Using Local Clocks to Reproduce Concurrency Bugs

Zhe Wang, Chenggang Wu, Xiang Yuan, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, *Fellow, IEEE*, Jeff Huang, *Member, IEEE*, Xiaobing Feng, Yanyan Lan, Yunji Chen, Yuanming Lai, and Yong Guan

Abstract—Multi-threaded programs play an increasingly important role in current multi-core environments. Exposing concurrency bugs and debugging such multi-threaded programs are quite challenging due to their inherent non-determinism. In order to mitigate such non-determinism, many approaches such as record-and-replay 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 efficient and effective approach, ReCBuLC (reproducing concurrency bugs using local clocks), to take advantage of the hardware clocks available on modern processors. The key idea is to reduce the recording overhead and the time to analyze events' global order by recording timestamps in each thread. These timestamps are used to determine the global order of shared accesses. To avoid the large overhead 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 skews 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 their recording overheads by up to 85 percent, and the analysis time up to 84.66%∼99.99%, respectively.

Index Terms—Concurrency, bug reproducing, local clock

1 Introduction

Parallel programming is essential to fulfill the full potential of multi-core processors. However, debugging such programs has become a major challenge because of the non-deterministic nature of parallel programs [39]. A survey showed that it could take an average of 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 located and fixed as quickly as possible.

One of the main debugging techniques is Record & Replay (RR). It faithfully records the thread interleaving during the execution and deterministically replays the same interleaving to reproduce bugs [20], [22], [26], [27], [37]. The main challenge in RR is the need to reduce the significant overhead incurred in the recording phase. Some RR techniques [14], [15] could incur 10X∼100X slowdown. Furthermore, the perturbation caused by the instrumented code and the recording overhead may alter the interleaving behavior of the program execution, which can obscure some bugs especially on systems with weak memory models [6].

To address those challenges, several schemes have been proposed to record only minimally required interleaving information, and reproduce the buggy interleaving using offline analysis and guided exploration. Because significantly less information is recorded, the runtime overhead can be substantially reduced. Many systems adopt this approach [6], [18], [21], [23], [25]. Although the interleaving thus reproduced may not be exactly the same as the original one, they are useful in practice because the same failure can still be faithfully reproduced.

For example, PRES [18] 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 order of the shared accesses that leads to the bug. At the function-call level, it can reproduce bugs in at most 10 tries, and experiences around 10%∼779% slowdown [18].

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 event logging or increment of a global event counter, which are the root cause of the significant overheads [6].

- *Z. Wang is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, and with the University of Chinese Academy of Sciences, Huairou, Beijing 100190, P.R. China. E-mail: wangzhe12@ict.ac.cn.*
- *C. Wu, X. Feng, Y. Lan, Y. Chen, and Y. Lai are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Huairou, Beijing 100190, P.R. China. E-mail: {wucg, fxb, lanyanyan, cyj, laiyuanming}@ict.ac.cn.*
- *P.-C. Yew is with the Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, MN 55455. E-mail: yew@cs.umn.edu.*
- *J. Huang is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843. E-mail: jeff@cse.tamu.edu.*
- *Y. Guan is with the College of Information Engineering, Capital Normal University, Huairou, Beijing 100190, P.R. China. E-mail: guanyong@mail.cnu.edu.cn.*
- *X. Yuan and Z. Wang are now with Huawei Technologies, Huairou, Beijing 100190, P.R. China. E-mail: {yuanxiang4, zj.wang}@huawei.com.*
- *J. Li is now with Horizon Robotics, Inc. Huairou, Beijing 100190, P.R. China. E-mail: jianjun.li@hobot.cc.*

Manuscript received 30 Dec. 2016; revised 30 Aug. 2017; accepted 10 Sept. 2017. Date of publication 14 Sept. 2017; date of current version 9 Nov. 2018. (Corresponding author: Chenggang Wu.)
Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2752158

To avoid such expensive synchronizations, an effective mechanism (called CLAP [6]) was 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 [38] and Z3 [28]. Thus, its slowdown is reduced to about 9∼294 percent. 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 traded off less time in the record phase with more time in the analysis and replay phase. It is thus very desirable to find a scheme that does not require these difficult tradeoffs. Such a scheme could greatly improve the efficiency of program debugging. One key insight here is to take 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. The hardware does not guarantee them to be consistent, i.e., there may be different skews among these clocks. It is quite difficult to get the precise skews among these local clocks (unless there is a global clock as assumed in [19]). 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 propose a new mechanism to reproduce currency bugs using local clocks, ReCBuLC, and to reconstruct the order of shared-memory accesses among threads using local timestamps. We apply ReCBuLC to two recent systems on the x86/Linux platform, and show that it can significantly improve their performance.

Our contributions are as follows:

- We propose to use hardware per-core clocks to determine the global order of shared accesses among threads that allows concurrency bugs to be reproduced with substantially reduced overheads.
- We present a methodology to obtain a range of skews among per-core clocks. We then use a statistical scheme to narrow the range of clock skews to less than 10 ticks (10 cycles) with a high confidence.
- ReCBuLC is applied to two recent systems and shows that it can improve their efficiency significantly.

In the rest of the paper, Section 2 gives some background and motivation. Section 3 presents two schemes to calculate the skews among local clocks. Section 4 applies ReCBuLC to PRES and CLAP. Section 5 details our implementations of PRES and CLAP with ReCBuLC. Section 6 presents our experimental results. Section 7 gives the discussion about the limitation and the future work. Section 8 covers the related work, and Section 9 concludes this paper.

2 Background and Motivation

2.1 Local Clocks on Commercial Processors

Almost all mainstream commercial processors provide local per-core clocks. Applications can access them for needed timing information. For example, Intel/AMD x86 processors provide a 64-bit Time Stamp Counter (TSC) since the Pentium family. The TSC is incremented at a near constant rate with respect to the wall-clock time. It is not affected by the dynamic frequency scaling [7]. Similar mechanisms exist on other processors. IBM Power processors have a 64-bit Time Base register on each core [11]. Its counting frequency can be changed by software. If we record the frequencies before and after the change, we can convert the value of Time Base register to the wall clock time [11]. MIPS processors also have a similar Count Register [10], but its size is only 32-bit. SPARC processors have a 63-bit Tick register [12] to keep clock cycles.

2.2 The Time Stamp Counter on Intel x86 Processors

Although most manufacturers have their own unique designs of local clocks, the main feature is very similar. In this section, we mainly focus on the TSC of Intel x86 processors.

There are three generations of TSC on x86 processors: *Variant TSC*, *Constant TSC* and *Invariant TSC* (time order). *Variant TSC* is the first generation from a very old processor. Because its triggering frequency can be impacted by the CPU frequency, it is not widely used. The *Constant/Invariant TSC*s can operate at a constant rate in most processor states even when CPU frequency is changed. The only difference between them is that the *Constant TSC* can be changed (e.g., stopped) when the CPU is run on ACPI deep C-state transitions [7]. Both can be changed (i.e., re-initialized and stopped) in some ACPI deep S-states [32]. The frequency of the *Constant TSC* is set by the ratio of its maximum core clock rate and the bus clock rate of the processor [7]. The *Invariant TSC* is based on the invariant timekeeping hardware that runs at the core crystal clock frequency [7]. For cores on the same chip, their TSCs operate at the same frequency. For processors of the same type (i.e., in the same CPU family and having/with the same maximum core clock frequency) and on the same board, their TSCs should operate at the same frequency too. In this paper, we only consider *Constant/Invariant TSC*s.

With *Constant/Invariant TSC*s, all local cores reset TSCs to 0 when the processor is powered up. At the boot time, all processors that are connected with the same RESET signal will get reset. The RESET signal is guaranteed to arrive at each processor at the same time. However, even with such facilities we still cannot ensure that all TSCs are synchronized at all time for the following reasons: i) A new processor can be introduced using CPU hotplug, which may not have synchronized TSC value with those on existing CPUs; ii) Software or firmware can modify a TSC through the *wrmsr* instruction [7], e.g., some BIOS SMI handler may hide its execution by changing the TSC value [34] and breaks synchronization with other TSCs; iii) In [33], it is mentioned that the thermal effect could cause TSCs to drift during a reset. Moreover, the Intel manual cautions that it is impractical to synchronize all logical processors using software at any given time [7].

2.3 Challenges to Reproduce Concurrency Bugs Using Local Clocks

If there were "ideal" local clocks that had the same timestamp across all different cores at any time (like a global

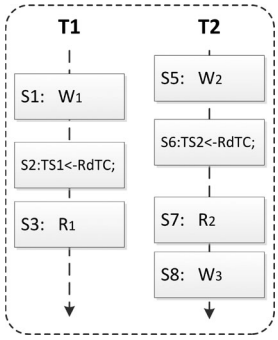


Fig. 1. Happen-before determined by local clock.

clock), each thread could locally record its own timestamps when accessing shared resources. The recorded timestamps could then be compared directly to determine their global order. We will need neither synchronization when they are being recorded, nor constraints solving when they are being reproduced. The overall efficiency can be significantly 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 the time stamps read from two 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 $S5 \prec S3$. Unfortunately, as mentioned in Section 2.2, we cannot derive such an easy conclusion because hardware cannot ensure that per-core clocks are synchronized at all time.

LReplay [19] expects that future processors will provide a global clock with a fast access time, which could dramatically reduce the runtime overhead and log size, as it only needs to record orders that cannot be inferred from the global clock. Most commercial processors allow local per-core clocks to be accessed in *user mode* while require the global clock to be accessed via a *system call* with a substantially higher overhead. For example, on Intel Xeon Phi, the overhead to access its global clock is in the order of ~1,600 cycles, while it only takes 6~10 cycles to access local per-core clock. Nevertheless, there are still significant challenges that need to be resolved in order to use the low-overhead per-core local clocks:

(1) We need to stabilize the frequency of the per-core local clocks. For example, the triggering frequency of *Constant/Invariant TSC*s could be changed in some ACPI states on x86 platforms, and the TSC value could be modified by firmware implicitly (mentioned in Section 2.2). Although the incremental frequency is the same before and after modifying the TSC value through the *wrmsr* instruction, the TSC value is not incremented linearly across all local clocks. Hence, in a sense, modifying the TSC value also affects the incremental frequency. Without the consistent incremental frequency, the local locks cannot be used to order shared accesses.

(2) Besides the consistent incremental frequency, the differences among different per-core clocks should be measured accurately. 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 per-core clocks. Therefore, to accurately measure these time differences and use them to order shared accesses posts another challenge.

(3) 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 give 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.

(4) We need to handle possible overflow of the clocks. Clocks on MIPS processors have 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.

The rest of this paper assumes the following environment: 1) For multi-cores, their local clocks count at the same frequency in most CPU states (e.g., ACPI P-state in x86 processors); 2) For multi-processors, all processors should be of the same type, use the same crystal oscillator and placed on the same mainboard. That is, their local clocks have the same incremental frequency. In such an environment, with the measured differences among these local clocks, we can use their values to determine the order of shared memory accesses.

3 Determining the Order by Local Clocks

When using the local clocks to reproduce concurrency bugs, almost all processors have the same challenges that are mentioned in Section 2. So to give our methods clearly, we mainly discuss how we addressed these challenges on Intel platforms. Since the clock overflow problem is more prominent on MIPS platforms that have only 32-bit clock, the discussion about this challenge is specific on MIPS platform. We believe that ReCBuLC supports all platforms.

3.1 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 its original order. An intuitive solution is to insert *FENCE* instructions before and after each *RdTC*, which is shown in Fig. 2a. This may seem to work, but it is much more complicated on today's multi-core processors.

In modern multi-core processors, the completion of a write operation can be divided into two phases: (1) Local Complete (LC), i.e., the data is written to the local write buffer, but is yet to be seen by other cores; (2) Globally Visible (GV), i.e., the written data is out of the local write buffer and is visible to all other cores through the cache coherence protocol. We use *W(LC)* and *W(GV)* to denote the time a write W is written to the local write buffer and the time W is globally visible to all other cores, respectively.

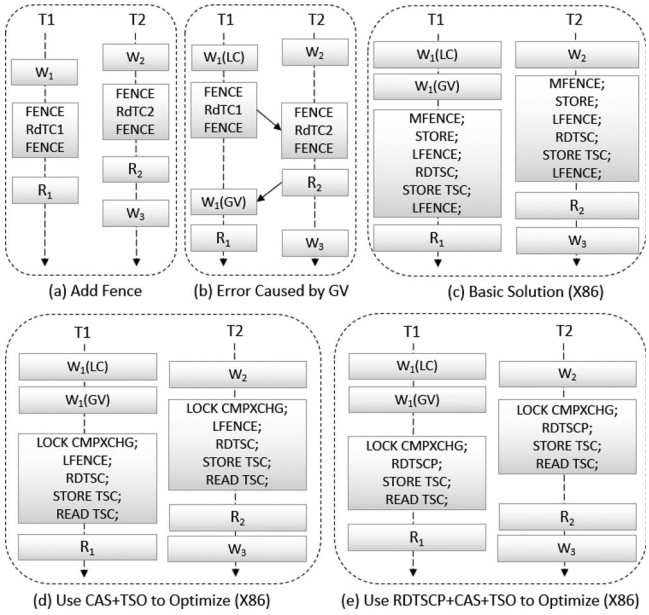


Fig. 2. TC order.

In Fig. 2b, a local *FENCE* only guarantees that *W1(LC)* $\prec$ *RdTC1*, but it cannot control *W1(GV)*. Based only on the value of the local clock, we could infer that *W1* $\prec$ *R2*, which may not the case. Therefore, a *FENCE* instruction must ensure that *RdTC* is not issued until all previous writes become GV. In Figs. 2c, 2d, and 2e, we present three solutions on Intel x86 platform to address the global visibility problem in Fig. 2b. These solutions can also be used on other platforms. On Intel x86, an *MFENCE* will hold *load*s and *store*s until all preceding *load*s and *store*s become globally visible, while an *LFENCE* will hold *all* instructions (not just *load*s and *store*s as in *MFENCE*) until all preceding instructions are locally complete.

A correct implementation on x86 is shown in Fig. 2c. The *STORE* in Fig. 2 C is any *store* instruction that has no relationship (e.g., data dependence) with other instructions. The *RDTSC* is the x86 instruction that reads timestamp counter. The *STORE TSC* stores the timestamp into the memory. The reason we put an *STORE* here is because *RDTSC* is not a regular memory access operation, *MFENCE* cannot guarantee that *W1(GV)* $\prec$ *RDTSC*. Placing an *STORE* will guarantee that *W1(GV)* $\prec$ *STORE*. Meanwhile, the first *LFENCE* will guarantee that *STORE* $\prec$ *RDTSC*, and *W1(GV)* $\prec$ *RDTSC* can be guaranteed. The last *LFENCE* is to guarantee that *RDTSC* $\prec$ *R1*. In this way, we can guarantee that a more precise local clock value be between *W1* and *R1*.

Although *MFENCE* and *LFENCE* instructions can guarantee the order of memory instructions, they also incur very high overhead, e.g., three such instructions are needed in Fig. 2c. To avoid excessive use of such instructions, we can instead use the CAS (Compare and Swap) instruction and the processor-specific features of its *memory consistency model*. The CAS instruction (e.g., the *cmpxchg* instruction on x86) is executed atomically during which the bus and the cache are locked until its completion. To improve the performance, modern processors often adopt a weaker *memory consistency model*. For example, x86 adopts a weaker Total Store Order (TSO) model [36]. In the TSO model, among the four possible Read and Write orders, i.e., Read→Write, Read→Read, Write→Write, and Write→Read, the processor will not guarantee the order of Write→Read if they access different memory locations. It allows Read to bypass the write buffer and execute before the Write if they access different memory locations.

Taking advantage of the CAS instruction and the TSO model, we only need to use one *LFENCE* instruction as shown in Fig. 2d. Before the execution of a CAS instruction, all Writes in the local write buffer must be made globally visible. In addition, the *LFENCE* instruction guarantees that *RDTSC* waits until the CAS instruction is completed. *READ TSC* (a regular *load* instruction) will read the value of the local clock from the memory location written by the *STORE TSC*. Because the last three instructions have data dependences, we can guarantee that *RDTSC* $\prec$ *STORE TSC* $\prec$ *READ TSC*. Based on the TSO model, all memory access instructions after the *READ TSC* cannot be executed before it. So, we can guarantee that *RDTSC* $\prec$ all following memory access instructions (including *R1*). Moreover, *READ TSC* will not incur a cache miss that improves the performance even further.

On some newer x86 processors, they provide another instruction *RDTSCP* to read the timestamp counter [7]. Besides the TSC value, it also provides the serial number of the logical core. It waits for all previous instructions to be completed locally before reading the TSC value [7]. With the help of this instruction, we can eliminate the *LFENCE* instruction in Fig. 2d. But *RDTSCP* cannot guarantee the preceding instructions are scheduled before it. Similar to *RDTSC*, *RDTSCP* is not a regular *load* instructions. Hence, a *READ TSC* still needs to be placed here to guarantee that *RDTSCP* $\prec$ *R1*. The final optimized code sequence is shown in Fig. 2e.

In Section 6.3, we provide some experimental results to compare the two implementations shown in Figs. 2d and 2e.

3.2 Stabilizing the Triggering Frequency of Per-Core Clocks

As mentioned in Section 2.2, there are two scenarios that can affect the frequency of a local clock: 1) CPU runs into some special states (e.g., ACPI C- or S- states on x86); 2) the software or the firmware modifies the TSC value. We adopt two different measures to handle these two situations.

(Scenario 1) Use a kernel module to control the CPU states. For example, for the x86 CPU with *Constant TSC*, the kernel module keeps the CPU from running into the ACPI deep C- and S- states during the record and replay. For the x86 CPU with *Invariant TSC*, the kernel module keeps the CPU from running into the ACPI deep S-states.

(Scenario 2) Measure the difference among per-core clocks before and after the record and replay. During the record and replay, other software or firmware could also run on the core (e.g., for process scheduling). They may modify the local clock while we cannot prevent such operations. Our strategy is to detect whether such operations actually occurred during the record and replay. To do so, we measure the difference among per-core clocks twice: before and after the record and replay. We compare their differences. If the differences are small, we consider the local clocks not modified. Otherwise, we regard the local clocks have been modified, and we need to repeat the

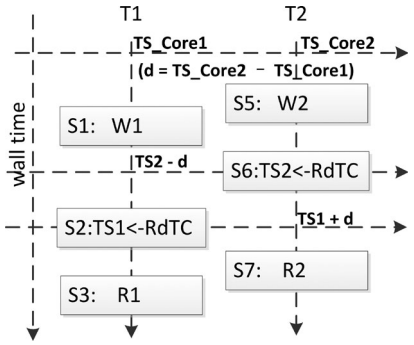


Fig. 3. Determine orders by local clock.

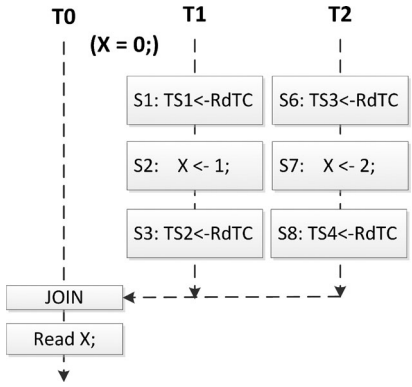


Fig. 4. Local clock difference tester.

record-and-replay. In fact, we have not encountered such a situation during all our experiments. But we still present an experiment in Section 6.5 that simulates the modification of the local clocks (speeding up one of the clocks) to show how much impact it can have.

3.3 Handling the Time Differences among Per-Core Clocks

Although per-core clock values among cores could vary at any time, we can still make use of them if we know their differences (called **d**). An example is shown in Fig. 3. Assume that the values of two local clocks are *TS_Core1* and *TS_Core2*, respectively. Then, **d** is *TS_Core2* - *TS_Core1*. $TS2 < TS1 + \mathbf{d}$ means $S6 \prec S2$ (i.e., $RdTC2 \prec RdTC1$). We can infer that $S5 \prec S3$. However, it is very difficult to determine the value of **d** precisely as mentioned in Section 2. 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 $\mathbf{d} \in$ [d1, d2], i.e., **d** is in the range of d1 and d2. If $TS2$ - d1 $< TS1$, we have $TS2 - \mathbf{d} < TS2$ - d1 $< TS1$, and this means $S6 \prec S2$. We can thus infer $S5 \prec S3$. Similarly if $TS1$ + d2 $< TS2$, we can infer $S1 \prec S7$. For other cases, their orders cannot be determined. Although the range of **d** is not as good as a precise **d**, it is still possible to determine their order if the range is small enough.

On processors on with we cannot obtain the value of **d** precisely, we propose two schemes to get a range of **d**:

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

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

3.3.1 Scheme 1-Use Test Programs

We designed a small test program shown in Fig. 4. The order of *RdTC* and other instructions is guaranteed. The fence instructions are not included for clarity. Threads T1 and T2 are bound to two cores on which **d** is measured. Each thread writes a different value to the shared 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 local clock for *TS1*, the value of the local core is $TS1 + \mathbf{d}$. Therefore, we have $TS1 + \mathbf{d} < TS4$, that is

$$\mathbf{d} < TS4 - TS1 \quad (if\ Read\ X\ returns\ 2\ in\ T0). \tag{1}$$

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

$$\mathbf{d} > TS3 - TS2 \quad (if\ Read\ X\ returns\ 1\ in\ T0). \tag{2}$$

We repeat the above process and obtain as many pairs of $\langle TS4_i, TS1_i \rangle$ and $\langle TS3_i, TS2_i \rangle$ as possible. According to the argument above, the value of **d** is less than any $TS4_i - TS1_i$, and greater than any of $TS3_i - TS2_i$. That is

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

As mentioned in Section 3.1, to ensure the execution order of the above instructions, we have to add some fences or similar instructions in the testing program. We designed four implementations for x86 platforms.

In Fig. 5a, we use the sequence of instructions sequence introduced in Fig. 2c, while in Fig. 5b, we use the *CPUID* instruction instead. *CPUID* instruction is a serializing instruction that forces the processor to complete all modifications to flags, registers, and the memory by earlier instructions, and drain all buffered writes to the memory before the next instruction is fetched and executed [7]. In Fig. 5c, we make use of the atomic instruction *XCHG*. This implementation does not guarantee that the GV of writing *X* happens before *RDTSC*, so we need to check whether it does. Fig. 5d is similar to that in Fig. 5c except it uses *CMPXCHG* instruction instead. Figs. 5c and 5d may generate a smaller range for **d** because none of the time-consuming *MFENCE* or

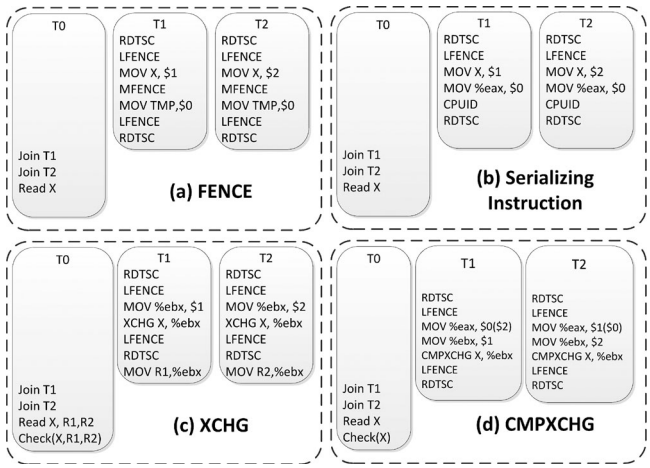


Fig. 5. Difference tester implementation.

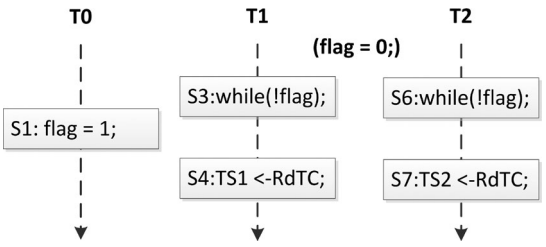


Fig. 6. Statistic tester.

CPUID instructions is used. The efficacy of these codes depends on the hardware implementation, but we can always run all of them many times to obtain a minimum range of **d**. In Section 6.4.1, we present an empirical study on using this scheme to calculate the range of **d**.

3.3.2 Scheme 2-Use Statistical Tests

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 range.

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

(1) The time needed for RdTC and fence instructions.
(2) The time needed to flush the write buffer.
(3) The time needed for 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). They are bound to 3 different cores. The initial value of *flag* is 0, so T1 and T2 will spin on *flag* (see Fig. 6). After thread T0 writes 1 to *flag*, T1 and T2 will break from the *while* loop and read the local clock, hopefully about the same time. The difference of their readings (*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 reasons:

(*R1*) The *while* loop contains at least 3 instructions: *load*, *compare* and *branch*. When T0 sets *flag* to 1, T1 and T2 may not execute the same instruction and will not exit the *while* loop at the same time.

(*R2*) The cache coherence protocol will serialize the worker threads. In most modern processors, each core has private L1 and L2 caches. The processor uses a coherence protocol (e.g., MESIF on Intel x86) to maintain data coherence among cache memories. When two cores simultaneously access the same cache line due to cache misses, they obtain the data serially [13]. 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 sets *flag* to 1, T1 and T2 may not know it at the same time.

(*R3*) Scheduling and interruptions may occur between the *while* loop and *RdTC*.

(*R4*) When T1 and T2 exit the *while* loop, I-Cache 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 avoid the case of (*R4*). To avoid the case of (*R3*), we need to prevent the kernel to schedule other threads to the cores that T1 and T2 are running by loading a kernel module. If an interruption occurs during the execution of the test program, it will 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 but different cores, and T1 is bound to a different processor. T2 will get the new value of *flag* sooner than T1 because it is closer to T0. If we want to calculate the **d** of the two cores on the same processor, T1 and T2 need to be bound to two cores on the same processor. Otherwise, T0 can be randomly bound to either of the two processors in each run.

To describe the effect of (*R1*), we use ε to represent that the time lag between T1 and T2 when they execute the *load* instruction to get the new value of *flag*. The value of ε is a positive number when T1 executes the *load* instruction sooner. Otherwise, the value is a negative number. But the absolute value of ε is less than the total cycles of executing the *load*, *compare* and *branch* instructions. To describe the effect of (*R2*), we use I to represent the time lag between core1 (T1 runs on it) and core2 (T2 runs on it) to get the new value of *flag* when core0 (T0 runs on it) sets the flag. The value of I is a positive number. We also use δ to represent whether core1 (T1 runs on it) gets the new value of *flag* sooner than core2 or not. The value of δ is either -1 or 1. We run the test program multiple times. Assume $<TS1_i, TS2_i>$ is the timestamp pair of the i-th run, we have $TS2_i = TS1_i + d + \varepsilon_i + \delta_i I_i$, or

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

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

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

After the test program runs many times, we have

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

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 will be different. The effects of (*R1*) on T1 and T2 are the same, implying that the expectation value of ε is 0. If the test program runs a sufficiently large number of 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 (*R2*) 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) can be converted to

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

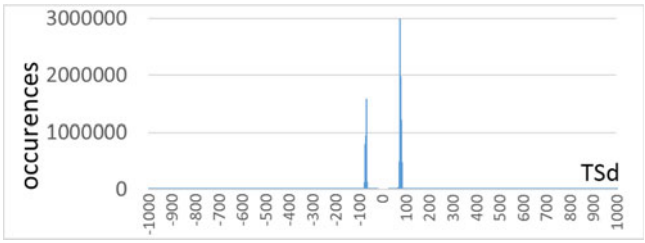


Fig. 7. A distribution of TSd.

By Wiener-Khinchin theorem for large numbers [35], 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 Equation (7), 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, whose absolute value is 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 middle of the two spikes in Fig. 7 as the boundary. If the value of TSd is on the right 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**). We still need to calculate the confidence interval of D. According to the central-limit theorem, the values of D is close to a normal distribution, that is $D \sim N(\mu, \sigma^2)$. The expectation value of this distribution is the approximate 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

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

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

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

An experimental analysis of using this scheme is given in Section 6.4.2. In this experiment, we calculate the confidence interval of **d** under different confidence coefficients.

3.4 Local Clock Overflow

As mentioned earlier, the clocks on most processors (except SPARC and MIPS) have 64 bits, which will take more than ten years to overflow with the current clock rates. The 63-bit SPARC clock also takes several years. It is enough for most applications. But, for a 32-bit MIPS clock, an overflow can occur every few seconds. Therefore, overflow must be considered and handled.

Assume the overflow period of a clock is **P**, we must ensure that the interval between two adjacent records is less than **P**. To do so, 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 has overflowed; if $TSC_{n+1} - TSC_n > 0$, it has not.

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

However, since the MIPS clock overflows every few seconds, an interruption or task scheduling may make the interval between two records larger than **P**. In practice, the time used to handle an interruption is short in most cases (in milliseconds), but task rescheduling will affect the accuracy if dozens of threads need to be scheduled on the same core. In our measurements, we did not find any two adjacent records whose interval is more than 1 second. Using a kernel module to record the wall clock time of the task scheduling and interruptions can also resolve this problem properly.

4 Reproducing Bugs Using Local Clocks

In this section, we select two well-known bug reproducing systems PRES [18] and CLAP [6], and show how to apply our approach to them. As mentioned in Section 1, we select them because PRES relies on an expensive scheme to record the global order of some special events. CLAP depends on sophisticated offline analysis to compute the buggy interleaving albeit with very low recording overhead. They represent the key dilemma of such schemes: either incurring large recording overhead or spending long analysis and replay time.

For PRES, its bottleneck is the recording phase. We record the timestamps using local clock instead of the expensive global order, and infer the global order of those special points as described in Section 3. Our experiments show that without recording global order, the overhead can be reduced by up to 85.24 percent.

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 global order by an efficient offline analysis. These key points can be selected at function call sites or entry/exit points of loops. We combine the inferred global order and the original constraints as new inputs to the SMT solver. Our experiments show that, for most benchmarks, more than 95 percent of shared accesses can be ordered.

For the remaining unordered shared accesses, we can further reduce the solving complexity with the help of local timestamps. Assume the memory operations in Fig. 8 access the same shared variable. In Fig. 8a, 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. 8b, if we could know $RdTC3 \prec RdTC1$ and $RdTC2 \prec RdTC4$ from local timestamps, we only need to infer the order of R_{11}, W_{21} and W_{22}. This could reduce a lot of constraints and achieve less solving time.

In addition, 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

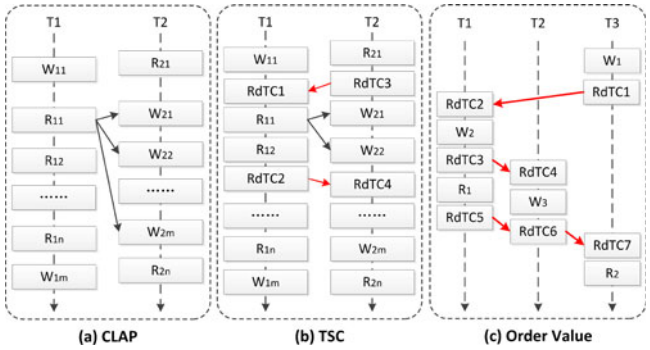


Fig. 8. Constraints reduction.

numbers and shorten the solving time. In Fig. 8c, the global order numbers of the five shared accesses are all within the interval [1,5]. If from the local timestamps, we know $RdTC1 \prec RdTC2 \prec RdTC3 \prec RdTC4 \prec RdTC5 \prec RdTC6 \prec RdTC7$, we can infer that $W1 \prec W2 \prec R1/W3 \prec R2$. It can reduce the range of their global order numbers to [1,1], [2,2], [3,4], [3,4], and [5,5], respectively.

5 Implementation Details

According to the schemes described in PRES [18] and CLAP [6], we implemented two systems, called PRES-impl and CLAP-impl, on the Linux/X86_32 platform. We believe our method can be applied to other platforms that have the support of the local clocks. We then apply ReCBuLC to these two systems, called PRES-tc and CLAP-tc, respectively. To apply our approach to PRES and CLAP, we need to bind each thread to a different core on the same processor during the record phase. We then use the technique described in Section 3.3 to calculate the range of **d** in advance. The aim of binding threads to cores is just to simplify two aspects of the works: 1) the recording of the TSC values; 2) the ordering of TSC values. If we choose not to bind threads, we may need to record the thread migration when a thread is scheduled to run on a different logical core. Besides the recording changes, the ordering job becomes more complicated. This is because the recording TSC values of each thread could come from many logical cores. But we should note that the binding method disturbs the normal cpu scheduling of the operating system, so it could introduce the extra overhead during the record phase.

In this section, we mainly focus on the implementation difference between the original systems and our re-implemented systems (i.e., PRES-impl and CLAP-impl).

5.1 The Implementation of PRES-impl and PRES-tc

The original PRES is implemented using Intel's general dynamic instrumentation tool, Pin [9], for both recording and replay. Different from the original implementation, we use our specialized static binary instrumentation tool for a better performance. The tool is implemented as a shared library on the Linux platform. It is loaded into the tested program's process using the environment variable *LD_PRELOAD*. In the constructor function of this shared library, it first finds all of the code in the current process by parsing the */proc/self/maps* file. It then disassembles the code, recognizes basic blocks and recognizes all functions. It needs the compiler to provide more precise information (e.g., need '-g' option in gcc) to recognize functions when compiling the tested programs. It then allocates the code cache area and places the instrumented code there. An important design is to make sure that all code pointers are not mutated, i.e., they should all point to the original targets. For example, the return addresses that are pushed by the call instructions onto the stack should point to the original return targets. To guarantee the correctness, it dynamically redirects the code pointers to their respective addresses in the code cache when they are de-referenced. This is implemented by instrumenting all indirect branch instructions (i.e., call returns, indirect calls and indirect jump instructions) to perform runtime address translation. Meanwhile, it also relocates the direct branch instructions (e.g., jcc's, direct function calls and direct jumps) when generating code into the code cache. Moreover, the signal handlers are registered at the locations in the code cache instead of the original code area. Using our instrumentation framework, most binary features can be supported (e.g., the *self-referencing code*[1]).

Based on our static instrumentation tool, we can perform instrumentation at any given point when generating code into the code cache. According to the scheme described in PRES, PRES-impl records the global order (implemented by using the spin locks) among synchronization points (SYNC), function calls (FUNC), basic blocks (BB), and memory operations (RW) during the online recording phase. Instead of recording the global order, PRES-tc only records the local timestamps. In the offline exploration phase (i.e., replay phase), PRES-impl and PRES-tc also use this static instrumentation tool. They also use the same synchronization method (implemented by using the spin locks) to schedule the threads.

5.2 The Implementation of CLAP-impl and CLAP-tc

The original CLAP is implemented based on LLVM and KLEE. It instruments the source code to record the path information. When running the tested program, it collects all of the threads' local paths. Then, CLAP uses KLEE offline to performs symbolic execution along the paths to collect and encode all of the necessary execution constraints (e.g., the path constraints, the bug manifestation constraints and the read-write constraints) over the order of the shared access points. Different from the original implementation, CLAP-impl uses the static instrumentation tool mentioned above to collect each thread's local paths by instrumenting all of the branch instructions and recording the jump targets. In the offline analysis phase, we use the dynamic binary translation technique to interpret the instructions by following the profiled paths and perform the symbolic execution, which is similar to the concolic execution [4], but the symbolic variables are the values of read/write accesses to the shared data and the orders of the shared accesses. This interpreter is implemented as a shared library. We use the environment variable *LD_PRELOAD* to intercept the tested program's *libc_start_main* routine (within which the *main* function is called). In the intercepted routine, the interpreter starts at the entry of the *libc_start_main* by following the

1. Self-referencing code usually treat the code pointers as data pointers and use these data pointers to read the content. For example, the libunwind library [8] uses the return address to read its own code and checks whether the instructions are PLT encoded or not.

TABLE 1
Platform Details

CPU	Intel Xeon E7-4807, 6 cores, 1.87 GHz
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 [28]

TABLE 2
Benchmarks

TYPE	BENCHMARKS	DESCRIPTION	BUG TYPES
Server	Apache HTTPD [30]	Web server	AV
	Cherokee [31]	Web server	AV
Desktop application	PBzip2	Compressor	OV
	Pfscan	File scanner	AV
	Aget	HTTP/FTP downloader	AV
Scientific application (SPLASH-2) [29]	Barnes	Barnes N-Body algorithm	OV
	LU	LU matrix multiplication	OV
	Radiosity	Graphics rendering	OV

profiled path. During the symbolic execution, we encounter similar implementation challenges as in CLAP, such as shared memory access identification and symbolic address resolution. We use similar approaches to address these challenges. For example, in symbolic address resolution, we first analyze the base address of the given symbolic address, and then find the target object by searching the DWARF information that is generated by the compiler when compiling the source code with the '-g' option [5]. For any read or write to this object, the loaded or stored value is resolved from each element in this object with a set of constraints. Finally, we merge all these constraints and use the SMT solver to solve them.

There are two main differences in CLAP-tc compared to CLAP-impl: 1) CLAP-tc instruments the code to record the local timestamps at runtime; 2) CLAP-tc performs the constraints reduction by using these local clocks (discussed in Section 4) in the offline analysis phase. To balance the recording overhead and the effectiveness of the constraints reduction, we only instrument to record the local timestamps at the FUNC, LOOP and FUNCLOOP levels instead of the Basic Block level. In FUNC, the recording is done at the entries and exits of functions. In LOOP, the recording is done at the loop entries, exits and back edges. In FUNCLOOP, it is a combination of FUNC and LOOP.

6 Experiments

In this section, we evaluate the performances of PRES-impl/PRES-tc and CLAP-impl/CLAP-tc. Table 1 shows a summary of the platform used. We select several bugs in real multi-threaded programs (Table 2) that include some widely-used applications on servers and desktop, and also some scientific programs. They 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. In the experiments, the performance of Apache and Cherokee is measured by their throughput, and the others are by the execution time. System library routines rarely access shared variables and their accesses can be inferred from their arguments easily, so we do not consider them. In Sections 6.1 and 6.2, we use the *basic solution* in Fig. 2c to record the local clocks. And in Section 6.3, we evaluate the effectiveness of the two optimizations shown in Figs. 2d and 2e.

6.1 Evaluating PRES-impl and PRES-tc

PRES-impl records the global order of certain operations, while PRES-tc records their local timestamps. Fig. 9 shows the normalized execution time of PRES-impl to PRES-tc instrumented at the synchronization point, function, basic-block, and memory operation level. The baseline is the native execution time.

PRES [18] can reproduce all of the bugs at the FUNC level within 1,000 tries. At the BB level, PRES reproduces all of the bugs within 10 tries. Taking recording overhead and the number of replays needed into consideration, instrumentation at these two levels seems reasonable for PRES-impl. PRES-tc reduces the recording overhead from 320.63 percent in PRES-impl to 133.48 percent at the FUNC level on average. At the BB level, the recording overhead is reduced from 1730.05 to 688.34 percent.

The main reason for the improvement is that PRES-tc avoids synchronization and allows each thread to record local timestamps concurrently. Take LU as an example, 56.49 and 64.53 percent of the recordings in PRES-tc are done concurrently at the FUNC and the BB levels, respectively, and thus 62.44 and 69.24 percent of the recording overheads are reduced.

At the SYNC level, the overheads of the two systems are similar. This is because the number of synchronization

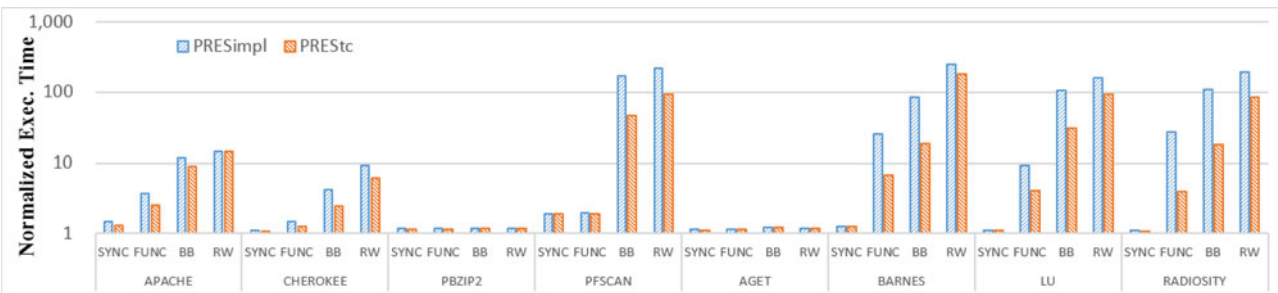


Fig. 9. Normalized exec. Time of PRES-impl/PRES-tc.

TABLE 3
Reproducing Tries

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

Add_UO means the additional unordered accesses.

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 are almost the same. The reason is that the main workload of PBZIP2 and AGET is compressing and downloading data 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 incur a small amount of unordered accesses at the recording points. Table 3 shows the percent of them to the total 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 3, respectively. We can see from these data that the percent of the unordered accesses is less than 1 percent at BB, FUNC, and SYNC levels, which is a very small percentage of all accesses. Besides LU at SYNC level and AGET at BB level, we can 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, they are caused by only a handful of shared accesses [1]. For LU at RW level, although there are 19.35%∼25.50% unordered shared memory 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. But for LU at SYNC level and AGET at BB level, we tried more replay attempts in PRES-tc than in PRES-impl. This is because the unordered access affected the successful replay in these two benchmarks. As we only keep two places of decimal digits after the decimal points for additional unordered accesses (Add_UO) in the table, it appears as if there is no unordered access. But unordered accesses did exist.

Fig. 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 percent of the recordings are done concurrently when there are 4, 8, and 16 threads, respectively. With more threads, a proportionately higher percentage of the recording time will be done concurrently.

6.2 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 [24] that contains massive data races and is very likely to produce a different result when the interleaving is

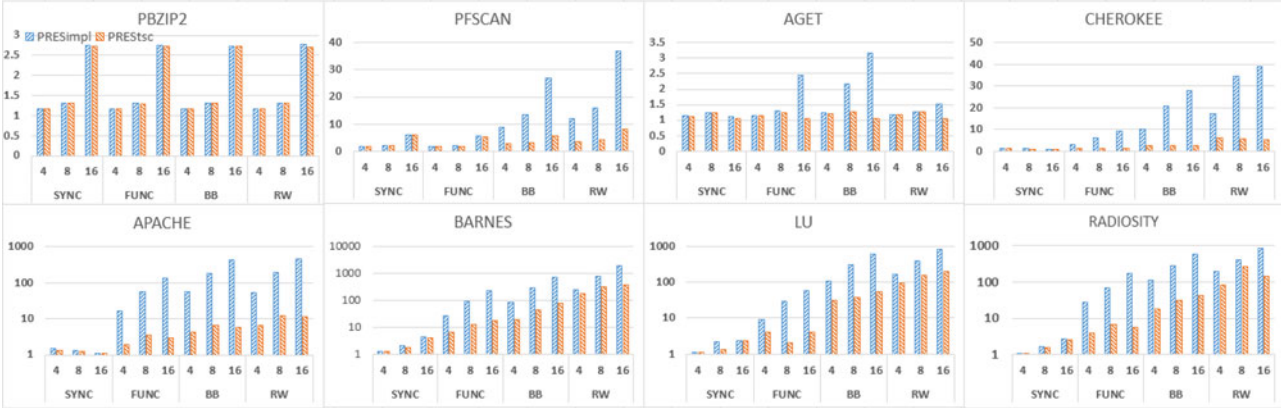


Fig. 10. Scalability of PRES-impl/PRES-tc. The y-axis is normalized exec. time, and in the 4 lower sub graphs are logarithmic.

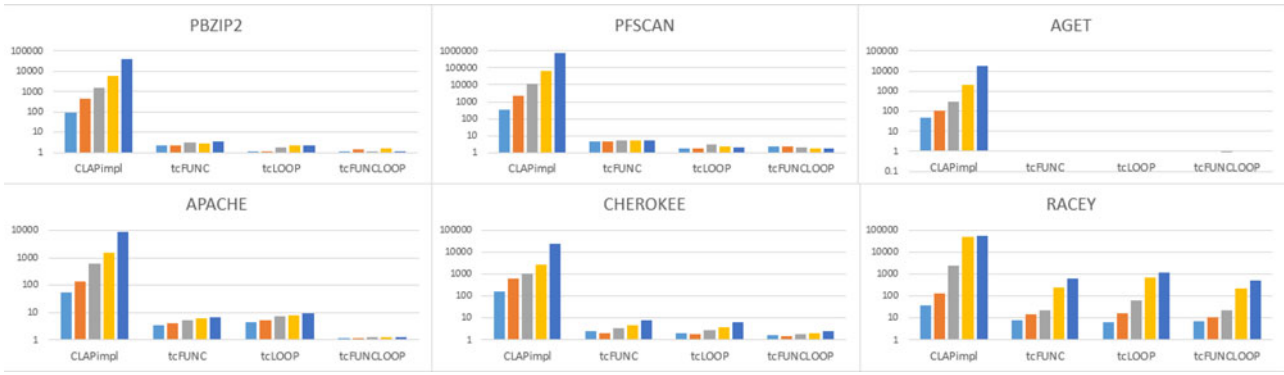


Fig. 11. Normalized solving time of CLAP-impl/CLAP-tc. Each benchmark is evaluated with 5 inputs.

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* (see Section 3.3).

Fig. 12 shows the recording overhead of CLAP-impl and CLAP-tc at different instrumentation levels. FUNC, LOOP and FUNCLOOP represent the different instrumentation levels. In Fig. 12, we can see that the slowdown caused by recording in CLAP-tc is 101%∼142% of CLAP-impl, and mostly less than 110 percent.

Fig. 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 replays, 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 be 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.

Fig. 11 shows that, compared to CLAP-impl, CLAP-tc reduces solving time by 84.66 percent ∼99.99 percent. 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 percent 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.

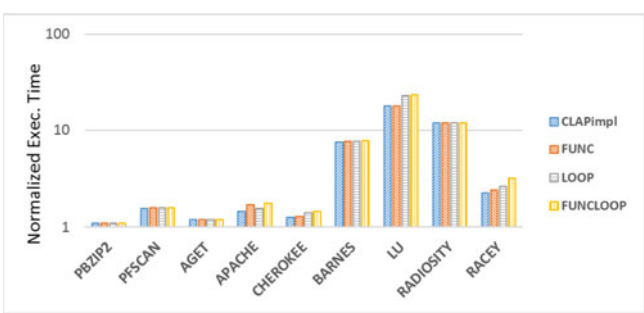


Fig. 12. Recording overhead of CLAP-impl/CLAP-tc. The overhead is normalized to the non-instrumented programs.

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 for all benchmarks.

In studying Figs. 11 and 12, we can see that the lower the instrumentation level is, the less solving time but the more overhead is observed. 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.

To show the effectiveness of ReCBuLC in reducing the solving time, we give overall results of CLAP-impl and CLAP-tc at the FUNCLOOP level in Table 4. There are two types of unknown variables in CLAP: one is the values returned by read accesses to the shared variables and the other is the order of those accesses to the shared variables in the to-be-computed schedule [6]. Due to the non-determinism in multi-threaded programs, there could be a lot of buggy executions to trigger the same bug. CLAP-impl and CLAP-tc may collect all buggy executions in the online phase that could affect the number of shared data accesses (#SDAs) and the number of unknown variables (#UVs). From the table, we can see that besides Racey, the results of #SDAs and #UV are different between CLAP-impl and CLAP-tc. This is because the number and the order of shared accesses in each thread execution are deterministic in Racey. We also give the results of the size of constraints (#Constraints) in CLAP-tc without using the local timestamps to perform reduction, i.e., CLAP-impl and CLAP-tc use the same method to construct constraints. The result shows that #Constraints in CLAP-impl and un-reduced #Constraints in CLAP-tc are different, but they are still in the same order of magnitude that are not enough to affect the significant change in the solving time in Fig. 11. The main reason is the significant constraint reduction from using the local timestamps. To show the effectiveness of the constraints reduction, we give the results of constraint reduction in CLAP-tc. We can see that besides Racey, CLAP-tc achieves very good constraint reduction in the remaining benchmarks (reducing more than 93 percent constraints). Fig. 11 also shows that CLAP-tc requires

TABLE 4
Overall Results in CLAP-impl and CLAP-tc with the Largest Input in Fig. 11

Benchmarks	#Ts	#SVs	Results in CLAP-impl			Results in CLAP-tc with the FUNCLOOP instrumentation level				
			#SDAs	#UVs	#Constraints	#SDAs	#UVs	#Constraints		
								BeforeReduction	AfterReduction	Reduction(%)
PBzip2	4	18	86	117	7,186	91	121	7,315	426	94.18%
AGET	4	30	2,159	2,731	1,812,455	2,203	2,621	1,810,845	22,414	98.76%
PFSCAN	3	13	3,357	4,046	8,461,860	3,581	4,247	8,624,384	550,236	93.62%
CHEROKEE	8	16	75,572	14,211	10,285,234	75,194	13,686	10,036,745	151,527	98.49%
APACHE	16	22	98,243	19,376	12,314,847	99,767	20,253	12,857,583	76,291	99.41%
RACEY	3	3	1,046,706	1,589,379	324,429,525	1,046,706	1,589,379	324,429,525	89,088,347	72.54%

Column 2 reports the number of threads (#Ts). Column 3 reports the number of shared variables (#SVs). Column 4-6 report the results in CLAP-impl - Column 4 reports the number of shared data accesses in the schedule (#SDAs). Column 5 reports the number of unknown variables(#UVs). Column 6 reports the size of the constraints (#Constraints). Columns 7-11 report the results in CLAP-tc with the FUNCLOOP instrumentation level - Column 9 reports the size of the original constraints that are not handled by using the local timestamps (BeforeReduction). Column 10 reports the size of the remaining constraints after reduction (After-Reduction). Column 11 reports how many constraints are reduced (Reduction).

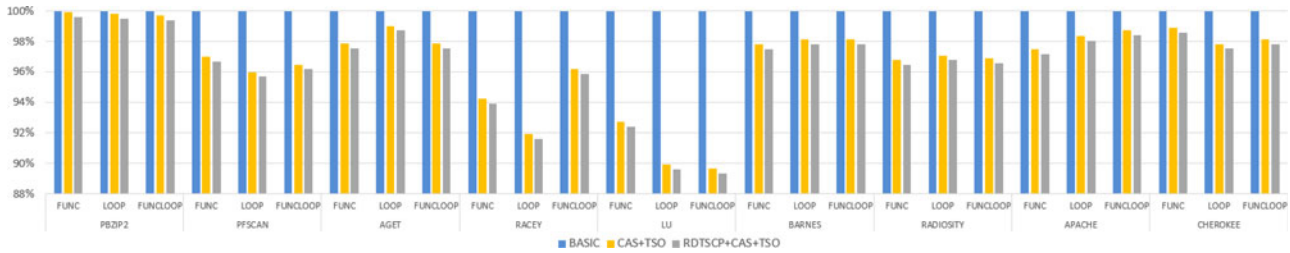


Fig. 13. Recording overhead of different implementation in CLAP-tc.

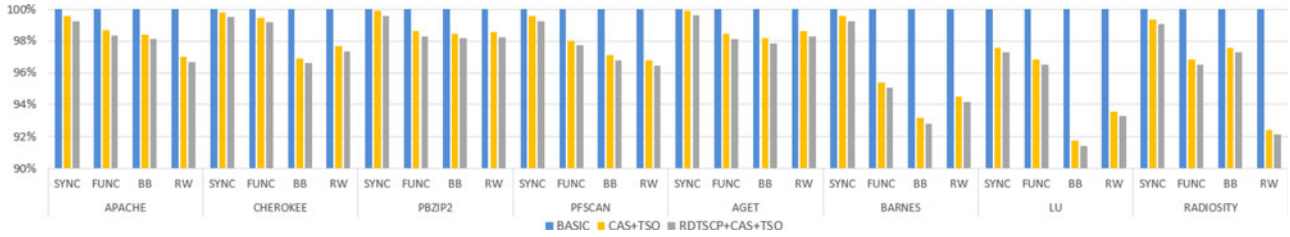


Fig. 14. Recording overhead of different implementation in PRES-tc.

much less solving time on these benchmarks, i.e., another indication on the effectiveness of its constraint reduction. In Racey, most addresses of the read and the 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, which also shows the effectiveness of using the local timestamps.

6.3 Evaluating Different Implementations of Recording Local Timestamps

Fig. 14 shows the recording overhead of different implementations in PRES-tc with 16 threads against the *Basic Solution* in Fig. 2c. Fig. 13 shows the recording overhead of different implementations in CLAP-tc against the Basic Solution in Fig. 2c. BASIC shows the *Basic Solution* in Fig. 2c. CAS+TSO shows the optimization that uses the CAS instruction with a TSO model in Fig. 2d. RDTSCP+CAS +TSO shows the optimization that uses the *RDTSCP* instruction to eliminate the *FENCE* instruction in Fig. 2e. As shown in Figs. 14 and 13, we can see that the optimization of CAS+TSO can reduce the recording overhead significantly. It is because the overhead of *FENCE* instructions is quite high and reducing them can improve the performance significantly. Although RDTSCP+CAS+TSO can reduce the overhead further, the gain is small. This is because the time difference between the *RDTSCP* instruction and the (*LFENCE*; *RDTSC*) pair is small. Both methods ensure that all prior instructions have completed. The subsequent instructions can be executed ahead of *RDTSCP*, but it is not allowed in the *LFENCE* instruction.

The optimizations of recording local timestamps not only can reduce the recording overhead in the online phase, but also can affect the offline analysis. Using these optimizations, it can introduce more unordered accesses than in the *Basic Solution*, especially, when using the finer-grained instrumentation to record the local timestamps. This is because the optimizations can reduce the execution time and increase the amount of parallelism. More specifically, it allows more recorded local timestamps from all threads in unit time and more pairs of values with differences within the range **d**. The more such pairs of values, the more unordered accesses we will get. Our experimental results substantiate such observations. Compared with the basic PRES-tc, the optimized

TABLE 5
Program Testing Results

Testing Program	1st Test		2nd 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

The bold values are the results of choosing a smaller range of **d**.

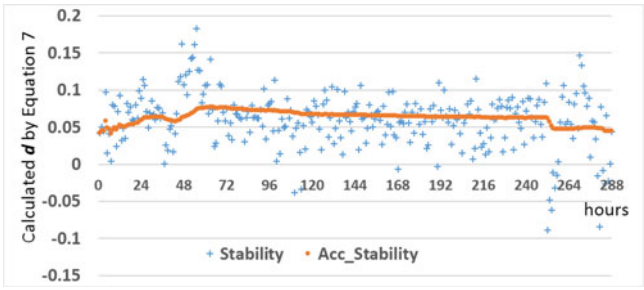


Fig. 15. Stability of Equation (7).

PRES-tc_S/PRES-tc_P didn't require more reproducing tries for all programs at all instrumentation levels. But, they introduced more unordered accesses. For all benchmarks instrumented at SYNC, FUNC and BB levels, there is not much difference in the number of unordered accesses between the basic and the optimized solutions. But for PFSCAN, LU and RADIOSITY at the RW level, the optimized (RDTSCP+CAS+TSO) PRES-tc_S/PRES-tc_P introduce more additional unordered accesses that are 3.62/5.70 percent, 23.67/28.59 percent and 5.37/7.08 percent, respectively. For the experiments using CLAP-tc, the optimized CLAP-tc didn't introduce more additional unordered accesses due to the coarse-grained instrumentations. Compared with the basic CLAP-tc, the optimized CLAP-tc has very similar results in constraints reduction and the solving time.

6.4 Differences of Local Clocks Among Cores

This section shows the results of our two schemes to calculate the range of **d**.

6.4.1 Program Testing Scheme

We designed four programs to test the ranges of **d**. Table 5 shows two of the 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 5. More test runs could generate smaller ranges. On our test platform, the test program in Fig. 5b 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 5, the range of **d** is about 200 cycles. Given two values of local clock on different cores, if their difference is larger than 200, we can ensure that its value is smaller in the real (wall) clock. For example, we can determine the order of *S6* and *S2* in Fig. 3. And then we can use it to determine the order of shared memory operations. If it is smaller than 200, we cannot give this confirmation. Using it to order shared access will not bring false positives or false negatives.

6.4.2 Statistics Scheme

Our proposed statistical scheme uses the statistical tester and Equation (7) to calculate the range of **d**. To use Equation (7), we need to know the value of δ_i, and the test procedure is as follows:

(1) Bind the worker and the trigger threads in Fig. 6 according to Section 3.3.
(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 3.3 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. 15.

In this figure, we calculate **d** every hour, using about 360,000 runs of the statistical tester. In Fig. 15, *Stability* marks the value of **d** that is calculated using the data collected in each hour, while *Acc_Stability* marks the value of **d** that is calculated using the data from the beginning of the run. 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 range of [-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 confidence interval of **d** under different confidence coefficients using Equation (9). The confidence interval requires many samples of **d**. We calculate **d** using the method described in Section 3.3 many times, and get $d_1; d_2; d_3; \ldots; d_M$. Each d_i is the result of N runs of the program shown in Fig. 6, and is calculated by using Equation (7) in which N is equal to the sum of r_1 and r_2. We use these M samples (i.e., $d_1; d_2; d_3; \ldots; d_M$) to calculate the confidence interval by using Equation (9). Finally, we get the data shown in Table 6.

In Table 6, 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 6 shows that when the confidence coefficient is 0.99999, N is 20 and M is 5. The range of the confidence interval is about 100, which is still smaller than the range obtained by program testing.

6.5 Detecting the Modification of Local Clocks

In Section 3.2 (Scenario 2), we propose to measure and compare the difference among per-core clocks twice: before and after the testing, and use the compared result to judge whether the TSC values have been modified or not by the software or the firmware. Although we have not encountered such modification in our experiments, we still give an experiment that simulates the modification of the TSC values and shows the difference between the results (the range **d**) before and after the TSC change.

TABLE 6
Confidence Intervals

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]

The first column is the confidence coefficient. The first row is the value of N, and the second row is the value of M. M means computing the value of **d** *using M sample values, each obtained by averaging N runs of the program in Fig. 6.*

To simulate the modification, we use Intel's support of timestamp counter adjustment: 1) Software can reset the TSC value of a logical core by using the *wrmsr* instruction to write to the IA32_TIME_STAMP_COUNTER MSR (we call it the *BASE* value); 2) It can also add or substract an offset to the TSC value to slow down or speed up the counter by using *wrmsr* instruction to write to the IA32_TSC_ADJUST MSR (we call it the *OFFSET* value). In the program, when we use the *RDTSC* or *RDTSCP* instruction to read the TSC value, the hardware will return the value: *BASE* + *OFFSET*. The initial value of the IA32_TSC_ADJUST MSR is 0. So we could set an non-zero value to IA32_TSC_ADJUST MSR on a logical core to simulate the modification of the TSC values.

In this experiment, we use the Statistics Scheme to evaluate the change of the range **d** due to its higher precision. We use the Statistics Scheme to calculate the range **d** twice. We simulate the modification by speeding up (add) 1 clock on the logical core that T2 runs on (see Fig. 6). The results of the first test are shown in Table 6. Table 7 shows the results of the second test. We can see that there is an obvious difference in these two tests with different confidence coefficients when M is 20 (bold ranges). Hence, using the Statistics Scheme is quite sufficient to detect such small modification of local clocks during record-and-replay.

7 Discussion

In this section, we discuss some limitations and possible future works for ReCBuLC.

Limitations. In this paper, we mainly propose a method that uses the local clocks to infer the global order in a multi-threaded program. We did not obtain the real difference value (**d**) between any two local clocks. Instead, we obtain a range of **d** as tightly as possible. If the difference between two timestamps from different logical cores is within this range, we cannot infer their order. So, if the programmer needs to know the order at any instrumentation points precisely, ReCBuLC may be not suitable for such a usage scenario. Nevertheless, it is quite useful in many real practices. For example, the aim of PRES is to reduce the required recording points, which can increase the number of unordered accesses, to get better runtime performance. CLAP only records the local execution paths. It doesn't care the global order during the runtime phase. ReCBuLC has proven to be quite usable and effective in such usage scenarios.

Possible future works. For CLAP-tc at the LOOP and FUN-CLOOP instrumentation levels, we instrument to record the local clocks at all the loops. If there is no shared data accesses in a loop, the instrumentation in this loop is unnecessary. So we plan to recognize the shared data accesses in the loop and determine if it needs the instrumentation. Besides PRES and CLAP, we plan to apply ReCBuLC to other systems, such as CCI [40] and CoopREP [41], to lower their recording overhead. For CCI, we plan to eliminate the use of global clock in CCI-prev scheme. For CoopREP, we plan to implement a lightweight logging system by using local clocks. We also plan to look into efficient algorithms that combine ReCBuLC and needed synchronizations to eliminate the effect of unordered accesses in application programs.

8 Related Work

In most record-and-replay and 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 to reduce both the recording overhead and the bug reproducing time.

PRES [18] does not record the global order of all events 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.

TABLE 7
Confidence Intervals After Speeding Up One of the Clock on the Logical Core That T2 Runs On (see Fig. 6)

N	20			50			100		
M	5	10	20	5	10	20	5	10	20
0.99	[-5.15, 9.84]	[-1.03, 4.30]	**[0.87, 2.44]**	[-0.61, 2.97]	[0.64, 2.16]	**[0.13, 1.75]**	[-0.16, 1.37]	[0.33, 1.27]	**[1.10, 1.39]**
0.999	[-12.01, 17.16]	[-1.78, 5.69]	**[0.01, 2.96]**	[-2.18, 4.29]	[0.19, 2.58]	**[0.53, 1.52]**	[-0.97, 2.12]	[-0.20, 1.53]	**[0.59, 1.63]**
0.9999	[-24.31, 27.90]	[-4.06, 7.27]	**[-0.54, 3.44]**	[-4.54, 6.90]	[-0.23, 2.68]	**[0.49, 1.67]**	[-2.60, 3.77]	[-0.65, 1.96]	**[0.39, 1.78]**
0.99999	[-43.49, 47.94]	[-5.53, 8.99]	**[-1.08, 3.94]**	[-9.27, 10.94]	[-0.62, 3.19]	**[0.41, 1.97]**	[-5.02, 6.16]	[-0.93, 2.61]	**[0.46, 1.66]**

Bold Ranges shows the significance difference, when compared with the Table 6.

Some systems try to reduce the recording overhead by only recording information that imply the global order of shared accesses. SMP-Revirt [16] and Scribe [17] 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 [25] divides the program into many epochs in 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 the 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 will 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 [6] allows each thread to record its own execution paths and searches for buggy interleaving by a SMT solver. ODR [21] reproduces concurrency bugs by ensuring the same output as in recording runs. It only records the global order of synchronization operations during execution. During the replay, similar to CLAP, it generates many interleavings and verifies their outputs by an SMT solver.

CoreDump [23] 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 point that the error occurs, 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 [19] 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.

Light [42] proposes a novel idea that recording only the flow dependence instead of recording the happen-before access order. Compared with other systems that recording the access orders, it could lower the performance and space overhead obviously.

CARE [43] presents an order-based deterministic replay technique that is capable of reducing the log size. It uses the value prediction cache to reduce the record cost. This recording method could only provide the value-deterministic replay. To address this problem, CARE presents two heuristics replay methods to make it practically useful for debugging.

Castor [44] is also a Record & Replay system that could provide consistently low overhead recording and real-time replay for modern multi-core workloads. To record the log efficiently during the record phase, it also uses timestamps for contention-free logging. It seeks for synchronizing all logical processors by using software method in OSes. Although the software method could not synchronizing the counters precisely, it claims that such difference of a few cycles is enough to use the local clock as the global clock in its usage scenarios. But it ignores that the frequency of the local clocks could be changed (mentioned in Section 2.2) during the record phase. That could affect the correctness of the recorded order. We think it could use our method in Section 3.2 to avoid such problems.

9 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∼85 percent and 84.66∼99.99 percent, respectively.

Acknowledgments

We would like to thank the anonymous reviewers whose suggestions have improved the presentation of our work. 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. Xiang Yuan, Zhenjiang Wang, and Jianjun Li did this work when they were in the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes a comprehensive study of real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2008, pp. 329–339.

[2] N. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *IEEE Comput.*, vol. 26, no. 7, 1993, Art. no. 18C41.

[3] SecurityFocus, "Software bug contributed to blackout." (2004). [Online]. Available: http://www.securityfocus.com/news/8016

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 263–272.

[5] DWARF Version 4 Released. (2010). [Online]. Available: http://dwarfstd.org/Announcement.php

[6] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording local executions to reproduce concurrency failures," in *Proc. 34th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2013, pp. 141–152.

[7] Intel 64 and IA-32 Architectures Software Developers Manual. (2017, Sep.). [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[8] Libunwind library. (2017). [Online]. Available: http://www.nongnu.org/libunwind/

[9] A dynamic binary instrumentation tool. (2012). [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[10] MIPS Architecture For Programmers. (2011, Apr.). [Online]. Available: https://www.imgtec.com/mips/architectures/mips32/

[11] Power ISA. (2013). [Online]. Available: https://www.ibm.com/developerworks/community/blogs/fe313521-2e95-46f2-817d-44a4f27eba32/entry/power_isa_version_2_07_the_latest_on_the_power_instruction_set_architecture?lang=en

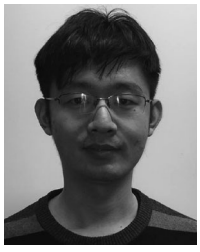
[12] Oracle SPARC Architecture. (2012). [Online]. Available: http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/sparc-processor-2516655.html

[13] J. R. Goodman and H. H. J. Hum, "MESIF: A two-hop cache coherence protocol for point-to-point interconnects," Univ. of Auckland, Tech. Rep., 2009, https://researchspace.auckland.ac.nz/bitstream/handle/2292/11593/MESIF-2004.pdf?sequence=7

[14] T. J. Leblanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. 36, no. 4, pp. 471–482, Apr. 1987.
[15] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, "Automatic logging of operating system effects to guide application-level architecture simulation," in *Proc. ACM SIGMETRICS/Int. Conf. Measure. Modeling Comput. Syst.*, 2006, pp. 216–227.
[16] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2008, pp. 121–130.
[17] S. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *Proc. ACM SIGMETRICS/Int. Conf. Measure. Modeling Comput. Syst.*, 2010, pp. 155–166.
[18] S. Park, et al., "PRES: Probabilistic replay with execution sketching on multiprocessors," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2009, 177–192.
[19] Y. Chen, W. Hu, T. Chen, and R. Wu, "LReplay: A pending period based deterministic replay scheme," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 187–197.
[20] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient online multiprocessor replay via speculation and external determinism," in *Proc. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 77–90.
[21] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multicore debugging," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2009, pp. 193–206.
[22] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 289–300.
[23] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," in *Proc. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 155–166.
[24] M. Xu, R. Bodik, and M. Hill, "A "flight data recorder" for full-system multiprocessor deterministic replay," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 122–135.
[25] K. Veeraraghavan, et al., "DoublePlay: Parallelizing sequential logging and replay," in *Proc. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, Art. no. 3.
[26] D. Hower and M. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 265–276.
[27] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 207–216.
[28] L. De Moura and N. Bjorner, "Z3: An efficient SMT solver," in *Proc. Theory Practice Softw. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.
[30] Apache HTTPD. (2016). [Online]. Available: http://httpd.apache.org/
[31] Cherokee Web Server. (2014). [Online]. Available: http://cherokee-project.com/
[32] Kernel Source Warning. (2014). [Online]. Available: http://lxr.free-electrons.com/source/arch/x86/kernel/tsc.c
[33] Temperature Problem. (2010). [Online]. Available: https://lwn.net/Articles/388188/
[34] SMI Problem. (2010). [Online]. Available: https://lwn.net/Articles/388286/
[35] WienerCKhinchin theorem. (2017). [Online]. Available: https://en.wikipedia.org/wiki/Wiener%E2%80%93Khinchin_theorem
[36] Memory Ordering. (2017). [Online]. Available: https://en.wikipedia.org/wiki/Memory_ordering
[37] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid program analysis for determinism," in *Proc. 34th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2012, pp. 463–474.
[38] B. Dutertre and L. De Moura. (2006, Sep.) "The Yices SMT solver." [Online]. Available: http://yices.csl.sri.com/tool-paper.pdf
[39] J. N. Gray, "Why do computers stop and what can be done about it?" in *Proc. Fifth Symp. Reliability Distrib. Softw. Database Syst.*, pp. 3–12, Jan. 1986.
[40] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.* 2010, pp. 241–255.
[41] N. Machado, P. Romano, and L. Rodrigues, "Lightweight cooperative logging for fault replication in concurrent programs," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12.
[42] P. Liu, X. Zhang, O. Tripp, and Y. Zheng, "Light: Replay via tightly bounded recording," in *Proc. 36th ACM SIGPLAN Conf. Program. Language Des. Implementation*, Jun. 2015, pp. 55–64.
[43] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "CARE: Cache guided deterministic replay for concurrent Java programs," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 457–467.
[44] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, "Towards practical default-on multi-core record/replay," in *Proc. 22nd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2017, pp. 693–708. DOI: https://doi.org/10.1145/3037697.3037751

Zhe Wang received the bachelor's degree from the Beijing University of Technology, in 2012. He is currently working toward the PhD degree in the State Key Laboratory of Computer Architecture, Institute of Computing Technology. His research interests include dynamic compilation, binary translation and optimization, and software security.

Chenggang Wu received the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences (ICT CAS), and his research was supported by National Science Foundation of China (NSF), the National High Technology Research and Development Program of China, and the National Science and Technology Major Project of China. He is currently a professor with the Key Laboratory of Computer System and Architecture of ICT CAS. He serves as the general co-chair of CGO 2013, program co-chair of APPT 2013, and the program committee member of PLDI 2012, PLC 2012, PPPJ 2014, CGO 2015-2017, PPoPP 2017, and AMAS-BT. He is serving as the member of Computer Architecture Professional Committee of China Computer Federation. His research interests include the dynamic compilation, including binary translation, dynamic optimization, bug detection on concurrent program, and software security.



Xiang Yuan received the bachelor's degree from Shandong University, in 2007 and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2015. And now he is in Huawei Technologies. His research interests include dynamic compilation, binary translation, and optimization.

Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He also served as the director in the Institute of Information Science (IIS) at Academia Sinica, in Taiwan between 2008 and 2011. Before joining the University of Minnesota, he was an associate director of the Center for Supercomputing Research and Development (CSRD), the University of Illinois, Urbana-Champaign. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems, the National Science Foundation, Washington, DC. He served as the editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems* between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization, compilers and architectural issues related multi-core/many-core systems. He is a fellow of the IEEE.

Zhenjiang Wang received the BS degree in computer science from Tsinghua University, in 2005, and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2011, and then joined the State Key Laboratory of Computer Architecture. And now he is in Huawei Technologies. His research interests include dynamic compilation, binary translation, and optimization.

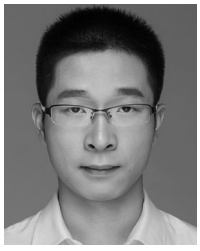
Jianjun Li received the BS degree in computer science from Harbin Engineering University, in 2006, and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2012. He is now in Horizon Robotics, Inc. His research interests include dynamic program analysis, program optimization, and binary translation.

Jeff Huang received the PhD degree from the Hong Kong University of Science and Technology, in 2012 and the postdoc degree from the University of Illinois, Urbana-Champaign, in 2014. He is an assistant professor with the Texas A&M University. His research focuses on developing practical techniques and tools for improving software reliability and performance. He has published extensively in premiere software engineering conferences and journals such as *ACM Transactions on Software Engineering and Methodology*, PLDI, OOPSLA, ICSE, FSE, ISSTA, etc. He is a member of the IEEE.

Xiaobing Feng joined the Institute of Computing Technology, Chinese Academy of Sciences since July, 1999. He was working there as an assistant professor, associate professor and then the director of Lab of Andance Compiling Technology. He is now a professor, doctoral advisor and the vice director of Key Laboratory of Computer System and Architecture, ICT. He was one of the main contributors of "Autopar" (which is a parallel compiler for dawning parallel computing systems) and "ParaVT" (which is a parallel program behavior and performance events monitor tool for dawning parallel computing systems). He held the project of developing early versions of binary translation from "X86/Linux" to "LOONGSON/Linux". He also held the project of developing and implementing of compiler and related tools for micro-architectures, such as LOONGSON 2E and LOONGSON 3A.

Yanyan Lan received the BE degree in statistics from Shandong University, Jinan, China, in 2005 and the PhD degree in probability and statistics from the Institute of Applied Mathematics, Academy of Mathematics and System Sciences, Chinese Academy of Sciences, Beijing, China, in 2011. She is currently an associate professor in the Institute of Computing Technology, Chinese Academy of Sciences. She leads a research group working on Big Data and Machine Learning. Her current research interests include machine learning, web search and data mining, and big data Analysis. She has published more than 30 papers on top conferences including ICML, NIPS, SIGIR, WWW et al., and the paper entitled "Top-k Learning to Rank: Labeling, Ranking, and Evaluation" has won the Best Student Paper Award of SIGIR 2012.

Yunji Chen received the bachelor's degree from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, and the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2002 and 2007, respectively. He is currently a full professor with ICT. His current research interests include parallel computing, microarchitecture, and computational intelligence. He has authored or co-authored one book and more than 60 papers in the above areas. He was a recipient of the Best Paper Award from ASPLOS'14 and MICRO'14 for the investigations in neural network accelerators.



Yuanming Lai received the BS degree in digital media technology from the Central China Normal University, in 2013, and the master's degree in pattern recognition and intelligence system from the Huazhong University of Science and Technology, in 2016. Now he is in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include information security and machine learning.

Yong Guan received the graduated degree from the China University of Mining and Technology, in 2004 and the PhD degree. He is currently a professor with the Capital Normal University. His research interests cover formal verification, robot, and the embedded system with high reliability.