2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

ReCBuLC: Reproducing Concurrency Bugs Using Local Clocks

Xiang Yuan[1,3], Chenggang Wu[1*], Zhenjiang Wang[1], Jianjun Li[1], Pen-Chung Yew[4],
Jeff Huang[5], Xiaobing Feng[1], Yanyan Lan[2], Yunji Chen[1] and Yong Guan[6]
[1]State Key Laboratory of Computer Architecture / [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
[3]University of Chinese Academy of Sciences, Beijing, China
[4]Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, USA. yew@cs.umn.edu
[5]Department of Computer Science and Engineering, Texas A&M University, Texas, USA. jeff@cse.tamu.edu
[6]College of Information Engineering, Capital Normal University, Beijing, China. guanyong@mail.cnu.edu.cn

***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, ReCBuLC, to take advantage of the hardware clocks available on modern processors. The key idea is to reduce the recording overhead and analyzing events' global order by using time stamps recorded in each thread. Those timestamps are used to determine the global orders of shared accesses. 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 obtain an 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% $\sim$ 85%, and the analysis time by 84.66% $\sim$ 99.99%, respectively.**

***Index Terms*—concurrency, bug reproducing, local clock**

I. Introduction

Parallel programming is essential to fully utilize the compute power of multi-core processors. However, debugging such programs has become a major challenge for software developers because of the non-deterministic nature of parallel programs [30]. A survey showed that it took about 73 days to fix a concurrency 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 fixed as quickly as possible.

One of the main debugging techniques is Record & Replay (RR). It faithfully records the execution interleaving and deterministically replays the same interleaving to reproduce bugs [28][18][16][22][23]. The main challenge in RR techniques is the need to reduce the significant performance penalty incurred at runtime to record the interleaving information. Some RR techniques [10][11] could incur 10X$\sim$100X slowdown. Furthermore, the perturbation caused by the instrumented code and the recording overhead may alter the interleaving sequence of the program execution, which can obscure some bugs especially on systems with weak memory models [4].

To address those limitations, several schemes have been proposed to improve RR techniques by recording only minimally needed interleaving information, and then reproduces the buggy interleaving with offline analysis and guided exploration. Because significantly less information is recorded, the runtime overhead can also be substantially lower. Many systems adopt this idea [17][19][21][14][4]. Although the interleaving reproduced by these schemes may not be exactly the same as the original one, they are useful in practice as the same failure can be reproduced.

For example, PRES [14] records the global orders of some special events, such as 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, it can reproduce bugs in 10 tries mostly, and experiences only 10%$\sim$779% slowdown [14].

Similar to other RR techniques, 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 synchronizations, an effective mechanism called CLAP [4] has been proposed. Each thread in CLAP only records its local information. During the offline analysis, CLAP generates constraints by symbolic execution and searches for buggy interleavings using a Satisfiability Modulo Theories (SMT) solver, such as Yices [29] and Z3 [24]. Thus, its slowdown is only about 9%$\sim$294%. However, it cannot get the buggy interleavings directly. Instead, it relies on an SMT solver, which is hard to scale because such constraint solving is NP-hard.

These systems trade off between less time in the record phase and more time in the analysis and replay phase. It

*To whom correspondence should be addressed

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.94

is thus very desirable to find a scheme that requires each thread to log only its own local information while allowing a quick offline analysis to get the order of shared accesses among threads during the replay. Such a scheme could greatly greatly improve the efficiency of program debugging for multi-threaded programs.

The key insight here is to take the advantage of the available hardware per-core local clocks 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 core can access its own local clock without any need for synchronization with other cores. The order of shared accesses can then be inferred accordingly. However, these local clocks are core-private. And the hardware don't guarantee them to be consistent, i.e., they may have different skews among themselves. 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 find an effective way to resolve these local timestamps and determine a global order among them.

In this paper, we proposes a new mechanism, ReCBuLC, to reconstruct the order of shared-memory accesses among threads using local timestamps. This scheme can thus be used to reproduce concurrency bugs more efficient. As a demonstration, we apply ReCBuLC to two recent systems, and show that it can significantly improved their performance.

Our contributions are as follows:

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

(2) We present a methodology to obtain a range of the skews among per-core clocks. We then use a statistical scheme to narrow down the range of clock skews to less than 10 ticks (10 cycles) with a high confidence.

(3) ReCBuLC is applied to two recent systems and shows that it can improve their efficiency significantly.

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

II. Motivation

Almost all mainstream commercial processors provide local per-core clocks, and applications can access them for needed timing information. For example, Intel/AMD x86 processors provide a 64-bit Time Stamp Counter (TSC) since Pentium family. The TSC is incremented at a constant rate with respect to the wall-clock time. It is not affected by the frequency of processors so as to avoid the impact of dynamical frequency scaling [5]. Similar mechanisms exist on other processors. On IBM Power processors, every core has a 64-bit Time Base register [7], while its counting frequency can be changed by software. 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 Count Register [6], but its size is only 32-bit. SPARC processors have a 63-bit Tick register [8] to keep the clock cycles.

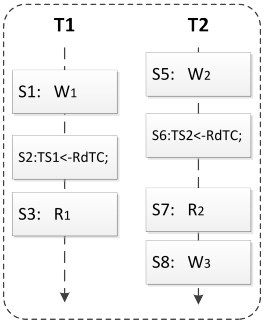

Fig. 1: Happen-Before Determined by Local Clock

"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. They will need neither synchronization when being recorded, nor constraints solving when being reproduced, 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* happens before *S2* (i.e. $S6 \prec S2$), we can infer that $S5 \prec S3$.

However, although hardware local clocks have existed for a long while, no systems has used them to record and 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 $\sim$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 per-core local clocks to determine the global order of shared accesses:

(1) We need to deal with the differences among different per-core local 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., *rdtsc* on x86. They can be recorded 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. Furthermore, there is no data dependency between *RdTC* and the target 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 has only 32-bits, 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 on the same chip, their local clocks are triggered by the same clock signal. They count at the same frequency, and the differences among them will be the same after processor reset. To different processors, if they are of the same type and use the same crystal oscillator, the difference of their local clocks is highly likely 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 rest 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. We thus begin with this challenge.

A. Out-Of-Order Execution Exclusion

Most modern 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 the 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-core 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 still not 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)* $\prec$ *RdTC1*, but cannot control *W1(GV)*. From the value of the local clock, we would infer that *W1* $\prec$ *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

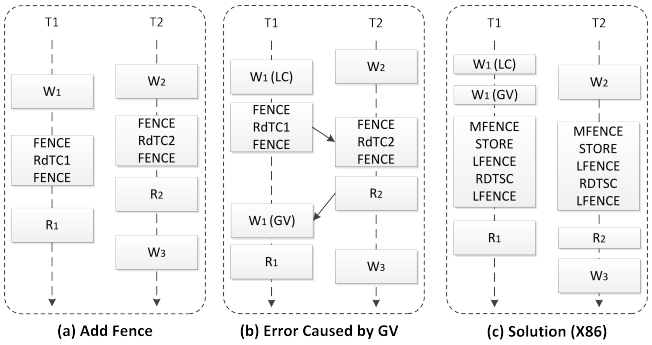

Fig. 2: TC Order

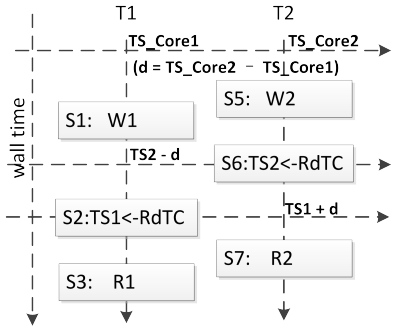

Fig. 3: Determine Orders by Local Clock

instructions until preceding instructions are locally complete. A correct implementation on x86 is thus shown in Fig. 2(c) (*STORE* means a memory store instruction, which is used by *MFENCE*). In thread T1, *MFENCE* and *LFENCE* guarantee *W1(GV)* $\prec$ *RdTC1*, while *LFENCE* guarantees *RdTC1* $\prec$ *R1*.

B. Handling the Time Difference among Per-Core Clocks

Although per-core clock values among cores could be different at any instance of time, we can still make use of them if we know their differences (called **d**). An example is 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* $\prec$ *S2* (i.e. *RdTC2* $\prec$ *RdTC1*). We can infer that *S5* $\prec$ *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 still can determine the order of shared accesses among threads.

Taking Fig. 3 as an example, assume d $\in$ [d1, d2]. If *TS2* - d1 $<$ *TS1*, we have *TS2* - d $<$ *TS2* - d1 $<$ *TS1*, and this means *S6* $\prec$ *S2*. We can thus infer *S5* $\prec$ *S3*. Similarly if *TS1* + d2 $<$ *TS2*, we can infer *S1* $\prec$ *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:

(Scheme 1) Use test programs to obtain an range of d.

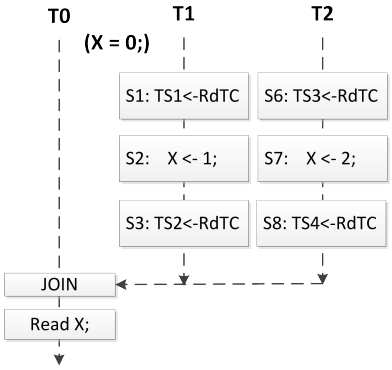

Fig. 4: Local Clock Difference Tester

(Scheme 2) Use statistical means to obtain a smaller range of d with high confidence

1) Scheme 1-Use Test Programs: We designed a small test 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, S7 in T2 must be later than S2 in T1, so we can infer *S1* $\prec$ *S2* $\prec$ *S7* $\prec$ *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:

$$d < TS4 - TS1 \quad (if\ \textit{Read X}\ is\ 2) \tag{1}$$

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

$$d > TS3 - TS2 \quad (if\ \textit{Read X}\ is\ 1) \tag{2}$$

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 to 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:

$$\max_i(TS3_i - TS2_i) < d < \min_i(TS4_i - TS1_i) \tag{3}$$

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 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

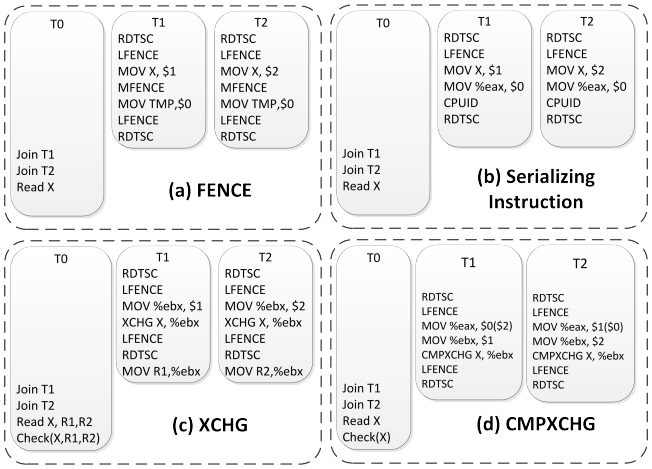

Fig. 5: Difference Tester Implementation

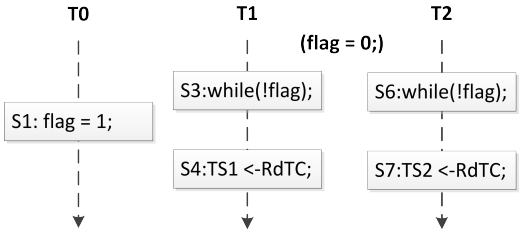

Fig. 6: Statistic Tester

implementation does not guarantee that the GV of writing *X* happen before *RDTSC*, so we need to check whether it does. 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 for the expensive *MFENCE* or *CPUID* is replaced. The efficacy of these versions depends on the hardware implementation, but we can always run all of them many times to obtain a minimum range of d.

2) Scheme 2-Statistical 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 coherence protocol.

In order to reduce their impact, we propose another scheme based on statistics. Fig. 6 shows our statistic tester. It has two worker threads (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, hopefully about the same time. The difference of their results (*TS2* - *TS1*) is the d we want.

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

(Factor 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.

(Factor 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 uses 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.

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

(Factor 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 codes of *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 that 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. 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 effect from thread-core mapping strategy. We then use such a thread-core mapping: (1) if we want to calculate the d of two cores on the same processor, T1 and T2 are bound to cores in one processor; or (2) T1 and T2 are bound to two different processors, and in each run, T0 is randomly bound to either of the two processors that T1 and T2 are bound to.

We use ε and I to represent the effects of factors (1) and (2), and run the test program repeatedly. Assume $< TS1_i, TS2_i >$ is the timestamp pair of the i-th run, then:

$TS2_i = TS1_i + d + \varepsilon_i + \delta_i I_i$, that is:

$$d + \varepsilon_i + \delta_i I_i = TS2_i - TS1_i \tag{4}$$

In Equation (4), if T1 obtains the new value of *flag* first, the value of δ_i is 1; otherwise, the value of δ_i is -1. We have:

$$\begin{cases} d + \varepsilon_i - I_i = TS2_i - TS1_i \ (ith\ T2\ gets\ data\ first) \\ d + \varepsilon_j + I_j = TS2_j - TS1_j \ (jth\ T1\ gets\ data\ first) \end{cases} \tag{5}$$

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

$$\begin{cases} d + \frac{1}{r_2}\sum_{l=1}^{r_2}\varepsilon_{i_l} - \frac{1}{r_2}\sum_{l=1}^{r_2} I_{i_l} = \frac{1}{r_2}\sum_{l=1}^{r_2}(TS2_{i_l} - TS1_{i_l}) \\ d + \frac{1}{r_1}\sum_{s=1}^{r_1}\varepsilon_{j_s} + \frac{1}{r_1}\sum_{s=1}^{r_1} I_{j_s} = \frac{1}{r_1}\sum_{s=1}^{r_1}(TS2_{j_s} - TS1_{j_s}) \end{cases} \tag{6}$$

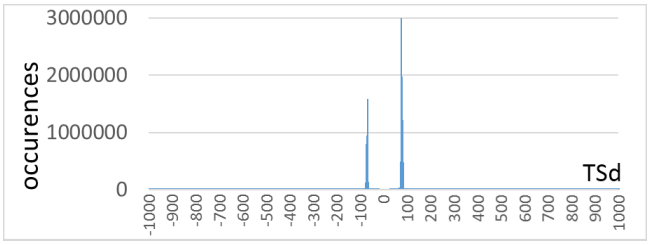

Fig. 7: A Distribution of TSd

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

When T0 set *flag* to 1, the instructions that T1 and T2 are executing are random. The effect of factor (1) on T1 and T2 are the same, implying that the expectation value of ε is 0. If the test program runs numerous times, we can assume that the average of ε is 0, that is $\frac{1}{r_2}\sum_{l=1}^{r_2}\varepsilon_{i_l} \approx \frac{1}{r_1}\sum_{s=1}^{r_1}\varepsilon_{j_s} \approx 0$. 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 $\frac{1}{r_2}\sum_{l=1}^{r_2} I_{i_l} \approx \frac{1}{r_1}\sum_{s=1}^{r_1} I_{j_s}$. Therefore, Equation (6) is converted to:

$$d \approx (\frac{1}{r_2}\sum_{l=1}^{r_2}(TS2_{i_l} - TS1_{i_l}) + \frac{1}{r_1}\sum_{s=1}^{r_1}(TS2_{j_s} - TS1_{j_s}))/2 \tag{7}$$

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 spikes in Fig. 7. In the i-th run, $TSd_i = TS2_i - TS1_i = d + \varepsilon_i + \delta_i I_i$. The value of d is fixed. And the value of ε_i is affected by 3 instructions, which take less than 10 cycles. In Fig. 7, the distance of the two spikes is more than 100 cycles. Therefore, the two spikes are generated by δI. We regard the center line of the two spikes in Fig. 7. as the boundary. If the value of TSd is on the left side of this boundary, it implies that T1 obtains data first, and the value of δ_i is 1. Otherwise, the value of δ_i is -1.

Using the above equations, we get an approximation of d (marked as D). However, it is still not precise enough. 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 value of this distribution is the approximated difference of the local clocks on different cores. Assume $D_1, D_2, \cdots, D_n$ are n samples, and $\bar{D}$ and S^2 are the sample average and variance respectively. To a given significance level α, we expect to find an interval that contains the expectation μ with a probability $1-\alpha$. Because the variance σ^2 of this distribution is unknown, we use sample variance instead of the real variance:

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

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

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

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, the time used to handle an interrupt is short in most cases (in milliseconds), but task rescheduling will affect the accuracy if dozens of threads are bound to one core. We did not find two adjacent records whose interval is more than 1 second. Furthermore, using a kernel module to record the wall clock time of the scheduling and interruption could solve this problem thoroughly.

IV. Reproducing Bugs Using Local Clocks

In this section, we select two well-known bug reproducing systems PRES and CLAP, and show how to apply our approach to them. The reasons to select them are: 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: 1.large recording overhead; 2.long analysis and replay time.

To apply our approach to PRES and CLAP, first, we need to bind each thread to a different core. We then use our technique described in Section III.B to calculate the range of *d* of these cores in advance.

For PRES, its bottleneck is the recording phase. 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 by up to 85.24%.

For CLAP, our goal is to shorten the constraint solving time. Besides recording the execution paths, we select some key points to record their local timestamps, 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 can be ordered.

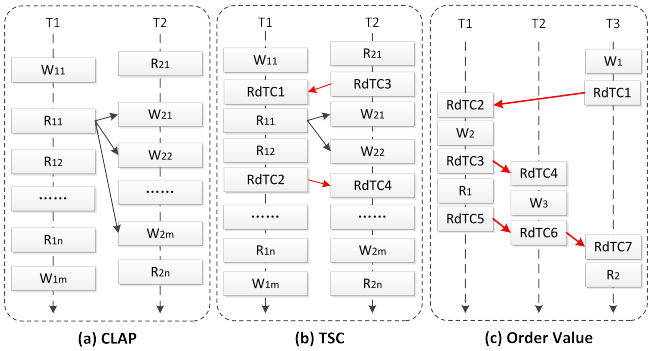

Fig. 8: Constraints Reduction

For the remaining unordered shared accesses, we can also reduce the solving complexity with the help of local timestamps. Assume the memory operations in Fig. 8 access the same shared variable. In Fig. 8(a), for R_{11} in thread T1, CLAP needs to infer the order between R_{11} and all the writes$(W_{21}, \cdots, W_{2m})$ in thread T2. However, in Fig. 8(b), if we could know *RdTC3* $\prec$ *RdTC1* and *RdTC2* $\prec$ *RdTC4* by local timestamps, 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 timestamps, we can restrict the range of these global order numbers and shorten the solving time. In Fig. 8(c), the global order numbers of the five shared accesses 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 can reduce 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 ReCBuLC to these two systems, called PRES-tc and CLAP-tc, respectively. In order to use the local clocks, each thread of them is bound to a different core. In this section, we will evaluate their performance and Table I shows the platform. 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 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.

TABLE III: REPRODUCING TRIES. ADD_UO MEANS THE ADDITIONAL UNORDERED ACCESSES

Benchmarks	SYNC			FUNC			BB			RW		
	PRES-impl	PRES-tc_S / PRES-tc_P		impl	tc_S / tc_P		impl	tc_S / tc_P		impl	tc_S / tc_P	
	Tries	Add_UO	Tries	Tries	Add_UO	Tries	Tries	Add_UO	Tries	Tries	Add_UO	Tries
APACHE	69	0.00%/0.00%	69/69	5	0.01%/0.01%	5/5	1	0.02%/0.07%	1/1	1	0.05%/0.09%	1/1
CHEROKEE	46	0.00%/0.00%	46/46	21	0.00%/0.00%	21/21	8	0.00%/0.00%	8/8	1	0.02%/0.03%	1/1
PBzip2	3	0.00%/0.00%	3/3	3	0.00%/0.00%	3/3	2	0.00%/0.00%	2/2	1	0.00%/0.00%	1/1
PFSCAN	32	0.00%/0.00%	32/32	11	0.00%/0.00%	11/11	1	0.05%/0.18%	1/1	1	2.94%/4.42%	1/1
AGET	14	0.00%/0.00%	14/14	9	0.00%/0.00%	9/9	1	0.00%/0.00%	3/3	1	0.00%/0.00%	1/1
BARNES	12	0.00%/0.00%	12/12	4	0.00%/0.00%	4/4	1	0.00%/0.00%	1/1	1	0.24%/0.36%	1/1
LU	3	0.00%/0.00%	10/10	6	0.04%/0.15%	6/6	1	0.27%/0.79%	1/1	1	19.35%/25.50%	1/1
RADIOSITY	-	0.00%/0.00%	-/-	98	0.00%/0.03%	98	1	0.07%/0.21%	1/1	1	3.10%/4.88%	1/1

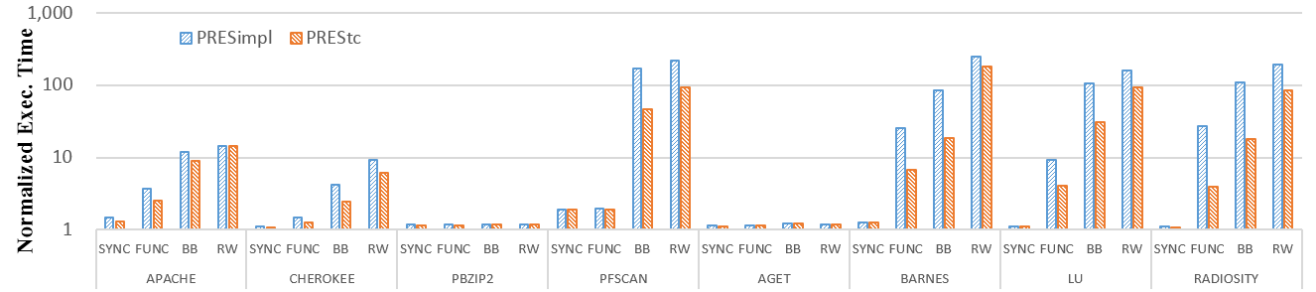

Fig. 9: Normalized Exec. Time of PRES-impl/PRES-tc

TABLE I: PLATFORM DETAILS

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

TABLE II: BENCHMARKS

TYPE	BENCHMARKS	DESCRIPTION	BUG TYPES
Server	Apache HTTPD[26]	Web server	AV
	Cherokee[27]	Web server	AV
Desktop application	PBzip2	Compressor	OV
	Pfscan	File scanner	AV
	Aget	HTTP/FTP downloader	AV
Scientific application (SPLASH-2) [25]	Barnes	Barnes N-Body algorithm	OV
	LU	LU matrix multiplication	OV
	Radiosity	Graphics rendering	OV

System library routines rarely access shared variables and their accesses can be inferred from their arguments easily, so we do not consider them.

A. Evaluating PRES-impl and PRES-tc

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

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

PRES [14] can reproduce all of the bugs at the FUNC level within 1000 tries. At the BB level, PRES 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. PRES-tc reduces 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 for the improvement is that PRES-tc avoids the synchronizations and allows each thread to record local timestamps 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 workload of PBZIP2 and AGET is compressing and downloading data

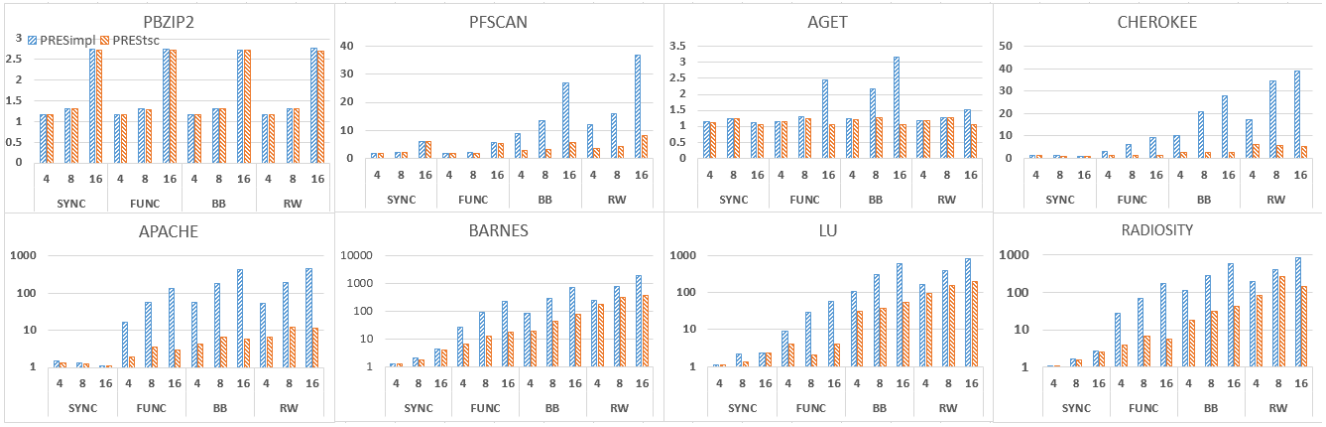

Fig. 10: Scalability of PRES-impl/PRES-tc. The y-axis is normalized exec. time, and in the 4 lower sub graphs are logarithmic.

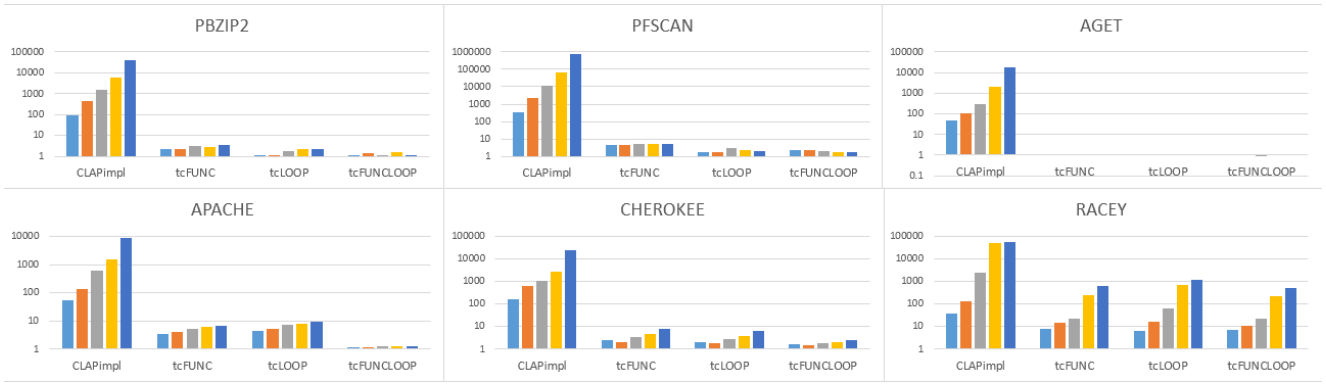

Fig. 11: Normalized Solving Time of CLAP-impl/CLAP-tc. Each benchmark is evaluated with 5 inputs.

using system library routines, but we do not instrument those routines as mentioned earlier.

PRES-tc determines the order of shared memory accesses by a range of d. Compared with PRES-impl, it will bring a small amount of additional unordered shared memory accesses. TABLE III shows the percent of them to the total shared memory accesses and the number of tries in both PRES-impl and PRES-tc. PRES-tc_P and PRES-tc_S 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, and for most concurrency bugs, the root cause is only related to a handful of shared accesses [1]. For LU at RW level, although there are 19.35%∼25.50% unordered shared accesses, the bug can still be reproduced in one try. That is because the bug in LU is caused by invalid synchronization operations, and the order of accesses determined by local timestamps is 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.

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. 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. We also use Racey to evaluate CLAP-tc. For better performance, the range of *d* used by CLAP-tc is calculated using *Statistics Testing*.

Figure 12 shows the recording overhead of CLAP-impl and CLAP-tc at different instrumentation levels. FUNC records the local timestamps at the entries and exits of functions;

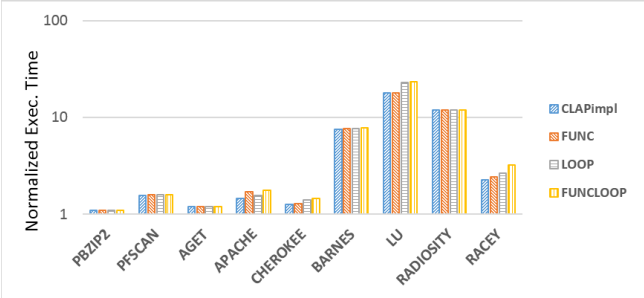

Fig. 12: Recording Overhead of CLAP-impl/CLAP-tc.

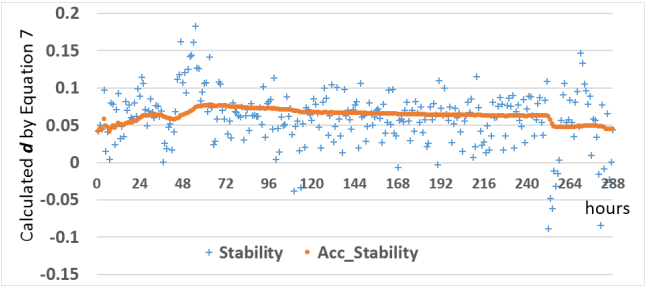

Fig. 13: Stability of Equation 7.

TABLE IV: PROGRAM TESTING RESULTS

Testing Program	1^{st} Test		2^{nd} Test	
	MIN	MAX	MIN	MAX
Fig. 5(a)	**-114**	**112**	**-116**	120
Fig. 5(b)	-190	182	-180	188
Fig. 5(c)	-128	128	-124	126
Fig. 5(d)	-116	120	-120	**110**
Result	-114	112	-116	110

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. During bug reproducing, 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 timestamps at three different levels.

Figure 11 shows that, compared to CLAP-impl, CLAP-tc reduces solving time by 84.66% ∼99.99%. This is because the orders of most shared memory accesses are determined by local timestamps. In PBZIP2 at the FUNCLOOP level, the local timestamps determine more than 99% of the orders. This reduces the solving time substantially. Furthermore, with larger inputs, the solving time of CLAP-impl increases much more quickly than that of CLAP-tc. In PBZIP2, the solving time of CLAP-impl with the largest input is about 1000X to the smallest input, while the ratio of 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. In TABLE IV, the range of d is about 200 cycles. Using is to order shared access will not bring false positives or false negatives.

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 δ_i, and the test procedure is as follows:

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

(2) Run the test 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 δ_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. We ran this program continuously for more than 10 days, collected around 100 million results that are shown in Fig. 13.

TABLE V: CONFIDENCE INTERVALS. THE FIRST COLUMN IS THE CONFIDENCE COEFFICIENT. THE FIRST ROW IS THE VALUE OF N, AND THE SECOND ROW IS THE VALUE OF M

N	20			50			100		
M	5	10	20	5	10	20	5	10	20
0.99	[-5.86, 9.39]	[-1.69, 3.66]	[-0.63, 1.75]	[-1.48, 1.88]	[-0.41 1.08	[-0.25 0.58	[-1.21 0.57	[-0.80 0.39	[-0.29 0.42
0.999	[-12.50, 16.03]	[-2.95, 4.92]	[-1.05, 2.17]	[-2.93, 3.34]	[-0.77 1.43	[-0.40 0.73	[-1.99 1.34	[-1.08 0.67	[-0.41 0.55
0.9999	[-23.98, 27.51]	[-4.45, 6.42]	[-1.47, 2.59]	[-5.46, 5.86]	[-1.18 1.85	[-0.55 0.88	[-3.33 2.68	[-1.41 1.00	[-0.54 0.68
0.99999	[-44.23, 47.77]	[-6.29, 8.26]	[-1.91, 3.03]	[-9.91, 10.31	[-1.70 2.36	[-0.70 1.03	[-5.69 5.04	[-1.82 1.41	[-0.67 0.81

In this figure, we calculate d every hour, using about 360,000 runs of the statistical tester. Stability means the d is calculated by the data collected in each hour, while Acc_Stability means the d calculated by the data from the beginning. 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. We calculate its confidence interval under different confidence coefficients using Equation 9. The confidence interval requires many samples of d. We calculates d using the method described in Section III-B2 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.

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. Our approach takes advantage of the local clock, and can reduce both recording overhead and the bug reproducing time.

PRES [14] does not record all the global order during recording, and tries 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. They 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 significant false sharing, its recording overhead could be very large. DoublePlay [21] divides the program into many epochs by time intervals. Besides concurrent execution, DoublePlay forks new processes to run epochs serially at the beginning of every epoch. It only needs to record the order of epochs, hence, dramatically reduce recording overhead. 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 bugs may 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 in loops at run time, and incurs little overhead. Depending on 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 timestamps. It expects future processors to 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 proposes ReCBuLC, which takes advantage of the local per-core clocks on modern processors. During the recording phase, each thread records its own data and local timestamps to avoid expensive synchronization operations among threads. The local clocks are used to determine the global order of shared-resource accesses. We have proposed two effective schemes to calculate the time difference among local clocks. Our experiments show that after applying ReCBuLC to PRES and CLAP, two well-known record-and-replay schemes, the recording overheads and solving time can be reduced by 1% $\sim$ 85% and 84.66% $\sim$ 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

[1] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. Learning from mistakes a comprehensive study of real world concurrency bug characteristics. In ASPLOS, 2008.

[2] Nancy Leveson, Clark S. Turner. An investigation of the therac-25 accidents. Computer, 26(7):18C41, 1993.

[3] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016

[4] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In PLDI, 2013.

[5] Intel 64 and IA-32 Architectures Software Developers Manual. September 2013.

[6] MIPS Architecture For Programmers. Revision 3.12. April 28, 2011.

[7] Power ISA Version 2.07. May 3, 2013.

[8] Oracle SPARC Architecture 2011. July, 2012.

[9] J.R. Goodman and H.H.J. Hum. MESIF: A Two-Hop Cache Coherence Protocol for Point-to-Point Interconnects (2009).

[10] Thomas. J. Leblanc and John. M. Mellor-Crummey. Debugging parallel programs with instant replay. IEEE Trans. Comput., 36(4), 1987.

[11] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In SIGMETRICS, 2006.

[12] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, Peter M. Chen. Execution replay of multiprocessor virtual machines. In VEE, 2008.

[13] Soren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In SIGMETRICS, 2010.

[14] 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 SOSP, 2009.

[15] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. LReplay: A Pending Period Based Deterministic Replay Scheme. In ISCA, 2010.

[16] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, Jason Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In ASPLOS, 2010.

[17] Gautam Altekar, Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In SOSP, 2009.

[18] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In ISCA, 2008.

[19] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In ASPLOS, 2010.

[20] Min Xu, Rastislav Bodik, and Mark Hill. A flight data recorder for full-system multiprocessor deterministic replay. In ISCA, 2003.

[21] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In ASPLOS, 2012.

[22] Derek Hower and Mark Hill. Rerun: Exploiting episodes for lightweight memory race recording. In ISCA, 2008.

[23] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In FSE, 2010.

[24] Leonardo De Moura and Nikolaj Bjorner. Z3: an efficient SMT solver. In TACAS, 2008.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In ISCA, 1995.

[26] Apache HTTPD. http://httpd.apache.org/

[27] Cherokee Web Server. http://cherokee-project.com/

[28] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: hybrid program analysis for determinism. In PLDI, 2012.

[29] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, 2006.

[30] J. GRAY. Why do computers stop and what can be done about it? In Buroautomation (1985), pp. 128-145.