Reproducing Concurrency Bugs Using Local Clocks

Xiang Yuan\textsuperscript{1}, Chenggang Wu\textsuperscript{1},*, Zhenjiang Wang\textsuperscript{1}, Jianjun Li\textsuperscript{1}, Pen-Chung Yew\textsuperscript{3},
Jeff Huang\textsuperscript{4}, Xiaobing Feng\textsuperscript{1}, Yanyan Lan\textsuperscript{2}, Yunji Chen\textsuperscript{1} and Yong Guan\textsuperscript{5}

\textsuperscript{1}State Key Laboratory of Computer Architecture / \textsuperscript{2}CAS Key Laboratory of Network Data Science and Technology
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
\{yuanxiang,wucg,wangzhenjiang,lijianjun,fxb,lanyanyan,cyj\} @ict.ac.cn
\textsuperscript{3}Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, USA. yew@cs.umn.edu
\textsuperscript{4}Department of Computer Science and Engineering, Texas A&M University, Texas, USA. jeff@cse.tamu.edu
\textsuperscript{5}College of Information Engineering, Capital Normal University, Beijing, China. guanyong@mail.cnu.edu.cn

\textbf{Abstract}—Multi-threaded programs play an increasingly important role in current multi-core environments. Exposing concurrency bugs and debugging such multi-threaded programs have become quite challenging due to their inherent non-determinism. In order to eliminate such non-determinism, many approaches such as record-and-replay and other similar bug reproducing systems have been proposed. However, those approaches often suffer significant performance degradation because they require a large amount of recorded information and/or long analysis and replay time. In this paper, we propose an effective approach that takes advantage of the hardware clocks available on modern commercial processors. The key idea is to reduce the overhead in recording and analyzing events’ global order by using time stamps recorded in each thread. Those time stamps are used to determine the order of shared accesses among threads. To avoid the large overhead incurred in accessing system-wide global clock, we opt to use local per-core clocks that incur much less access overhead. We then propose techniques to resolve differences among local clocks and to get accurate global event order. By using per-core clocks, state-of-the-art bug reproducing systems such as PRES and CLAP can reduce the recording overheads by 1% \textasciitilde 85%, and the analysis time by 84.66\% \textasciitilde 99.99\%, respectively.

\textbf{Index Terms}—concurrency, bug reproducing, local clock

\section{I. INTRODUCTION}

Parallel programming has become essential to fully utilize the compute power of multi-core processors. However, debugging has been a daunting task for developers of multi-threaded programs because of their non-deterministic nature [30]. A survey shows that it took about 73 days to fix a multi-threaded bug [1]. These bugs can have serious consequences. Well-known incidents include the Therac-25 medical accident [2] and the 2003 North American blackout [3]. Such bugs need to be detected and fixed as quickly as possible.

A classical solution is the technique of Record & Replay (RR). It faithfully records the execution interleaving and deterministically enforces the same interleaving to reproduce bug, such as [28][18][16][22][23]. The chief challenge faced by RR systems is the performance penalty that they incur at runtime for recording the schedule information. Some RR systems [10][11] could introduce 10X\textasciitilde100X runtime slowdown. Besides, the observer effect of RR systems may alter the time sequence of real-world runs, which can foreclose the discovery of many bugs especially those on weak memory models [4].

To address the limitations of RR, researchers have proposed a new technique to reproduce bugs, which records only some key information rather than all schedules, and instead reproduces the buggy interleaving with offline analysis and guided exploration. Because much less information is recorded at runtime, the runtime overhead of this technique is much lower than RR. Many systems adopt this idea [17][19][21][14][4]. Although the schedule reproduced by these systems may not be exactly the same as the original one, they are useful in practice as the same failure always can be reproduced.

For example, PRES [14] records the global orders of some special events by instrumentation. Those events may include synchronizations, system calls, function calls, basic blocks, and memory instructions. When a bug turns up, it tries to analyze the unordered shared accesses. At the function-call level, PRES can reproduce bugs in 10 tries for most benchmarks and experience only 10\%\textasciitilde779\% slowdown [14].

Similar to RR systems, PRES needs to explicitly record the global order of shared-resource accesses among threads. They use synchronization operations to serialize the logging to a shared buffer or incrementing a global event counter, which are the root cause of execution slowdown [4].

To avoid such expensive synchronization operations, an effective mechanism called CLAP [4] has been proposed. Each thread in CLAP only records its own execution paths and some local information. It does not need to access the global buffer or the global event counter. During the offline analysis, CLAP generates constraints by symbolic execution and searches for buggy interleavings by a Satisfiability Modulo Theories (SMT) solver, such as Yices [29] and Z3 [24]. Thus its slowdown is only about 9\%\textasciitilde294\%. However, it cannot get the buggy interleavings using those recorded data directly, instead, it relies on an SMT solver, which is hard to scale because the constraint solving program is NP-hard.

These systems trade off less time in the record phase with more time in the analysis and replay phase. Hence, it is very desirable to find a scheme that could allow each thread to log only its own local information while use a quick offline analysis on these logs to get the order of shared

\textsuperscript{*}To whom correspondence should be addressed

\section{II. RELATED WORKS}


discussion of related work...
accesses among threads during the replay. Such a scheme could greatly improve the efficiency of program debugging for multi-threaded programs.

The key insight of this work is to explore the effectiveness of using the hardware local clock to reproduce concurrency bugs, such that to reduce both the recording overhead and the bug reproduction time. Most commercial processors today, such as Intel/AMD x86, IBM Power, MIPS, and Sun SPARC, provide such clocks. Each thread can record values of these local clocks without any need for synchronization with other threads. The order of shared accesses can then be inferred accordingly. Unfortunately, these local clocks are core-private, and the hardware is not designed to guarantee their consistency, i.e., these local clocks may have different skews among themselves. Also, it is quite difficult to get the precise skews among these local clocks (unless there is a global clock as assumed in [15]). The main challenge here is thus to make use of these local clocks to determine a global order among memory accesses. As a result, there has not been any bug reproducing system that uses such local clocks.

This paper proposes a new mechanism to reconstruct the order of shared-access among threads in prior runs. We take advantage of the per-core clocks provided on modern processors to overcome the aforementioned challenges. This mechanism is applied to two recent bug reproducing systems and has significantly improved their efficiencies.

Our contributions are as follows:

(1) We use per-core local clocks on commercial processors to determine the global order of shared accesses among different threads that allows concurrency bugs to be recorded and reproduced with substantially reduced overheads.

(2) We propose a methodology to get a range of the skews on per-core local clocks.

(3) We propose a statistical scheme that narrows down the range to less than 10 ticks (10 cycles) with high confidence.

(4) The proposed local-clock approach is applied to two recent systems and improves their efficiency significantly.

Following, Section II gives our motivation. Section III proposes two schemes to calculate the skews among local clocks. Section IV applies our approach to PRES and CLAP. Section V discusses our experimental results. Section VI covers the related work, and Section VII concludes this paper.

II. Motivation

Many mainstream commercial processors provide local per-core clocks, and applications can use them to get their needed timing information. For example, Intel/AMD x86 processors provide a 64-bit Time Stamp Counter (TSC) since Pentium family. The Time Stamp Counter is incremented at a constant rate with respect to the wall-clock time. It is not affected by the frequency of processors [5], so as to avoid the impact of dynamical frequency scaling. Similar mechanisms exist on other processors. On IBM Power processors, every core has a 64-bit Time Base register [7]. Comparing to x86’s TSC register, its counting frequency can be changed by software. In such a case, if we record the change of counting frequency and the frequencies before and after the change, we can convert the value of Time Base register to the real wall-clock time [7]. MIPS processors also have a similar register, called Count Register [6], but its size is only 32-bit. SPARC processors have a 63-bit Tick register [8] to count the clock cycles.

"Ideal" local clocks should have the same value at the same time across all different cores (like a global clock). Each thread can then locally record their own clock values when accessing shared resources. The recorded time values in different threads can then be compared directly to determine their global order. It will need neither synchronization when recording, nor constraints solving when reproducing, so the overall efficiency is improved.

An example is shown in Fig.1. T1 and T2 are two threads bound to different cores. RdTC is the instruction that reads the per-core clock. Suppose in an execution the time stamps read from the local clocks are TS1 and TS2, respectively, and TS2 is smaller than TS1. It means that S6 happened before S2 (i.e., S6 ≺ S2), we can infer that S5 ≺ S3.

However, although hardware local clocks have existed for a long while, no systems have used them to reproduce bugs. An important reason might be that these clocks are not "ideal". Hardware does not ensure that the values read from local clocks on different cores are identical at the same time.

LReplay [15] expects that future processors will provide a global clock with a fast access time, which would dramatically reduce the runtime overhead and log size, as it only needs to record orders that cannot be inferred from the global clock. Unfortunately, most commercial processors provide only local core-private clocks that can be accessed in user mode. It usually requires system calls to access the global clock with a substantially higher overhead. For example, on Intel Xeon Phi, the overhead to access its global clock is in the order of ∼1600 cycles, while it only takes 6-10 cycles to access local per-core clock. However, there are some significant challenges that need to be resolved in order to use the low-overhead local per-core clocks to determine the global order of shared accesses:

(1) We need to deal with the differences among different local per-core clocks. In Fig.1, such differences are needed to infer whether TS2 is earlier than TS1. Unfortunately, it is very difficult to get the differences among different per-core clocks. Therefore, to accurately measure these time differences...
and use them to order shared accesses are the first challenge.

(2) We need to determine the precise clock value when each thread accesses shared resources. Clocks are read by specific instructions, e.g., rdsc on x86. They can be put before or after an instruction accessing a shared resource. However, in neither case does the clock value stand for precisely when the shared resource is actually accessed. Besides, there is no data dependency between RdTC and the shared resource access instruction. Hence, they can be scheduled dynamically in any order on processors that support out-of-order execution. This means in Fig.1, S6 may happen before S5, and S3 may happen before S2. For this reason, we cannot naively use the results of RdTC instructions to order shared accesses directly.

(3) We need to handle possible overflow of the clocks. Clocks on MIPS processors are only 32-bit long, so overflows can occur every few seconds. Even a 64-bit clock can still overflow depending on when we start taking the clock values.

For cores in the same processor, their local clocks are triggered by the same clock signal. They count at the same frequency, and the difference among them will be the same after processor reset. To different processors, if they are the same type and use the same crystal oscillator, the difference of their local clocks are highly possible to be consistent. In such cases, with the difference among these local clocks, we can use their values to determine the orders of shared memory accesses. The following of this paper is based on such cases.

III. Determining the Order by Local Clocks

In this section, we propose our solutions for the challenges mentioned in Section II. To use the local clocks, challenge (2) must be solved first, then we begin from it in this section.

A. Out-Of-Order Execution Exclusion

Most high-performance processors execute instructions out of order for higher performance. Although instructions are retired in order, RdTC reads per-core clock before its retirement, and thus could be out of desired order. An intuitive solution is to insert FENCE instructions before and after each RdTC, which is shown in Fig. 2(a). This may seem to work, but on multi-core platforms things are much more complicated.

In modern multi-cores processors, the completion of a write operation can be divided into two phases: (1) Local Complete (LC), i.e. the written data is held in the local write buffer, but not still seen by other cores yet. (2) Globally Visible (GV), i.e. the written data reaches the cache memory and is visible to all other cores, guaranteed by the cache coherence protocol.

Figure 2(b) shows the example of a wrong inference. A local FENCE only guarantees that W1(LC) ≺ RdTC1, but cannot control W1(GV). From the value of local clock, we would infer that W1 ≺ R2, which is not the case.

Therefore, the selected FENCE instruction must be able to ensure that RdTC is not issued until all previous write instructions become GV. Fortunately, modern processors do provide instructions to ensure such orders among memory instructions, or to flush the pipeline. For example, on x86, MFENCE will hold the following loads and stores until preceding loads and stores become globally visible; LFENCE holds the following instructions until preceding instructions are locally complete. A correct implementation on x86 is thus shown in Fig. 2(c).

In thread T1, MFENCE and LFENCE guarantee W1(GV) ≺ RdTC1, while LFENCE guarantees RdTC2 ≺ R1.

B. Handling the Time Difference among Per-Core Clocks

Although per-core clock values could be different at any instance of time, we can still make use of them if we know their differences (called d). We use the example shown in Fig. 3 here. Assume that the values of the two local clocks are TS_Core1 and TS_Core2 at a certain time, respectively. Then, d is TS_Core2 - TS_Core1. TS2 < TS1 + d means S6 ≺ S2 (i.e. RdTC2 ≺ RdTC1). We can infer that S5 ≺ S3.

However, it is very difficult to get the precise value of d because of those mentioned in Section II. Fortunately, it turns out that if we can get a range of possible values on d, we can still determine the order of shared accesses among threads.

Taking Fig. 3 as an example, assume d ∈ [d1, d2]. If TS2 - d1 < TS1, we have TS2 - d < TS2 - d1 < TS1, and this means S6 ≺ S2. We can thus infer S5 ≺ S3. Similarly if TS1 + d2 < TS2, we can infer S1 ≺ S7. In other cases, these operations cannot be ordered. Although the range of d is not as good as a precise d, it is still possible to determine the order of most shared accesses if the range is small enough.

For commercial processors that cannot provide the value of d precisely, we propose two schemes to get a range of d:

(1) A program testing scheme to obtain a range of d:
(2) A statistical scheme to obtain a smaller range of d with high confidence

1) Scheme 1-Program Testing: We designed a small testing program shown in Fig. 4 for this scheme. The order of RdTC and other instructions is guaranteed, and the fence instructions are not included for clarity. Threads T1 and T2 are bound to two cores for which d is measured. Each thread writes a different value to the variable X. Both threads read the local clock before and after the write operation, and they get TS1, TS2, TS3 and TS4, respectively. The final value of X is checked after both T1 and T2 exits.

If X is 2, the assignment of X in T2 must be later than that in T1, so we can infer S1 < S2 < S7 < S8. At the time that S1 reads the value TS1 from core1’s local clock, the value of core’s local clock is TS1 + d. Therefore, we have TS1 + d < TS4, that is:

\[ d < TS4 - TS1 \quad (if \ \text{Read X is 2}) \quad (1) \]

Similarly, if the value of X read by thread T0 is 1. We can infer that S6 < S7 < S2 < S3, and TS3 < TS2+d:

\[ d > TS3 - TS2 \quad (if \ \text{Read X is 1}) \quad (2) \]

We can repeat the above process so as to collect many cases satisfying either Equation (1) or (2), and obtain many pairs of \( < TS4_i,TS1_i > \) or \( < TS3_i,TS2_i > \). According the above inference, the value of d is less than any \( TS4_i - TS1_i \), and greater than any \( TS3_i - TS2_i \). That is:

\[ \max_i (TS4_i - TS2_i) < d < \min_i (TS4_i - TS1_i) \quad (3) \]

As mentioned in Section III-A, in order to ensure the execution order of the above instructions, we have to add some FENCE or similar instructions in the testing program. We designed four implementations for x86 platforms.

In Fig. 5(a), we use the instructions sequence introduced in Fig. 2(c), while in Fig. 5(b), we use the serializing instruction CPUID instead. Serializing instructions force the processor to complete all modifications to registers and memory by previous instructions and flush all buffered writes to memory before next instruction is fetched [5]. In Fig. 5(c), we make use of atomic instruction XCHG. This implementation does not guarantee that the GV of writing X happen before RDTSRC, so we need to check whether it does. The implementation of Fig. 5(d) is similar to that in Fig. 5(c), but it uses CMPXCHG instead. Fig. 5(c) and Fig. 5(d) may generate a smaller range for d because an expensive MFENCE or CPUID is replaced. The efficacy of the four versions depends on the hardware implementation of different instructions, but we can always run all of them many times to obtain a minimum range of d.

2) Scheme 2-Statistics Testing: Although the range obtained by Scheme 1 can be used to identify the order of most shared accesses, we would still like to have a smaller one.

In Scheme 1, when the operations X=1 and X=2 are executed close to the same time, we may get a smaller range of d. However, even in such cases, the range cannot be shrunk to 1. The reasons include:

1) The time needed by RdTC and fence instructions.
2) The overhead brought by the write buffer flushing.
3) The overhead brought by the cache coherency protocol.

In order to reduce the impact by such influences, we propose another scheme based on statistics. Fig. 6 shows our statistic tester. It has two worker thread (T1, T2) and a trigger thread (T0), which are bound to 3 different cores. The initial value of flag is 0, so T1 and T2 will spin on flag. After thread T0 writes 1 to flag, T1 and T2 finish the while loop and read the local clock, probably at the same time. The difference of their results (TS2 - TS1) is the d we want.

However, in practice, the RdTC instructions in T1 and T2 are unlikely to be executed at the same time for the following reasons:

1) The while loop contains at least 3 instructions: load, compare and branch. When T0 sets 1 to flag, T1 and T2 may not execute the same instruction and they will not exit the while loop at the same time.
2) The cache coherence protocol will delay one of the worker threads. In most modern processors, each core has
private L1 and L2 caches, and the processor use a coherence protocol (e.g., MESIF on Intel x86) to maintain data coherence among cores. When two cores simultaneously read one cache line absent in their private cache, they obtain the data serially [9]. Therefore, one of the cores will suffer a delay. Besides, according to the thread-core mapping strategy, data transfer distance between T1, T2 and T0 may be different. When T0 set 1 to flag, T1 and T2 may not know it at the same time.

(3) Scheduling and interruption may occur between the while loop and RdTC.

(4) When T1 and T2 exit the while loop, ICache Miss or Page Fault may occur.

For the test program in Fig. 6, the effect of the above factors needs to be reduced. Putting the while loop and RdTC in the same cache line can eliminate the factor (4). For factor (3), a kernel module will be helpful. It will prevent the kernel to schedule other threads to the cores which T1 or T2 is bound to. If an interruption occurs during the execution of the test program, it can notify the test program that its result is invalid.

On most modern processors (x86, Power, SPARC and MIPS, etc.), each processor has several cores. Each core has its private L1 cache but share its last level cache (LLC) with other cores. Suppose in Fig. 6 T0 and T2 are bound to the same processor, and T1 is bound to a different processor. T2 will get the new value of flag faster than T1. This is the affection produced by 3 instructions, which take less than 100 cycles. Therefore, the two peaks are generated by factor (3).

There are two peaks in Fig. 7. In the ith run, the time stamp pair of the ith run. In the ith run:

\[ TS2_i = TS1_i + d + \varepsilon_i + \delta_i I_i \]

In Equation (4), if T1 obtains the new value of flag first, the value of \( \delta_i \) is 1; if T1 obtains data first, the value of \( \delta_i \) is -1. We have:

\[
\begin{align*}
    d + \varepsilon_i + \delta_i I_i &= TS2_i - TS1_i \quad \text{(ith T2 gets data first)} \\
    d + \varepsilon_j + \delta_j I_j &= TS2_j - TS1_j \quad \text{(jth T1 gets data first)}
\end{align*}
\]

When the test program is run numerous times, we have:

\[
\begin{align*}
    \sum_{i=1}^{r_1} \sum_{s=1}^{l_1} \varepsilon_i = 0, \quad \sum_{i=1}^{r_1} \sum_{s=1}^{l_1} I_i = 0 \\
    \sum_{l=1}^{r_2} \sum_{i=1}^{r_1} \varepsilon_i = 0, \quad \sum_{l=1}^{r_2} \sum_{s=1}^{l_2} I_i = 0
\end{align*}
\]

Assume in \( r_2 \) runs, \( i_1, i_2, \ldots, i_{r_2} \), T2 obtains data first, while in the \( r_1 \) runs, \( j_1, j_2, \ldots, j_{r_1} \), T1 obtains data first.

When T0 set 1 to flag, which instructions T1 and T2 are executing are random. The effect of factor (1) on T1 and T2 are the same, implying that the expectation of \( \varepsilon \) is 0. If the test program runs numerous times, we can assume that the average of \( \varepsilon \) is 0, that is

\[
\frac{1}{r_2} \sum_{l=1}^{r_2} \varepsilon_i \approx 0, \quad \frac{1}{r_1} \sum_{s=1}^{l_1} I_i \approx 0
\]

According to our thread-core mapping strategy, if the number of runs is large enough, the delay caused by factor (2) is the same for both T1 and T2, that is

\[
\frac{1}{r_2} \sum_{l=1}^{r_2} I_i \approx \frac{1}{r_1} \sum_{s=1}^{l_1} I_i
\]

Therefore, Equation (6) is converted to:

\[
d \approx \frac{1}{r_2} \sum_{l=1}^{r_2} (TS2_{i_l} - TS1_{i_l}) + \frac{1}{r_1} \sum_{s=1}^{l_1} (TS2_{j_s} - TS1_{j_s})/2 \quad \text{(7)}
\]

By Wiener-Khintchines law for large numbers, when the number of test increases to a very large number, the value of \( d \) in Equation 7 will approach a constant. Therefore, we can use the test program in Fig. 6 to estimate the difference of the local clocks on the cores to which T1 and T2 are bound.

To use the above equations, we need to know the thread in Fig. 6 that obtains the new value of flag first. We run the test program in Fig. 6 20 million times. A distribution of \( TSd = TS2 - TS1 \) is shown in Fig. 7.

There are two peaks in Fig. 7. In the ith run, \( TSd_i = TS2_i - TS1_i = d + \varepsilon_i + \delta_i I_i \). The value of \( d \) is fixed. And the value of \( \varepsilon_i \) is affected by 3 instructions, which take less than 10 cycles. In Fig. 7, the distance of the two peaks is more than 100 cycles. Therefore, the two peaks are generated by \( \delta_i I_i \). Then, we regard the center line of the two peaks in Fig. 7 as the boundary. If the value of TSd is on the left side of this boundary, implies that T1 obtains data first, and the value of \( \delta_i \) is 1. Otherwise, the value of \( \delta_i \) is -1.

Using the above equations, we get an approximation of \( d \) (marked as D). However, it is still not precise; we need to calculate the confidence interval of D. According to the central-limit theorem, the value of D approximately has a normal distribution, that is

\[ D \sim N(\mu, \sigma^2) \]

The expectation
of this distribution is the approximated difference of the local clocks on different cores. Assume $D_1, D_2, \ldots, D_n$ are $n$ samples, and $\bar{D}$ and $S^2$ are the sample average and variance respectively. To a given significance level $\alpha$, we expect to find an interval that contains the expectation $\mu$ with a probability $1 - \alpha$. Because the variance $\sigma^2$ of this distribution is unknown, we use sample variance instead of the real variance:

$$P\{\bar{D} - \frac{S}{\sqrt{n}} t_{\frac{1}{2}} \sqrt{n} (n-1) \leq \mu \leq \bar{D} + \frac{S}{\sqrt{n}} t_{\frac{1}{2}} \sqrt{n} (n-1)\} = 1 - \alpha \quad (8)$$

Assume the sample size is $n$, the expectation $\mu$ (i.e., the difference value $d$) has a confidence interval with confidence coefficient $1 - \alpha$:

$$[\bar{D} - \frac{S}{\sqrt{n}} t_{\frac{1}{2}} (n-1), \bar{D} + \frac{S}{\sqrt{n}} t_{\frac{1}{2}} (n-1)] \quad (9)$$

3) Local Clock Overflow: As mentioned earlier, the size of the clock in most processors (expect SPARC and MIPS) is 64-bit, which takes more than ten years to overflow with the current clock frequency. The 63-bit SPARC clock also takes several years. It is enough for most applications. But, for a 32-bit MIPS clock, overflows can occur every a few seconds. Therefore, overflow must be considered and handled.

Assume the overflow cycle of a clock is $P$, we must ensure that the interval between two adjacent records is less than $P$. In such a case, we only need to compare the value of two adjacent records $TSC_{n+1}$ and $TSC_n$: if $TSC_{n+1} - TSC_n < 0$, the clock overflowed; if $TSC_{n+1} - TSC_n > 0$, it did not.

We scan all the records during the offline analysis. When we found the clock overflows, an overflow counter is increased by 1. When we order shared accesses among threads, both the clock and the overflow counter are taken into consideration.

However, since the MIPS clock overflows every few seconds, an interrupt or task rescheduling may make the interval between two records larger than $P$. In practice, we handle an interrupt is short in most cases (in milliseconds), but task rescheduling will affect the accuracy. For half of threads are bound to one core. We did not find two adjacent records whose interval is more than 1 second. Furthermore, a kernel module to record the wall time of scheduling and interruption will solve this problem thoroughly.

IV. REPRODUCING BUGS WITH LOCAL CLOCKS

According to the approach proposed in Section III, we can determine the order of shared accesses among threads by local clocks effectively. In this section, we select two well-known bug reproducing systems PRES and CLAP, and show how to apply our approach to them.

The reason to select them is that: as mentioned in Section I, PRES relies on an expensive scheme to record the global order of some special events. CLAP depends on offline analysis to compute the buggy interleaving, with very low recording overhead. They represent the two key problems: either large recording overhead or long analysis and replay time.

For PRES, its bottleneck is the recording slowdown. We record the value of local clock instead of the global order, and infer the orders of such special points as described in Section III. Our experiments show that without globally recording, the overhead can be reduced up to 85.24%.

For CLAP, our target is to shorten the constraint solving time. Besides recording the execution paths, we select some key points to record their local time stamps, and infer their orders by an efficient offline analysis. These key points can be selected at function calls or loops. We then combine the inferred orders and the original constraints as new input to the SMT solver. Our experiments show that for most benchmarks more than 95% of shared accesses order are determined.

For the remaining unordered shared accesses, we can reduce the solving complexity with the help of local time stamps. Assume the memory operations in Fig. 8 access the same shared memory and all the writes are an integer as its global order number. With the help of local time stamps, we can restrict the value of these integers (i.e., global order number) and shorten the solving time. In Fig. 8(a), for $R_{11}$ in thread T1, CLAP needs to infer the order between $R_{11}$ and all the writes($W_{21}, \ldots, W_{2m}$) in thread T2. However, in Fig. 8(b), if we could know $RdTc3 \prec RdTc1$ and $RdTc2 \prec RdTc4$ by local clock, we only need to infer the order of $R_{11}$, $W_{21}$ and $W_{22}$.

On the other hand, for every shared access, CLAP assigns an integer as its global order number. With the help of local time stamps, we can restrict the value of these integers (i.e., global order number) and shorten the solving time. In Fig. 8(c), to all the five shared accesses, their global order numbers are all within the interval [1,5]. If by the value of local clock, we know $RdTc1 \prec RdTc2 \prec RdTc3 \prec RdTc4 \prec RdTc5 \prec RdTc6 \prec RdTc7$, we can infer that $W1 \prec W2 \prec R1/W3 \prec R2$. This reduces the range of the global order numbers of these shared accesses to [1,1], [2,2], [3,4], [3,4], and [5,5] respectively.

V. EXPERIMENTS

Two systems, PRES-impl and CLAP-impl, are implemented according to the schemes described in PRES [14] and CLAP [4]. We then apply our approach to these two systems, called PRES-tc and CLAP-tc, respectively. In this section, we will evaluate their performance. Table I shows the platform we used in our experiments.

We select several bugs in real multi-threaded programs (TABLE II) as benchmarks. They include widely used servers, desktop applications, and scientific programs. The types of
TABLE III: Reproducing Tries. Add_UO means the additional unordered accesses

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SYNC</th>
<th>FUNC</th>
<th>BB</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRES-impl</td>
<td>PRES-te_S / PRES-te_P</td>
<td>impl</td>
<td>tc_S / tc_P</td>
<td>impl</td>
</tr>
<tr>
<td>PRES-tc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tries</td>
<td>Tries</td>
<td>Tries</td>
<td>Tries</td>
<td>Tries</td>
</tr>
<tr>
<td>APACHE</td>
<td>69</td>
<td>69/69</td>
<td>0.00%/0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>CHEROKEE</td>
<td>46</td>
<td>46/46</td>
<td>0.00%/0.00%</td>
<td>21/21</td>
</tr>
<tr>
<td>PBzip2</td>
<td>3</td>
<td>3/3</td>
<td>0.00%/0.00%</td>
<td>3/3</td>
</tr>
<tr>
<td>PFSCAN</td>
<td>32</td>
<td>32/32</td>
<td>0.00%/0.00%</td>
<td>11/11</td>
</tr>
<tr>
<td>AGENT</td>
<td>14</td>
<td>14/14</td>
<td>0.00%/0.00%</td>
<td>9/9</td>
</tr>
<tr>
<td>BARNES</td>
<td>12</td>
<td>12/12</td>
<td>0.00%/0.00%</td>
<td>4/4</td>
</tr>
<tr>
<td>LU</td>
<td>3</td>
<td>3/3</td>
<td>0.00%/0.00%</td>
<td>6/6</td>
</tr>
<tr>
<td>RADIOSTITY</td>
<td>-</td>
<td>-/ -</td>
<td>0.00%/0.00%</td>
<td>98/98</td>
</tr>
</tbody>
</table>

Fig. 9: Normalized Exec. Time of PRES-impl/PRES-tc

TABLE I: Platform Details

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon E7-4807, 6 cores, 1.87GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>4</td>
</tr>
<tr>
<td>Level 1 Cache (I/D)</td>
<td>6 * 24K / 6 * 24K</td>
</tr>
<tr>
<td>Level 2 Cache</td>
<td>6 * 256K</td>
</tr>
<tr>
<td>Level 3 Cache</td>
<td>18M</td>
</tr>
<tr>
<td>Memory</td>
<td>16G</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.32</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.6.0</td>
</tr>
<tr>
<td>SMT Solver</td>
<td>Z3[24]</td>
</tr>
</tbody>
</table>

TABLE II: Benchmarks

<table>
<thead>
<tr>
<th>TYPE</th>
<th>BENCHMARKS</th>
<th>DESCRIPTION</th>
<th>BUG TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Apache HTTPD[26]</td>
<td>Web server</td>
<td>AV</td>
</tr>
<tr>
<td>Desktop application</td>
<td>PBzip2</td>
<td>Compressor</td>
<td>OV</td>
</tr>
<tr>
<td></td>
<td>Pfscan</td>
<td>File scanner</td>
<td>AV</td>
</tr>
<tr>
<td></td>
<td>Aget</td>
<td>HTTP/FTP downloader</td>
<td>AV</td>
</tr>
<tr>
<td>Scientific application (SPLASH-2) [25]</td>
<td>Barnes</td>
<td>Barnes N-Body algorithm</td>
<td>OV</td>
</tr>
<tr>
<td></td>
<td>LU</td>
<td>LU matrix multiplication</td>
<td>OV</td>
</tr>
<tr>
<td></td>
<td>Radiosity</td>
<td>Graphics rendering</td>
<td>OV</td>
</tr>
</tbody>
</table>

bugs cover common concurrency bugs, such as atomicity violation (AV) and order violation (OV).

In this section, we compare PRES-tc/CLAP-tc with PRES-impl/CLAP-impl, and evaluate our approach. In the experiments, the performance of Apache and Cherokee is measured by their throughputs, and the others are by the execution time. Each thread of these benchmarks is bound to a certain core. System library routines rarely access shared variables, and their accesses can be inferred from their arguments easily, so we do not instrument the system library routines.

A. Evaluating PRES-impl and PRES-tc

PRES-impl records the global order of certain operations, while PRES-tc records their local time stamps instead. It reduces the recording overhead significantly.

Fig. 9. shows the normalized execution time of PRES-impl to PRES-tc with the instrumentation at the synchronization (SYNC), function-call (FUNC), basic-block (BB), and memory-operation (RW) level. The baseline is the native execution time.

PRES-impl can reproduce all of the bugs at the FUNC level within 1000 tries. At the BB level, PRES-impl reproduces all of the bugs within 10 tries. Taking recording overhead and the number of reproducing tries into consideration, instrumentation at these two levels seems reasonably good for PRES-impl. Using local time stamps, PRES-tc can reduce the recording overhead from 320.63% in PRES-impl to 133.48% at the FUNC level on average. At the BB level, the recording overhead is reduced from 1730.05% to 688.34%.

The main reason of the improvement is that PRES-tc removes the synchronizations and allows each thread to record local time stamps concurrently. Take LU as an example, 56.49% and 64.53% of the recordings in PRES-tc are done concurrently at FUNC and BB levels, respectively, and thus 62.44% and 69.24% of the recording overheads are reduced.
At SYNC level, the overheads of the two systems are similar. This is because the number of synchronization operations is very small, and the recording overhead is hidden by the time-consuming synchronization operations. During the execution of LU with default inputs, it has more than 3 million function calls, but only 300 synchronization operations.

For PBZIP2 and AGET, their overheads in both schemes are nearly the same. The reason is that the main work of PBZIP2 and AGET is compressing and downloading via system library routines, but we do not instrument those routines.

PRES records the global orders at RW, BB, FUNC, or SYNC level. Except for the RW level, there could be shared memory accesses that PRES cannot determine their global orders by logs directly. In those cases, PRES will reconstruct the buggy interleavings with more tries.

PRES-tc determines the order of shared memory accesses by a range of d. Compared with PRES, it will bring a small amount of additional unordered shared memory accesses. TABLE III shows the percent of these additional accesses to the total shared memory accesses and the number of tries in both PRES-impl and PRES-tc. PRES-tc_S and PRES-tc_P use the ranges of d calculated by the two schemes described in Section III respectively. We can see from these data that the percent of additional unordered accesses is less than 1% at BB, FUNC, and SYNC levels, which is a small percentage of all shared memory accesses. We also see that PRES-tc needs no more tries than PRES-impl. This is because the goal of PRES is to reproduce bugs, not the interleavings that lead to the bugs. For most concurrency bugs, the root cause is only related to a handful of shared accesses [1], and the majority of shared accesses do not influence the reproduction of the bug. For benchmark LU at RW level, although there are 19.35% to 25.50% of shared accesses whose orders cannot be determined, the bug can still be reproduced in one try. That is because the bug in LU is an error caused by invalid synchronization operations, and the order of accesses determined by local time stamps are enough to reproduce this bug.

Figure 10 shows the recording overhead of PRES-impl and PRES-tc with different numbers of threads. When the number of threads increases, the overhead of PRES-impl increases more quickly in most cases, because the lock is more frequently accessed. For PRES-tc, the thread-private recording benefits its scalability. For LU at the FUNC level, 56.49%, 77.09%, and 83.27% of the recordings are done concurrently when there are 4, 8, and 16 threads, respectively. If there are more threads, a higher percentage of the recording time will be...
done concurrently. As described above, the recording overhead of Pbzip2 and Aget is more or less the same in PRES-impl and PRES-tc because we do not instrument system library routines.

B. Evaluating CLAP-impl and CLAP-tc

CLAP uses an SMT solver to reproduce the buggy interleavings, but the floating-point operations supported by SMT solvers are limited. The bugs in BARNES, LU and RADIOSITY are related to floating point operations. Hence, CLAP does not use them as benchmarks. Therefore, in CLAP-impl, we use these three benchmarks to measure the recording slowdown only. Furthermore, CLAP uses a well-designed test case Racey [20] that contains massive data races and is very likely to produce a different result when the interleaving is different. CLAP uses it to show its capability. Then we also select Racey to evaluate CLAP-tc.

Figure 12 shows the recording overhead of CLAP-impl and CLAP-tc at different instrumentation levels. FUNC records the local time stamps at the entries and exits of functions; LOOP records at the entries, exits and back edges of loops; FUNCLOOP is a combination of FUNC and LOOP. In Fig. 12, we can see that the recording slowdown of CLAP-tc is 101% ∼ 142% of CLAP-impl, and mostly less than 110%.

Figure 11 shows the solving time of CLAP-impl and CLAP-tc at different instrumentation levels. From small to large, each benchmark is tested with 5 different inputs. CLAP reproduces interleavings by solving the constraints with an SMT solver. For the input constraints, we can get the results from the SMT solver first and combine the results with the original input as a new input. The time the solver takes to solve the new input is approximated to the minimum solving time, and we call it near-optimal solving time (NOST). In Fig. 11, we show the ratios of CLAP-impl and CLAP-tc to NOST. CLAP-tc records the local time stamps at three different levels.

Figure 11 shows that compared to CLAP-impl, 84.66% ∼ 99.99% solving time is reduced by CLAP-tc. This is because the local time stamps can determine the orders of most shared memory accesses. In Pbzip2 at the FUNCLOOP level, the local time stamps determine more than 99% of the orders. This makes the solving time of it is very small. Furthermore, with larger inputs, the solving time of CLAP-impl increases much more quickly than CLAP-tc. For example, the solving time of CLAP-impl for Pbzip2 with the largest input is about 1000X longer than that with the smallest input, while the same ratio in CLAP-tc is only 4X.

On the other hand, for most benchmarks, the solving time of CLAP-tc is less than 10X of NOST. Especially, the solving time of Aget is nearly the same as NOST. In our experiments, NOST of all benchmarks is at most several seconds.

In studying Fig. 11 and 12, we can see that the lower the instrumentation level is, the less solving time but the more overhead is introduced. At the FUNC and LOOP levels, the loop bodies may contain complicated function calls, and a function body may contain many loops. This makes their solving time much longer than that at the FUNCLOOP level. The recording overhead at the FUNCLOOP level is a bit more than that at the FUNC and LOOP level. Altogether, we believe FUNCLOOP is a suitable level for instrumentation.

On the other hand, CLAP-tc is less effective for Racey. In Racey, most addresses of write operations are calculated by shared variables. In such cases, if a read happens before a write, it is difficult to infer whether the read and the write access the same shared variable or not. Thus, a few redundant constraints remain in the input for the SMT solver. Even so, the solving time of CLAP-impl is about 5X-100X longer compared to CLAP-tc in our experiments, which also shows the effectiveness of using local time stamps.

C. Value Differences of Local Clocks among Cores

This subsection shows the results of our two schemes to calculate the range of $d$.

1) Program Testing Scheme: We designs four programs to test the ranges of $d$. TABLE IV shows two test results for these programs on the same cores. In each test, every program executes 10K times.

The test platform and the number of test runs could affect the results in TABLE IV. More test runs could generate a smaller range. On our test platform, the test program in Fig. 5(b) gets a larger range than other programs in Fig. 5. This is because the implementation of the serializing instructions on this processor is more time-consuming than others. The results of the other programs are more or less the same. If we want to get a smaller range, we need to run all of the test programs as many times as possible. In TABLE IV, the range of $d$ is from -114 to 110, and is about 200 cycles. Ordering shared access using it will not bring false positives or false negatives.
TABLE IV: Testing Results

<table>
<thead>
<tr>
<th>Testing Program</th>
<th>1st Test</th>
<th>2nd Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN</td>
<td>MAX</td>
</tr>
<tr>
<td>Fig. 5(a)</td>
<td>-114</td>
<td>112</td>
</tr>
<tr>
<td>Fig. 5(b)</td>
<td>-190</td>
<td>182</td>
</tr>
<tr>
<td>Fig. 5(c)</td>
<td>-128</td>
<td>128</td>
</tr>
<tr>
<td>Fig. 5(d)</td>
<td>-116</td>
<td>120</td>
</tr>
<tr>
<td>Result</td>
<td>-114</td>
<td>112</td>
</tr>
</tbody>
</table>

2) Statistics Scheme: Our proposed statistical scheme uses the statistical tester and Equation 7 to calculate the range of \( d \). For Equation 7, we need to know the value of \( \delta_i \), and the testing procedures are the following:

(1) Bind the worker and trigger threads in Fig. 6 according to Section III-B2.

(2) Run the testing program \( N \) times, and get \( N \) results by using Equation (4) \( \delta_i = d + \varepsilon_i + \delta_i I_i = TS2_i - TS1_i \).

(3) Build the distribution of \( \delta_i \) according to Section III-B2 and infer the value of \( \delta_i \) in each execution.

(4) Calculate the value of \( d \) by Equation 6.

Stability. If the number of test runs of the statistical tester is large enough, the result of Equation (7) will be stable. Then we ran this program continuously for more than 10 days, collected 100 million results that shown in Fig. 13.

In this figure, we calculate \( d \) every hour, using about 360,000 runs of statistical tester. Stability means the \( d \) is calculated by data collected in each hour, while Acc Stability means the \( d \) calculated by data since the start of the testing. From these data, we can see that over a long time period (more than 10 days), the calculated \( d \) in each hour are all in the interval \([-0.0885, 0.1827]\), and their sample variance is 0.001379. This means that the calculated \( d \) is very stable.

Confidence Interval. Now we calculate an approximation of \( d \). Following, we calculate the confidence interval of \( d \) under different confidence coefficient by Equation 9. The confidence interval requires many samples of \( d \). We calculates \( d \) using the method in Section III-B2 for many times and get \( d_1, d_2, d_3, \cdots, d_M \). Each \( d_i \) is the result of \( N \) runs of the program in Fig. 6. Finally we get the data shown in TABLE V by Equation 9 using these \( d_i \).

In TABLE V, the higher the confidence coefficient is, the larger the range is. When the confidence coefficient is fixed, the values of \( N \) and \( M \) vary inversely with the confidence intervals. In practice, we could calculate confidence intervals with different confidence coefficients according to the target program. TABLE V shows that when the confidence coefficient is 0.99999, the \( N \) is 20 and the \( M \) is 5. The range of the confidence interval is about 100, which is still smaller than the range obtained by program testing. If the \( N \) is 100 and \( M \) is 20, the range is only about 1.5.

VI. RELATED WORK

For most record-and-replay or other bug reproducing systems, the focus has been on reducing the recording overhead, however, this is often traded with high offline analysis cost and long bug reproducing time. Our approach takes advantage of the local clock, and can reduce both recording slowdown and the bug reproducing time.

PRES [14] do not record all the global order during recording, and try to reproduce bugs by offline analysis. It only records the global order of some special events, such as synchronizations, system calls, function calls, basic blocks, and memory instructions. During offline analysis, it searches for the buggy interleaving by exploration.

Some systems reduce the recording slowdown by record other information that imply the global order of shared accesses. SMP-Revirt [12] and Scribe [13] make use of the page protection mechanism, and record the ownership transfer of pages among threads to infer the order of shared accesses. For programs with little false sharing, Scribe has good performance. However, for programs with intensive false sharing, its recording overhead could be very large. DoublePlay [21] divides the program into many epochs by time interval. Besides concurrent execution, DoublePlay forks a new process to run epochs serially at the beginning of every epoch. It only needs to record the order of epochs, hence dramatically reduce recording overhead. And if the results of concurrent and serial execution are different, a rollback is needed. For programs with many races, the rollback overhead can be large. Besides, these systems affect the behavior of multi-threaded programs, and some bug will never be exposed.

There are also systems that record mostly local information to avoid global synchronization. CLAP [4] makes each thread record its own execution paths and searches for buggy interleavings by a SMT solver. ODR [17] reproduces concurrency bugs by ensuring the same output as recording execution. It only records the global order of synchronization operations during execution. In reproducing, similar to CLAP, it generates many interleavings and verifies their outputs by an SMT solver.

CoreDump [19] makes use of the core dump when a program crashes. It records the number of iterations for loops at run time, and incurs little overhead. According to the error point, it searches for a similar point to generate a right core dump. Comparing the core dumps of these two points, it tries to explore the buggy interleaving.

LReplay [15] uses global clock. It expects that future processors will provide a global clock with a fast access time.
With such a global clock, LReplay only needs to record orders that cannot be inferred from the global time.

VII. CONCLUSION

In order to reproduce the concurrency bugs in multi-threaded programs more efficiently, this paper takes advantage of the local per-core clocks on modern processors. During the recording phase, each thread records its own data and local time stamps to avoid expensive synchronization operations among threads. The local time stamps are used to determine the global order of shared-resource accesses. We have proposed two effective schemes to calculate the time difference among local per-core clocks. Our experiments show that after applying the proposed approach to PRES and CLAP, two well-known record-and-replay schemes, the recording overheads and solving time can be reduced by 1% ∼ 85% and 84.66% ∼ 99.99% respectively.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their useful feedback. This research is supported by the National High Technology Research and Development Program of China under grant 2012AA010901, the National Natural Science Foundation of China (NSFC) under grants 61303051, 61303052, 61332009, 60925009, and 61100011, the Innovation Research Group of NSFC under grant 61221062.

REFERENCES