sigma',\iota',\mathcal{K}',\mathcal{L}',s'}$ with $\mathcal{K}$ the lock state and s the statement. For a write statement, it appends the value c to the end of the stream and a write event to the log. For a lock acquisition, if the lock is available or being held by the current thread, it updates the lock state and allows evaluation to proceed. Note that the lack of an evaluation rule when the lock is held by other threads means that the evaluation of the current thread cannot proceed. The global evaluation will pick another thread to continue. For a lock release, the state of the lock becomes available, which may allow some previously blocked thread to proceed. For an assertion statement, if the assertion fails, a log entry is appended and the whole evaluation terminates (through the **fail** statement). Otherwise, it allows the evaluation to proceed, without adding a log entry.

Global rules $\boxed{\sigma,\ \iota,\ \mathcal{K},\ \mathcal{L},\ (s_1,\ldots,s_n) \longrightarrow \sigma',\ \iota',\ \mathcal{K}',\ \mathcal{L}',\ (s'_1,\ldots,s'_n)}$, denote the evaluation of n threads with each thread i executing statement s_i. Each step corresponds to a change in a single thread i, so $\forall j \neq i, s_j = s'_j$. The choice of which thread advances at any given point is *non-deterministic*, modeling concurrent execution on multiple cores. Terminated threads are left in the list with the **skip** statement. The whole evaluation terminates normally if all threads terminate normally. Rule [SPAWN] spawns a method as a thread, by expanding the list of threads.

In our implementation, each thread has its own log file to avoid contentions on a single log file. The log entry id remains global.

3.2 Replay Semantics

Replay is driven by a log and a schedule. It is deterministic, modeling execution on a single core. Our replay strategy is to evaluate the same thread as much as possible, unless it is indicated by the replay log or the schedule that a context switch should be performed. Initially, the schedule is empty. Replay is carried out following only the replay log. If such basic replay does not succeed, an exploration process is triggered to generate a schedule that can advance more during replay, until eventually all the events in the replay log, including the failure event, are correctly replayed.

The replay log serves the following three purposes. (1) The global timestamps specify a global order. Replay must follow the same order. (2) The values stored in the read events are used as inputs to drive the replay execution, avoiding accessing the devices. (3) The log is also used as a validation of the replayed execution.

Replay is facilitated by a schedule generated by the schedule exploration process to provide an additional harness. It specifies a set of preemptions that are at synchronization primitives. We will extend it to include preemptions at shared variable accesses in Section 4.3. The syntax of a schedule is presented in Fig. 7. It is a sequence of synchronization points. An entry $sync\langle n,t\rangle$ denotes that switching to thread t upon the nth synchronization operation.

<u>ADDITIONAL LANGUAGE SYNTAX</u>

$$\begin{array}{lll}
\textit{Preempt} & \pi ::= & \textit{pevnt} \mid \textit{psync} \\
\textit{DynChk} & \omega ::= & \textit{chkEvnt}(\ell) \mid \textit{chkWrt}(\ell,e) \mid \textit{chkAssrt}(\ell,e) \\
\textit{Expr} & e ::= & \ldots \mid \textit{pevnt}?\ \mathbf{read}^{\ell}(d)\ \mathbf{req}.\ \textit{chkEvnt}(\ell) \\
\textit{Stmt} & s ::= & \ldots \mid \mathbf{invalid\_replay} \mid \\
 & & \textit{pevnt}?\ \mathbf{write}^{\ell}(d,e)\ \mathbf{req}.\ \textit{chkWrt}(\ell,e) \mid \\
 & & \textit{pevnt}?\ \mathbf{spawn}^{\ell}\ m()\ \mathbf{req}.\ \textit{chkEvnt}(\ell) \mid \\
 & & \textit{psync}?\ \mathbf{acquire}^{\ell}(k) \mid \mathbf{release}^{\ell}(k)\ \textit{psync}? \mid \\
 & & \textit{pevnt}?\ \mathbf{assert}^{\ell}(e)\ \mathbf{req}.\ \textit{chkAssrt}(\ell,e)
\end{array}$$

<u>ADDITIONAL DEFINITIONS FOR EVALUATION</u>

$$\begin{array}{lll}
\textit{Schedule} & \mathcal{S} ::= & \overline{sync\langle n,t\rangle} \\
\textit{InstCnt} & n \in & Z^{+}
\end{array}$$

Fig. 7. Definitions for Replay Semantics

New definitions relevant to the replay semantics are presented in Fig. 7. Preemption π denotes a preemption test, which determines whether a preemption should be performed, following the schedule or the log order. There are two kinds of preemption tests for syscalls (*pevnt*) and synchronizations (*psync*), respectively. Dynamic check ω denotes the runtime checks performed to validate a replay. There are three kinds of dynamic checks: checking a write event (*chkWrt*), an assertion failure (*chkAssrt*), and other events (*chkEvnt*).

The syntax of kernel language is extended. Statements and expressions that could produce events in the logging phase are preceded with preemption tests and followed by checks. Additionally, a preemption test precedes each lock acquisition and follows each lock release. Given a program in the original language in Fig.4, one can consider the corresponding program in the extended language is automatically generated.

We also introduce a special counter variable *sync_cnt* to record the number of synchronizations that have been evaluated. We use $\sigma[sync\_cnt \uparrow_c]$ to denote increasing the counter.

The replay rules are presented in Fig. 8. The evaluation order is given on the top. Observe that a preemption preceding an expression/statement is evaluated before the expression/statement, suggesting that the evaluation may switch to a different thread before evaluating the expression/statement. A check following an expression/statement is also evaluated *before* the expression/statement. We have to perform the check first as an expression/statement cannot be properly evaluated if there is any inconsistency. If a preemption test follows a statement (as for the release statement), it is evaluated after the statement evaluation.

Preemption Rules. They have the form $\boxed{\sigma,\mathcal{S},\mathcal{L},t : \pi \overset{\pi}{\rightarrow} \sigma', \mathcal{S}',t'}$. Given store σ, schedule $\mathcal{S}$, replay log $\mathcal{L}$ and the current thread id t, a preemption test evaluates to a new thread id t', together with the new store and schedule. A preemption is indicated by $t' \neq t$. For a preemption test regarding a log event (i.e. *pevnt*), the resulting thread id is the one indicated by the next log event. For a synchronization (i.e. *psync*), if the value of the synchronization counter, acquired by $\sigma[sync\_cnt]$, equals to that specified in the next preemption in the schedule $\mathcal{S}$, it yields the thread id specified in $\mathcal{S}$ ([P-SYNC-PRMPT]). Otherwise, it increases the synchronization count and continues evaluation with the thread specified by the log (P-SYNC-NOPRMPT]).

Checking Rules. The second set of rules is to validate a replay. They are of the form $\boxed{\mathcal{L},t : \omega \overset{\omega}{\rightarrow} b}$. A check ω evaluates to a boolean value b. We define replay validity as follows.

Definition 1 (Replay Validity). *Given a log $\mathcal{L}$, a replay execution is valid if the execution must encounter the exact sequence of events as specified in $\mathcal{L}$.*

It dictates observable equivalence between the original and the replayed runs. Observe that a valid replay must successfully reproduce the same failure as the failure event is part of the log. According to the rules, checking events other than writes and assertions (i.e. *chkEvnt*) is to test whether the program point of the syscall and the current thread id are those specified in the log. To validate a write event (i.e. *chkWrt*), we additionally check the equivalence of the parameter computed in the replay and that in the log. To validate an assertion (i.e. *chkAssrt*), we ensure that if the assertion passes, there is not a *FAIL* event in the log; and if the assertion fails, the appropriate failure event must be present in the log.

Expression and Local Statement Rules. The configurations of expression and local rules are similar to those in the logging semantics. The difference is that the device state ι is not part of the configurations as devices are not accessed during replay. Inputs

$E ::= \ldots \mid [\cdot]_\pi\, e \mid e\, [\cdot]_\omega \mid [\cdot]_\pi\, s \mid s\, [\cdot]_\omega \mid [\cdot]_s\, \pi? \mid \textbf{skip}\, [\cdot]_\pi \mid \mathit{chkWrt}(\ell, [\cdot]_e) \mid \mathit{chkAssrt}(\ell, [\cdot]_e)$

Preemption Rules $\boxed{\sigma, \mathcal{S}, \mathcal{L}, t :\ \pi \overset{\pi}{\to} \sigma',\ \mathcal{S}', t'}$ $\qquad \alpha.t$ denotes the t field of a relation α.

$$\sigma,\ \mathcal{S},\ \alpha\cdot\mathcal{L},\ t :\ \mathit{pevnt}? \overset{\pi}{\to} \sigma, \mathcal{S}, \alpha.t \qquad \sigma,\ \mathit{sync}\langle\sigma(\mathit{sync\_cnt}), t_0\rangle\cdot\mathcal{S},\ \mathcal{L},\ t :\ \mathit{psync}? \overset{\pi}{\to} \sigma, \mathcal{S}, t_0 \qquad \text{[P-SYNC-PRMPT]}$$

$$\sigma,\ \mathit{sync}\langle n_0, t_0\rangle\cdot\mathcal{S},\ \alpha\cdot\mathcal{L},\ t :\ \mathit{psync}? \overset{\pi}{\to} \sigma[\mathit{sync\_cnt}\uparrow],\ \mathcal{S}, \alpha.t \quad \textbf{if } n_0 \neq \sigma(\mathit{sync\_cnt}) \qquad \text{[P-SYNC-NOPRMPT]}$$

Dynamic Check Rules $\boxed{\mathcal{L}, t :\ \omega \overset{\omega}{\to} b}$ $\qquad \mathit{type}(\alpha)$: return the type of a log entry α.

$$\alpha\cdot\mathcal{L},\ t :\ \mathit{chkEvnt}(\ell) \overset{\omega}{\to} \alpha.t = t \wedge \alpha.\ell = \ell$$

$$\alpha\cdot\mathcal{L},\ t :\ \mathit{chkWrt}(\ell, c) \overset{\omega}{\to} \mathit{type}(\alpha) = \mathit{WRITE} \wedge \alpha.t = t \wedge \alpha.c = c \wedge \alpha.\ell = \ell$$

$$\alpha\cdot\mathcal{L},\ t :\ \mathit{chkAssrt}(\ell, c) \overset{\omega}{\to} (\mathit{type}(\alpha) \neq \mathit{FAIL} \wedge c \neq 0) \vee (\mathit{type}(\alpha) = \mathit{FAIL} \wedge \alpha.t = t \wedge \alpha.\ell = \ell \wedge c = 0)$$

Expression Rules $\boxed{\sigma, \mathcal{L} :\ e \overset{e}{\to} \mathcal{L}', e'}$ $\qquad \sigma,\ \mathit{READ}\langle i, t, d, \ell, c\rangle\cdot\mathcal{L} :\ \textbf{read}^\ell(d) \overset{e}{\to} \mathcal{L}, c$

Local Statement Rules $\boxed{\sigma, \mathcal{K}, \mathcal{L} :\ s \overset{s}{\to} \sigma', \mathcal{K}', \mathcal{L}', s'}$

$$\sigma,\ \mathcal{K},\ \mathit{WRITE}\langle i, t, d, \ell, c\rangle\cdot\mathcal{L} :\ x := \textbf{write}^\ell(d, c) \overset{s}{\to} \sigma,\ \mathcal{K},\ \mathcal{L},\ \textbf{skip}$$

$$\sigma,\ \mathcal{K},\ \mathit{FAIL}\langle i, t, \ell\rangle\cdot\mathcal{L} :\ \textbf{assert}^\ell(0) \overset{s}{\to} \sigma,\ \mathcal{K},\ \mathcal{L},\ \textbf{fail}$$

$$\sigma, \mathcal{K}, \mathcal{L} :\ \textbf{assert}^\ell(c) \overset{s}{\to} \sigma,\ \mathcal{K},\ \mathcal{L},\ \textbf{skip} \quad \textbf{if } c \neq 0$$

Global Rules $\boxed{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_n) \longrightarrow \sigma',\ \mathcal{K}',\ \mathcal{L}',\ \mathcal{S}',\ t', (s_1', \ldots, s_n')}$

$\mathit{deterministic\_next\_thread}(t, \mathcal{L})$: deterministically selects the next thread given the current thread t and the log.

$$\frac{\sigma, \mathcal{K}, \mathcal{L} :\ s \overset{s}{\to} \sigma',\ \mathcal{K}',\ \mathcal{L}',\ s'}{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_{(t-1)},\ E[s]_s,\ s_{(t+1)}, \ldots, s_n) \longrightarrow \sigma',\ \mathcal{K}',\ \mathcal{L}', \mathcal{S},\ t, (s_1, \ldots, s_{(t-1)},\ E[s']_s,\ s_{(t+1)}, \ldots, s_n)}$$

[R-SAME-THRD]

$$\frac{\mathcal{K}(k) \neq \perp \qquad \mathcal{K}(k) \neq t \qquad t' = \mathit{deterministic\_next\_thread}(t, \mathcal{L})}{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_{(t-1)},\ E[\textbf{acquire}^\ell(k)]_s,\ s_{(t+1)}, \ldots, s_n) \longrightarrow \sigma,\ \mathcal{K},\ \mathcal{L}, \mathcal{S},\ t', (s_1, \ldots, s_{(t-1)},\ E[\textbf{acquire}^\ell(k)]_s,\ s_{(t+1)}, \ldots, s_n)}$$

[R-LOCKFAIL]

$$\frac{\sigma, \mathcal{S}, \mathcal{L}, t :\ \pi \overset{\pi}{\to} \sigma', \mathcal{S}', t'}{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_{(t-1)},\ E[\pi]_\pi,\ s_{(t+1)}, \ldots, s_n) \longrightarrow \sigma',\ \mathcal{K},\ \mathcal{L}, \mathcal{S}',\ t', (s_1, \ldots, s_{(t-1)},\ E[]_\pi,\ s_{(t+1)}, \ldots, s_n)}$$

[R-PREEMPT]

$$\frac{\mathcal{L}, t :\ \omega \overset{\omega}{\to} \textbf{true}}{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_{(t-1)},\ E[\omega]_\omega,\ s_{(t+1)}, \ldots, s_n) \longrightarrow \sigma,\ \mathcal{K},\ \mathcal{L}, \mathcal{S},\ t, (s_1, \ldots, s_{(t-1)},\ E[]_\omega,\ s_{(t+1)}, \ldots, s_n)}$$

[R-CHK-PASS]

$$\frac{\mathcal{L}, t :\ \omega \overset{\omega}{\to} \textbf{false}}{\sigma,\ \mathcal{K},\ \mathcal{L},\ \mathcal{S},\ t,\ (s_1, \ldots, s_{(t-1)},\ E[\omega]_\omega,\ s_{(t+1)}, \ldots, s_n) \longrightarrow \sigma,\ \mathcal{K},\ \mathcal{L}, \mathcal{S},\ t, (s_1, \ldots, s_{(t-1)},\ \textbf{invalid\_replay},\ s_{(t+1)}, \ldots, s_n)}$$

[R-CHK-FAIL]

Fig. 8. Replay Semantics. The subscripts in evaluation contexts denote the evaluation kind.

are loaded from the log instead. The rules in Fig. 8 are not complete, showing only those different from the logging semantics. In particular, a read expression reads the value from the first entry in the log. Note that its preceding check ensures progress of the evaluation. Statement rules are mainly removing the first log entry.

Global Rules. These rules model deterministic execution on a single core. In the configuration, we introduce a thread id t to explicitly constrain the thread where the evaluation happens; the resulting thread t' may be different, indicating a context switch. Rule [R-STMT-SAME-THRD] dictates that evaluation remains within the same thread as much as possible, ensured by the same thread id before and after the evaluation. Rule [R-LOCKFAIL] deterministically selects the next thread when it fails to acquire a lock. In our implementation, we select the next available thread following the log order. Rule [R-PREEMPT] switches to the thread t' indicated by the evaluation of a preemption test. It is a no-op if $t = t'$. Rules [R-CHK-FAIL] specifies that the evaluation terminates with **invalid_replay** if a check fails.

Example. Lets revisit the example in Section 2. In the initial replay in Fig. 2 (a), schedule $\mathcal{S} =$ **nil** and the log is shown in Fig. 1 (c). It ends with an **invalid_replay**. In Fig. 2 (b), schedule $\mathcal{S} = sync\langle 2,2\rangle$, representing the preemption at Ⓓ, that is, switching to thread 2 upon the 2nd synchronization.

4 Incremental Schedule Exploration

When replay fails to make progress, our technique starts to explore different sub-schedules within a window close to the inconsistent event. The part of the schedule that happens before the window is considered finalized. The goal of exploration is to advance the replay, that is, to be able to replay at least one more event. The sub-schedule leading to advance is then admitted to the final schedule. The process is incremental and demand-driven. Exploration could be at two levels: the *coarse-grained* level that explores different orders of code blocks protected by synchronizations, and the *fine-grained* level that explores different orders of memory accesses. The algorithm first explores coarse-grained schedules, if it succeeds in advancing the replay, it will skip the fine-grained exploration.

4.1 Exploration Window

An important concept in our technique is the *exploration window*, which defines the scope of sub-schedule exploration. This allows us to avoid logging and reordering memory accesses for the whole execution as in PRES [25]. Intuitively, we consider that an inconsistent event α_x by state differences (compared with the original run) that occur in between the preceding event α_p *in the same thread* and α_x. Note that we consider the validity of the program state of the thread up to α_p is endorsed by the valid replay up to that event. The state between α_p and α_x could be affected by any parallel execution in other threads. Hence, *the exploration window includes the execution durations of all threads that could happen in parallel with the duration from* α_p *to* α_x. We consider two durations could happen in parallel if the happens-before relation between the two cannot be inferred from the event log order. Next, we formally define the window computation.

$$immPrec(\alpha_x^{tt},t) = \alpha^t \;\; s.t. \alpha^t \prec \alpha_x^{tt} \wedge \nexists\, \alpha_0^t \;\; \alpha^t \prec \alpha_0^t \prec \alpha_x^{tt}$$
$$immSucc(\alpha_x^{tt},t) = \alpha^t \;\; s.t. \alpha_x^{tt} \prec \alpha^t \wedge \nexists\, \alpha_0^t \;\; \alpha_x^{tt} \prec \alpha_0^t \prec \alpha^t$$

We first define two auxiliary functions. Function $immPrec(\alpha_x^{tt},t)$ computes the immediate preceding event in thread t regarding the given event α_x^{tt} in thread tt. We use the superscript to describe the thread where an event happens. We use operator $\prec$ to denote precedence in the log order. Similarly, function $immSucc()$ computes the immediate succeeding event. Given the two functions, the exploration window of a thread t regarding a given inconsistent event α_x^{tt} is computed as follows.

$$window(\alpha_x^{tt},t) = \langle immPrec(\alpha_p^{tt},\, t),\; immSucc(\alpha_x^{tt},\, t)\rangle$$
$$\textbf{where } \alpha_p^{tt} = immPrec(\alpha_x^{tt},tt)$$

In particular, α_p^{tt} denotes the immediate preceding event of the inconsistent event in the same thread tt. Hence, the window is delimited by an event in t that immediately precedes α_p^{tt} and an event in t that immediately succeeds α_x^{tt}.

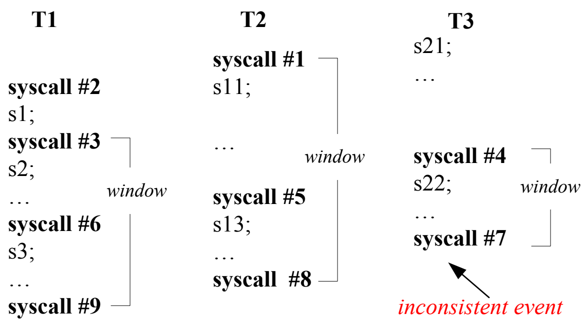


Fig. 9. Example for exploration window.

Example. Consider the example in Fig. 9. The syscall numbers represent their global order. The inconsistent event is *syscall #7* in thread 3, denoted as α_7^3 for short.

$$immPrec(\alpha_7^3,3) = \alpha_4^3$$
$$window(\alpha_7^3,2) = \langle immPrec(\alpha_4^3,2), immSucc(\alpha_7^3,2)\rangle = \langle \alpha_1^2, \alpha_8^2\rangle$$

That is to say the window for thread **T2** is from *syscall #1* to *syscall #8*. Observe that

from the event order, we cannot tell the happens-before of statement *s11* in **T2** and *s22* in **T3**. The window for thread **T1** is similarly computed. Note that although α_2^1 happens after α_1^2, it is not in the window while α_1^2 is. In other words, an exploration window is not a consecutive sequence of global evaluation steps, but rather the aggregation of durations from all threads.

Algorithm 1 One preemption coarse-grained exploration.

```
In: $\alpha_x$: the inconsistent event;
Out: $\alpha'$: the new inconsistent event;
Def: $R$: ordered list of all threads;
    SyncTrace $T ::= \overline{\langle n, \ell, \alpha \rangle}$ with $n$ the sync. counter, $\ell$ the program point of the sync., and $\alpha$ the
      first logged event that happens after the sync.;
    syncTraceInWindow($\alpha_x$): replay the execution and produce the sync trace within the explo-
      ration window of $\alpha_x$;
    replay ($\mathcal{L}$,$\mathcal{S}$): replay and return the inconsistent event;
    sortThreadByLog ($\mathcal{L}$): sort all threads by the first event in $\mathcal{L}$
```

CoarsegrainedExplore (α_x)

```
 1:  $T \leftarrow$ syncTraceInWindow ($\alpha_x$)
 2:  foreach $\langle n, \ell, \alpha \rangle \in T$ in the backward order do
 3:      let $\mathcal{L}_p \cdot \alpha \cdot \mathcal{L}_s = \mathcal{L}$
 4:      $R \leftarrow$ sortThreadByLog($\mathcal{L}_s$)
 5:      foreach $t \in R$ in the ascendent order do:
 6:          $\mathcal{S}' \leftarrow sync\langle n,\ t\rangle$
 7:          $\alpha' \leftarrow replay(\mathcal{L}, \mathcal{S} \cdot \mathcal{S}')$;
 8:          if $\alpha_x \prec \alpha'$ then
 9:              $\mathcal{S} \leftarrow \mathcal{S} \cdot \mathcal{S}'$
10:              return $\alpha'$
11:  return $\alpha_x$
```

4.2 Coarse-grained Exploration

The coarse-grained exploration aims to reorder the synchronized blocks within the window. Given a bound m, it tries to perform up to m preemptions. At each preemption, the algorithm tries to switch to a thread selected based on the order of the threads' first events in the remaining log. The intuition is that a thread that appears later is less likely to be part of the target interleaving. The exploration is backward: priority is given to preemptions close to the end of the window. The intuition is that perturbing schedules close to the inconsistent event is more likely to affect the event. Our implemetation supports multiple preemptions, but we observe that $m = 1$ is sufficient in this work.

Algorithm 1 presents the backward search algorithm of one preemption. It takes the inconsistent event as input and returns a new inconsistent event. It first collects the synchronization trace within the window. The main loop in lines 2-10 enumerates each synchronization in a backward fashion, and preempts at that point. For each preemption point, lines 3 and 4 sort the threads based on their first events that happen after the synchronization precluding the first such event α, as the default replay order has already followed the schedule inidicated by α. The loop in lines 5-10 tries the different target threads of the preemption based on the sorted order. In lines 8-9, if the new schedule leads to progress, it is appended to the final schedule to allow future replay.

Example. Consider the example in Fig. 9. Assume statement *s22* in **T3** is a selected synchronization for preemption. Lines 3-4 sorts the threads to $R_1 = \{t_1, t_2\}$, suggested by *syscall#6* and #8. Hence, the algorithm first preempts to t_1. Note that the original replay switches to **T2** at *s22* by default. □

For $m > 1$, we cannot simply enumerate an m subsequence of the synchronization trace as a preemption may change the control flow such that the following synchronization sequence is different. Hence, the implemented algorithm is a recursive version of Algorithm 1. Essentially, it first tries the different options of the first preemption and tentatively admits it to the final schedule and then recursively calls itself to look for the second preemption, and so on. Details are elided as $m = 1$ is sufficient in our experience.

4.3 Fine-grained Exploration

If the coarse-grained exploration fails to make progress, the algorithm resorts to reordering shared variable accesses within the window. The idea is to first detect data races within the window. Then the algorithm selects a subset of races and reverses the order of the accesses in each race[1]. The size of the subset is limited by the preemption bound m. Reversing the order of a racy access pair is achieved by disabling the thread right before the first access, and then enabling it right after the second access (in a different thread). We call the set of races to be reversed the *memory schedule*. The search of memory schedule is also backward, giving priority to accesses close to the end of the window.

Our technique can continue to explore the preceding window if we cannot find a valid solution in the current window, although we haven't experienced such cases.

5 Caching Replay Failures

According to our study in Section 2, the same replay failure tends to happen repetitively. To avoid redundant schedule exploration, we develop a caching mechanism. We have two caches, corresponding to the two possible replay failures, *unmatched events* and *deadlocks*, respectively.

An unmatched event replay failure means that we expect to see an event α_c during replay but the next event α_x in the log is different. *Ideally*, the unmatched event cache $\mathcal{C}_{event}$ should have the following signature.

$$\mathcal{C}_{event} : LogEntry \times LogEntry \rightarrow ThreadId \times InstCnt \times ThreadId$$

$\mathcal{C}_{event}(\alpha_c, \alpha_x) = \langle t_0, n, t_1 \rangle$ means that upon a replay failure denoted by $\langle \alpha_c, \alpha_x \rangle$, the preemption should be performed in thread t_0 at the nth synchronization within the window when counting backward[2]. Note that we cannot use the global count as it is unique for each synchronization instance. The execution should switch to thread t_1. However in practice, it is not desirable to hard-code the thread id in the cache because it is very common that the different occurrences of the same replay failure may involve different sets of threads. For example, worker threads tend to execute the same piece of code, such as in `Apache` and `MySQL`. It is very likely that a replay failure such as (A) in Fig. 3 happens between worker threads T1 and T2 this time but T2 and T4 next time.

[1] Here, a race is defined as a pair of accesses in the window on the same shared variable from different threads, with at least one being a write.

[2] Our discussion is limited to coarse-grained schedule for brevity.

Hence, we use the label of the next statement to execute in a thread to denote the thread. Therefore, in our design, $\mathcal{C}_{event}(\alpha_c, \alpha_x) = \langle \ell_0, n, \ell_1 \rangle$ means that the preemption should occur in a thread that is about to execute statement ℓ_0 and the target thread is a thread that is about to execute ℓ_1.

For example, after the coarse-grained search succeeds in the example in Fig. 2 (a)-(b). A cache entry $\mathcal{C}_{event}(\text{syscall\#3}, \text{syscall\#2}) = \langle 10, 1, 3 \rangle$ is added. Number 10 means that we should preempt a thread that is about to execute statement 10, which is thread one; 1 means the preemption point is the last synchronization in the window, i.e. ©; 3 means that the target thread is about to execute statement 3, i.e. thread two.

Note that our discussion limits to one preemption, extending the cache design to support multiple preemptions is omitted.

A deadlock may involve multiple threads. A complex design is needed if we use all the involved threads as the cache key. We have developed a much simpler design that is very effective in practice. We use the label of the replay failure statement of the thread of the next event in the log as the hash key. For example, in Fig. 3 (C), *syscall#2* is the next event to replay and its thread fails to make progress at the conditional wait. We use the label of the wait as the key.

6 Implementation

In the following, we highlight some of our engineering efforts.

System Call Recording. Minimizing logging overhead is one of our design goals. Therefore, we only log a subset of system calls which are necessary for replay. We currently intercept 84 out of 326 Linux system calls. Most of them are related to input, such as file and socket inputs, `select` and `gettimeofday`. We do not intercept/record output system calls.

Minimal Binary Rewriting. One of the important features of our technique is that it is hardware/software infrastructure-free. It works directly on binaries. Therefore, our technique intercepts syscalls by binary rewriting. In particular, it intercepts the dynamic linker interface. When an executable or a library is loaded, it scans the binary image and replaces all the syscall instructions with calls to our functions that realize the logging/replay functionalities. The binary rewriting component is adapted from that in Jockey [27], a logging/replay tool that does not support concurrent execution. Note that the rewriting is dynamic and very simple. It directly overwrites a small number of instructions in the code segment.

Intercepting Shared Variable Accesses. Without compiler support, it requires non-trivial efforts to intercept shared variable accesses, which is only needed during replay. It is not optimal to use dynamic instrumentation infrastructures such as Pin or Valgrind, due to their high cost. Our solution is to use memory protection to intercept memory accesses. In fine-grained exploration, we start protecting all global and heap pages *at the beginning of the window*. Upon a page fault (i.e. an access), we unprotect the page, and set the trap register to trap the next instruction. The execution is thus trapped after the access. We then log the access and re-protect the page. Observe that tracing is a one-time cost and it happens only within a window. Once the trace is acquired, the algorithm iteratively replays, reversing a set of races each time. In these replayed executions, only the pages specified by the memory schedule are protected, causing very few page faults. The majority of a replayed execution has no overhead.

Other Challenges. In the formal semantics, we use labels of syscall statements, which can be considered as the program counters (PCs). However, syscalls are mostly within libraries. We use stack-walk to identify the corresponding invocation in the user space and use its PC as the label. Threads in real-world programs tend to use *pipe* and *epoll* syscalls to communicate. They may send pointers through these syscalls. We cannot simply log the content of these syscalls and restore it during replay. We choose not to restore from the log but rather re-execute the relevant syscalls.

7 Evaluation

Applications	LOC	Threads	Bug description
Apache-2.0.48	157K	7	#1: Unprotected buffer #2: Atomicity violation (21287)
Apache-2.2.6	198K	7	#3: Atomicity violation (45605)
MySQL-5.0.11	934K	14	#1: Atomicity violation (47761) #2: Atomicity violation (12845)
Cherokee-0.9.4	43K	4	Atomicity violation (326)
Transmission-1.4.2	59K	2	Null pointer access (1818)
PBZip2-0.9.4	1.5K	6	Lock destroyed before it is accesses
Gftp-2.0.19	38K	5	Crash (546035)

Table 2. Application and Bug description.

	Original time(s)	Recording overhead (%)	PRES-like		Two-layer Exploration				Two-layer Exploration with caching					Replay time(s)
			FG Rep.	Time (sec)	CG Rep.	Log Mem	FG Rep.	Time (sec)	Cache Hit	CG Rep.	Log Mem	FG Rep.	Time (sec)	
Apache #1	12.43	3.22	28	301.21	24	1	1	40.21	10	14	1	1	29.85	1.64
Apache #2	7.14	3.78	22	615.42	32	1	1	281.42	8	10	1	1	92.59	6.12
Apache #3	10.89	2.94	16	196.73	27	1	1	43.85	9	14	1	1	31.23	1.28
MySQL #1	5.21	3.84	62	1342.6	46	1	2	137.1	17	24	1	2	81.47	2.47
MySQL #2	4.27	3.51	59	1429.1	39	1	1	151.82	15	22	1	1	92.5	3.71
Cherokee	120.42	2.11	15	684.29	12	1	3	61.51	2	7	1	3	42.11	4.15
Transmission	1.58	0.63	2	4.61	3	-	-	1.32	0	3	-	-	1.32	0.43
PBZip2	9.87	1.11	8	1615.78	6	-	-	201.42	0	6	-	-	201.42	35.42
Gftp	131.12	2.61	2	115	2	-	-	24.68	0	2	-	-	24.68	13.41

Table 3. Recording and replay performance. CG denotes coarse-grained schedule exploration. FG denotes fine-grained schedule exploration.

In this section, we evaluate the performance and the practicality of our technique. All experiments in this section were conducted on a quad-core Intel Xeon 2.40GHz with 4GB of RAM running Linux-2.6.35.

In the first experiment, we evaluate the performance of our technique over a set of real world bugs from 6 applications. Table 2 presents the programs and bugs. `Cherokee` is a web server. `Transmission` is a BitTorrent client. `PBZip2` is a parallel implementation of the bzip2 file compressor. `Gftp` is a multithreaded file transfer client. We use test inputs provided with the programs if available or randomly generated inputs otherwise and we weave these inputs with the failure inducing inputs to trigger the bugs. For the UI program `Gftp`, the failures are induced by a sequence of user actions.

In the experiment, we also compare our technique with PRES. Note that it is difficult to compare the logging overhead of PRES with our technique as PRES was implemented on PIN and it logs a lot more events than our technique. The comparison hence focuses on the replay/schedule-reconstruction cost. We implemented PRES's replay algorithm according to the published paper [25]. We call it the PRES-like algorithm. Since PRES has multiple strategies, leveraging various kinds of information with some of them expensive to collect, we only adapted one of the strategies such that it operates on our log, which mainly consists of system calls, signals and thread spawns. Upon replay failures, the PRES-like algorithm identifies and logs all shared memory accesses, and then tries to reverse the order of the racy pairs.

Table 3 presents the results. Column 2 presents the original execution time without our logging tool for each application and column 3 shows recording overhead.

Columns 4-5 show the number of schedule exploration attempts and the accumulated time for the PRES-like approach. Columns 6-9 show the cost of two-layer exploration without caching. Column 6 presents the number of tries for coarse-grained schedules, column 7 shows the number of times of collecting shared-memory access trace within the window. Column 8 shows the number of tries for fine-grained schedules and column 9 presents the accumulated time. Columns 10-14 show the cost of two-layer exploration with caching. Column 10 shows the number of cache hits.

From the data, we make the following observations : (1) The logging overhead is very low, with the maximum 3.84% and average 2.6%. (2) Replay does not fail often compared to the total number of events in the log (see Column 2 in Table 4), showing the effectiveness of the default replay strategy. (3) Fine-grained exploration is rarely needed. (4) The caching mechanism is very effective in avoiding redundant exploration. (5) Our technique is more efficient than the PRES-like algorithm. Our schedule exploration with caching is an order of magnitude faster in most cases. This is mainly because we have two layers of exploration, limit schedule exploration within a window, and use caching.

Once we find a schedule to trigger a bug, we can replay the bug as many times as we want. The last column shows the replay time when we have the correct schedule. It is significantly less than the original run except for `Apache#2` and `PBZip2`. This results from the time saved by emulating all syscalls during replay - no waiting time is incurred when replaying syscalls.

Table 4 presents the statistics about windows. Column 3 shows the average number of coarse-grained schedules which we can explore within an event's window. Column 4 shows the average number of data race pairs. For `PBZip2`, `Transmission` and `Gftp`, the correct schedules can be found with coarse-grained exploration and hence we do not need to detect data-races.

Practicality Study With Real Workload. In order to evaluate the practicality of our technique, we acquired the high level web request log for our institution's web-site for

Bugs	# of logs	Window size	
		Coarse-grained	Fine-grained
Apache #1	32,578	50.32	13.03
Apache #2	128,589	40.32	3.46
Apache #3	33,261	48.27	10.15
MySQL #1	95,974	20.24	2.01
MySQL #2	87,425	25.19	2.42
PBZip2	3,426	31.77	-
Transmission	36	0.25	-
Cherokee	36,841	30.17	0.15
Gftp	22,332	28.15	-

Table 4. Average window size

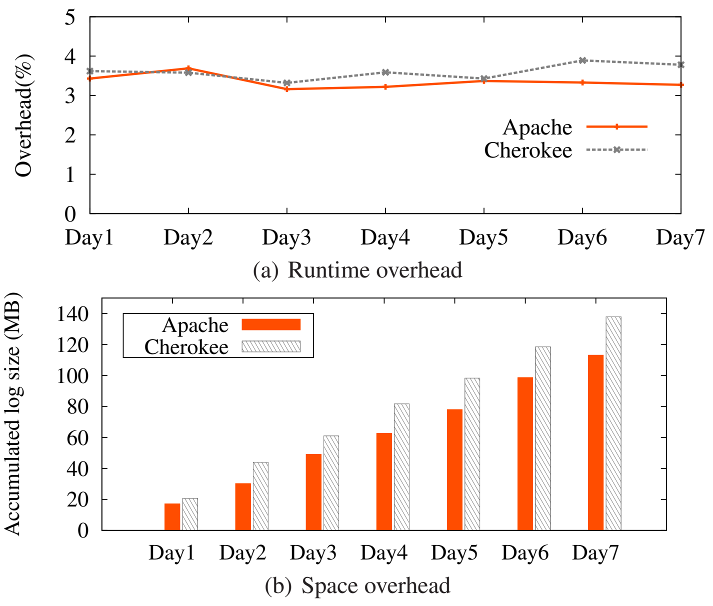


Fig. 10. Runtime and space overhead with real-world workload.

Apps.	Exploration w/o caching			Exploration with caching			
	CG	FG	Time(s)	Hit	CG	FG	Time(s)
Apache	181	3(2)	13184.72	61	72	3(2)	5270.41
Cherokee	89	1(1)	6269.34	39	50	1(1)	3612.3

Table 5. Performance with one day log. Number in braces indicates the number of times a memory access trace is collected.

one week. We wrote a script to regenerate the workloads for 1-7 days and fed them to the Apache and Cherokee server programs. At the end of each workload, we supplied the failure inducing requests to trigger the failure. The average logging overhead and aggregated space overhead are presented in Fig. 10. Observe that the logging overhead is more or less consistent and the space overhead is reasonable for a few day's execution. These results show the practicality of our logging technique.

Table 5 presents the replay cost with the real-world workload. Here we replay the last day's log only. The schedule exploration cost is high (2-4 hours without caching, 1-2 hours with caching), because we have to pay the cost of re-executing from the beginning for each exploration. We expect checkpointing would help a lot in this case, but we will leave it for our future work. The number of schedule explorations is not that high compared to the long duration of the workload. Caching substantially improves the performance. Note that a cache hit might save multiple exploration tries.

Synchronization Order Recording. If we have the global order of synchronization operations in the log, we can narrow down the search space. However the logging overhead increases. We measured the recording overhead including the global order of synchronization functions. It shows that the logging overhead becomes 7.6% for Apache and is increased by a factor of 2-3 for other benchmarks.

8 Related Work

The prior work most relevant to ours is PRES [25], which first tries to replay with syscall, synchronization, or even basic block log. If none of these succeeds, it tries to reverse shared memory access order. In comparison, PRES logs more information and it relies on PIN [18], entailing higher logging overhead. Moreover, we introduce exploration window, two-layer exploration, and caching, which are critical for reducing search space and improving replay performance. Our results show that our schedule exploration/reconstruction algorithm is 10.55 times faster on average. We have also formalized the technique and revealed in-depth observations about replaying real concurrent execution in general.

CLAP [12] presents a search-based deterministic replay system, which uses SMT solver and thread-local profiling to achieve replay determinism and to reduce the recording overhead. Compared to our technique, CLAP records more information such as control-flow paths, causing higher recording overhead (up to 296%).

There are also software based replay systems that record individual memory accesses and their happens-before relations [5, 8]. Such systems entail substantial runtime overhead. In [2], a constraint solver is used to reproduce concurrent failures from incomplete log. There has been substantial work on software-based recording and replay for applications such as parallel and distributed system debugging [23, 27, 26, 9, 3, 15, 22, 13]. These systems only perform coarse-grained logging at the level of system calls or control flow and hence are not sufficient for reproducing concurrency failures. We consider these techniques complementary to ours.

Recently, it has been shown that with architectural support, concurrent execution can be faithfully replayed [10, 19, 21, 31, 29]. While such techniques are highly effective, they demand deployment of special hardware, which limits their applicability.

Lee *et al.* [16] propose an execution reduction technique that aims to faithfully replay a failure with a reduced log. A key technique of their work is the unit-based loop analysis that reduces unnecessary iterations from the replay log. We consider this technique complementary to ours.

In recent years, significant progress has been made in testing concurrent programs. CHESS [20] is a stateless bounded model checker that performs systematic stress testing to expose bugs in concurrent programs. It can be adopted to reproduce Heisenbugs. CTrigger [24], PENELOPE [30] and PACER [6] are other concurrency testing techniques that search for schedule perturbations to break usual patterns of shared variable accesses to expose faults. Random schedule perturbations are also shown to be effective in debugging races and deadlocks [28, 14]. These techniques do not log the original runs. They usually assume the (simplified) failure inducing inputs are provided.

DoublePlay [32] proposes a time-slicing technique of execution that runs multiple time intervals of a program on spare cores. Dthreads [17], PEREGRINE [7] and Coredet [4] propose deterministic execution system for multi-threaded applications.

9 Conclusion

We have developed a logging and replay technique for real concurrent execution. The technique is self-contained, does not require any infrastructure support. It features very low logging overhead as it does not log any synchronization operations or shared memory accesses. Replay is an incremental and demand-driven process. The technique always tries to replay by the log, but it may fail due to schedule differences. Upon a replay failure, an exploration process is triggered to search within a window for a schedule that allows progress. We have developed two kinds of explorations: one is at the synchronized block level and the other is at the shared memory access level. A sophisticated caching mechanism is developed to leverage the reoccurrences of replay failures. Our results show that the technique is effective and practical, and substantially improves the state of the art.

10 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research is supported in part by the National Science Foundation (NSF) under grants 0917007 and 0845870. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

1. Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '90.
2. Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles*, SOSP '09.

3. Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05.
4. Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '10.
5. S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the second ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '06.
6. Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10.
7. Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11.
8. George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08.
9. Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI '08.
10. D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture*, ISCA '08
11. Jeff Huang, Peng Liu, and Charles 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 '10.
12. Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13.
13. Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: taming nondeterminism in distributed systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13.
14. Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09.
15. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05.
16. Kyu Hyung Lee, Yunhui Zheng, Nick Sumner, and Xiangyu Zhang. Toward generating reducible replay logs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11.
17. Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11.
18. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumen-

tation. In *Proceedings of the 26rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05.
19. Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09.
20. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multi-threaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07.
21. Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06.
22. Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '94.
23. Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88.
24. S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09.
25. Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles*, SOSP '09.
26. Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communcation of the ACM*, 2003.
27. Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the Automated and Algorithmic Debugging* ,AADEBUG '05.
28. Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08.
29. Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11.
30. Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10.
31. Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, ATEC'04.
32. Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11.
33. Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: detecting concurrency bugs through sequential errors. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11.