Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation

Ziyi Zhao*, Zhang Jiang*, Ying Chen*, Xiaoli Gong*, Wenwen Wang†, Pen-chung Yew‡
*College of Computer Science, Nankai University, Tianjin, China
†Department of Computer Science, University of Georgia, Athens, GA, USA
‡Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

***Abstract*—Dynamic Binary Translation (DBT) is a key enabler for cross-ISA emulation, system virtualization, runtime instrumentation, and many other important applications. Among several critical requirements for DBT, it is important to provide equivalent semantics for atomic synchronization instructions such as `Load - Link / Store - Conditional (LL/SC)`, which are mostly included in the *reduced-instruction set architectures (RISC)* and `Compare-and-Swap(CAS)`, which is mostly in the *complex instruction set architectures (CISC)*. However, the state-of-the-art DBT tools often do not provide a fully correct translation of these atomic instructions, in particular, from RISC atomic instructions (i.e. LL/SC) to CISC atomic instructions (i.e. CAS), due to performance concerns. As a result, some may cause the well-known ABA problem, which could lead to wrong results or program crashes. In our experimental studies on QEMU, a state-of-the-art DBT, that runs multi-threaded lock-free stack operations implemented with ARM instruction set (i.e. using LL/SC) on Intel x86 platforms (i.e. using CAS), it often crashes within 2 seconds.**

Although attempts have been made to provide correct emulation for such atomic instructions, they either result in heavy execution overheads or require additional hardware support. In this paper, we propose several schemes to address those issues and implement them on QEMU to evaluate their performance overheads. The results show that all of the proposed schemes can provide correct emulation and, for the best solution, can achieve a min, max, geomean speedup of 1.25x, 3.21x, 2.03x respectively, over the best existing software-based scheme.

***Index Terms*—Dynamic Binary Translation, Scalability**

I. Introduction

Dynamic binary translation (DBT) has been widely used in many application areas [1] [2] [3] [4]. Therefore, a considerable amount of effort has been dedicated to enhance its performance and functionality, in particular, for multi-threaded applications [5] [6] [7]. As multi-core and many-core processors have become ubiquitous, DBT tools such as QEMU [8] [9] and PICO [10] have been either extended or designed to run on such platforms.

One of the most challenging requirements for emulating multi-threaded applications on multicore systems is semantically equivalent emulation of the code regions that involve atomic instructions. There are different support of atomic instructions in different instruction set architectures (ISAs). Examples include `Compare-and-Swap(CAS)` instruction on CISC machines (e.g. `cmpxhg` on Intel x86 platforms), and `Load-Link/Store-Conditional(LL/SC)` instructions on RISC machines (e.g. `ldrex` and `strex` on ARM platforms).

Xiaoli Gong (gongxiaoli@nankai.edu.cn) is the corresponding author.

For multi-threaded *guest* applications emulated on a DBT that runs on a multi-core *host* machine, such as QEMU, there are several ways to carry out such an emulation. We can use a *single host thread* to emulate *multiple guest threads*. In this case, the emulation of guest atomic and synchronization instructions is fairly simple. *Each* guest atomic instruction can be safely emulated by *multiple* host instructions, as long as they are *completed* before the next guest instruction after the atomic instruction is emulated, or before it is switched to another guest thread. In this case, the *atomicity* of a guest atomic instruction is guaranteed, i.e. its atomicity can be achieved "for free", with multiple non-atomic host instructions.

The situation is much more complex when multiple host threads are involved to emulate multiple guest threads for scalability concerns. In this case, the emulation of guest atomic instructions requires more careful orchestration by a host machine that has a different set of atomic instructions. From the prior work [11], it is found that is is easier to emulate guest atomic instructions such as `CAS` from a CISC machine with host synchronization instructions such as `LL/SC` on a RISC host machine. However, it is much more challenging when it is the other way around. It is because additional hardware support is usually provided to guarantee the *strong atomicity* [12] on the *synchronization variable* involved in the `LL/SC` instructions on RISC machines. The *strong atomicity* guarantees that any update/modification to the synchronization variable of the `LL/SC` pair by *any other thread* will break its atomicity. Using software to emulate `LL/SC` without enforcing the strong atomicity will cause the so-called *ABA problem* [13].

The ABA problem can occur when multiple threads interleave their accesses to a shared memory location. It is a correctness issue of DBT that leads to crashes or even wrong results without crashing, making it even more difficult for programmers to locate the bug. *Any* program that relies on `LL/SC`, such as those in PARSEC, has potential ABA problems on QEMU. Our test case using multi-threaded Arm lock-free stack always crashes within 2 seconds with 16 threads on QEMU. Hence, our focus is on how to ensure that ABA does not occur and with a minimal cost on performance.

Assume one thread reads the shared memory location and gets a value A. The shared memory location is then modified

with a new value B by another thread, and immediately restored back to the value A by the same or a different thread. When the first thread revisits the shared memory location, it will find the value is unchanged. It can draw a wrong conclusion that the location has never been accessed and modified between its two visits. Unless additional effort is made, which often requires a lot of overhead, the thread has no way of knowing whether the content of the shared memory location has been changed or not between the two visits.

It is nontrivial to resolve the ABA problem. Prior research has proposed to use a version number-based approach [13] [14] and Hazard Pointers [15] to address this problem. In some recent work, PICO [10] addresses the ABA problem in the context of emulating atomic/synchronization instructions in DBT. But, they all involve a substantial amount of runtime overhead or work only for some special cases, but may work incorrectly on others, to cut down the overhead. We will further discuss those issues in Section II-A.

In this paper, we explore the design space of emulating atomic instructions from a RISC guest machine to a CISC host machine that usually involves emulating `LL/SC-like` instruction pairs using `CAS-like` instructions. We propose two new strategies. One is called *HST* (Hash Table-Based Store Test) scheme, which maintains a non-blocking hash table to record the latest thread that updates a memory location, and requires a thread to check the access permission before accessing the memory location. The other is called *PST* (Page Protection-Based Store Test) scheme, which takes advantage of the page protection mechanism in OS to capture memory access violation. In addition, we discuss the design trade-off between performance overhead and correctness issues among different design strategies. Comprehensive experiments are conducted to evaluate their effectiveness and efficiency.

In summary, we have made the following contributions.

- We have proposed two new strategies, HST and PST, to allow correct and efficient emulation of atomic/synchronization instructions in cross-ISA DBTs.
- We have implemented and evaluated various design tradeoffs on QEMU. Our experimental results show that our proposed schemes can correctly emulate the target atomic and synchronization instructions. The best scheme can achieve an average of 2.03× speedup over the best existing software-based scheme with a similar capability.

The rest of this paper is organized as follows. Section II provides some background on DBT, `LL/SC` instructions, the ABA problem and our motivation. Section III describes the design details of our approaches. Section IV shows some experimental results with discussions. Section V present some related work, and Section VII concludes the paper.

II. Background and Motivation

In this section, we give a brief introduction on the `LL/SC` instruction pair (Section II-A), their emulation in DBT (Section II-B), the potential ABA problem (Section II-C), and the issues related to atomicity (Section II-D).

A. Load-Link (LL) and Store-Conditional (SC)

`LL` and `SC`, which are in the form of `ldrex` and `strex` in ARM's instruction set, are a pair of instructions used in multi-threaded programs to support synchronization [16]. The `LL` instruction loads the value stored in a target memory location for its *synchronization variable* into a register. It is also associated with an *exclusive flag* to monitor whether the target memory location has been modified before the subsequent `SC` instruction is executed. The `SC` instruction will store a new value to the target location only if no update has occurred since its `LL` instruction. In other words, there is a period between the `LL` instruction and the `SC` instruction that should be kept "atomic". If the target location has been modified by *other threads* during this "atomic period" using either `SC` instructions or regular `store` instructions, the exclusive flag will be nullified and its corresponding `SC` instruction will fail.

Note that the `store` instruction from the same thread will not nullify the exclusive flag. In addition, the `LL/SC` pair cannot be nested, which means only one memory location can be monitored by one thread at a time. If there are multiple `LL` instructions and each with its own synchronization variable, only the target location of the last `LL` instruction will be monitored.

In general, the `LL` and `SC` instructions are designed for RISC architectures. They are often used in system libraries for critical sections and functions such as *atomic_add* and *mutex_lock*. However, the `LL` and `SC` instructions can be used separately in other scenarios, for example, to avoid the ABA problem in implementing a lock-free stack in a SHA-1 pattern searching algorithm [17].

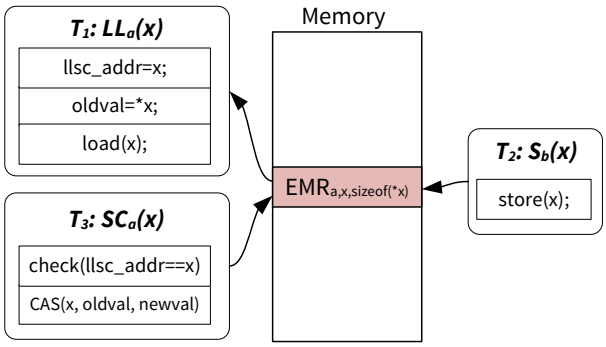

Fig. 1: "Slightly incorrect" emulation of `LL/SC` in current QEMU.

B. LL/SC Emulation in DBT

Prior work has proposed techniques to correctly emulate `LL/SC` instructions on CISC machines. A straightforward technique, called *PICO-ST* [10], associates the target address of the synchronization variable and the thread ID of a `LL/SC` pair with a software exclusive flag. The `SC` instruction will check the exclusive flag before updating the target location. It also instruments all `store` instructions of other threads to check their associated addresses against the target address

of the LL/SC pair, and clears the exclusive flag if there is a conflict. PICO-ST can correctly emulate LL/SC instructions. However, it incurs significant runtime overheads as every store instruction has to be instrumented and checked against the target address of the active LL/SC pair. Furthermore, it has to be done in a multi-threaded environment, i.e. such a check and update has to be done atomically as well. The instrumentation is implemented as a *helper_function* in PICO-ST, which incurs extremely heavy runtime overheads.

Another technique, called *PICO-HTM* [10], tries to reduce the runtime overhead by using the hardware transactional memory (HTM) support. In this technique, HTM hardware will detect the memory updates to the synchronization variable during the "atomic period". It encapsulates the code between the LL/SC pair as a *transaction*. "Strong atomicity" is enforced by HTM during the execution of a transaction. That is, if there is any update/change to the memory locations accessed in an active transaction, it will be detected by the HTM hardware, and the transaction will fail and be aborted. The aborted transaction will either be rolled back and restarted from the beginning, or follow a fallback path provided by the programmer to ensure forward progress. However, it is not trivial to integrate HTMs into a DBT as the emulation part of the software in DBT will often interfere with the emulated code in HTM [18], which can cause unexpected and frequent HTM aborts unless extreme care is given. As HTM is not commonly supported on today's machines, HTM-based techniques are also less portable.

Due to the performance and portability concerns, the implementation in the latest version of QEMU uses a *slightly incorrect* design, called *PICO-CAS* in [10]. It is based on the observation that atomic operations in C/C++11 are mostly implemented through the CAS-like instructions. In the Linux kernel, LL/SC instructions are also only used to emulate the CAS-like operations [10]. Thus, in QEMU, the code section that is encapsulated by LL/SC instructions is directly translated into the corresponding code section using CAS instructions without enforcing the *strong atomicity* in the encapsulated code section, i.e. it assumes it is highly unlikely that the target memory location will be updated/changed during this short period of time.

Figure 1 shows some details of the *PICO-CAS* code. Specifically, when a thread executes the LL instruction, the value and the address of its synchronization variable are recorded in *oldval* and *llsc_addr* after loading the memory value. When the thread executes the subsequent SC instruction, after matching the *llsc_addr*, the CAS instruction is used to compare the current value of the target memory location with the *oldval*. If these two values match, it assumes that no memory update has occurred, and the emulated SC can update the target memory location. Obviously, it is not always true, and the current implementation of QEMU can suffer from the ABA problem even if the guest binary is *"ABA safe"*.

C. The ABA Problem

The ABA problem can arise when we try to implement a *lock-less data structure* using the atomic instructions such as

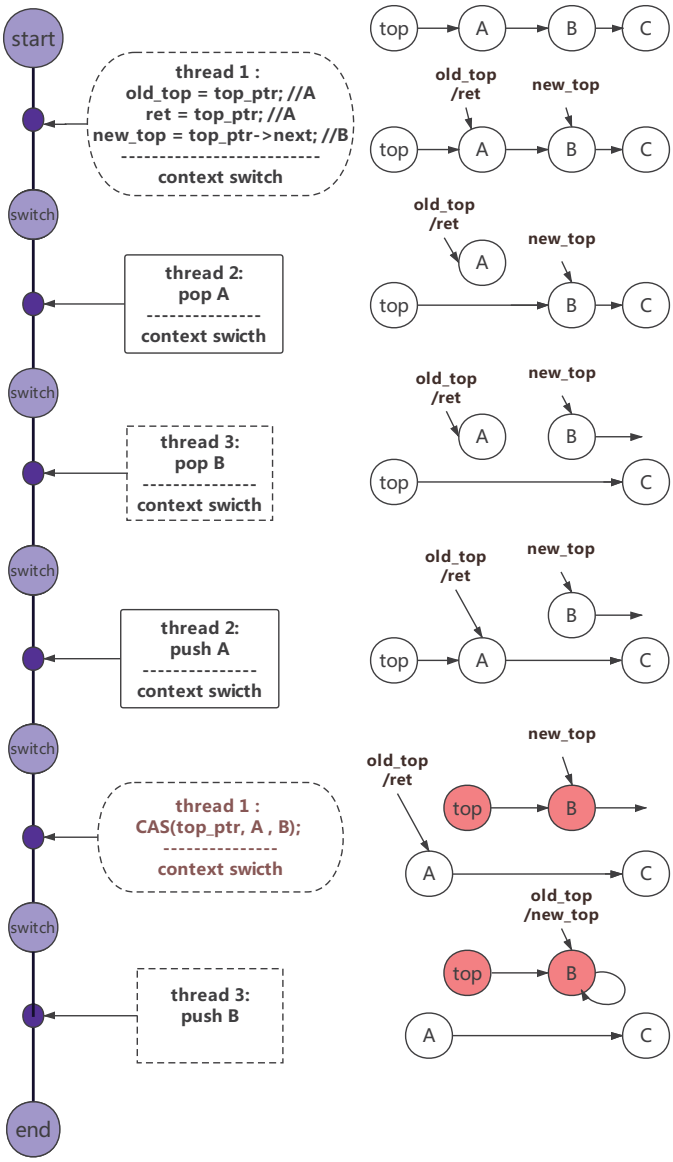

Fig. 2: An example of the ABA problem.

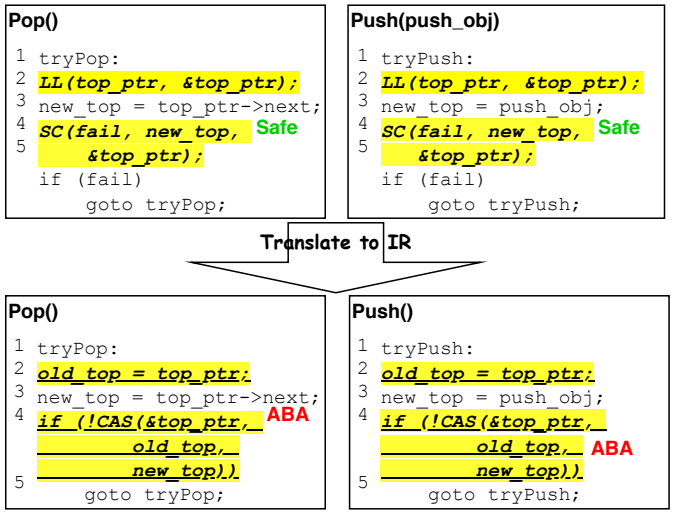

Fig. 3: A lock-free stack before and after QEMU transalation.

CAS to check for memory updates in a critical section. Figure 2 gives an example of a *lock-less stack* implemented by the code shown in Figure 3. Figure 3 shows the code *before* and *after* the QEMU translation. In Figure 2, we assume there are 3 threads concurrently accessing the lock-free stack by repeating POP and PUSH operations using the code shown in Figure 3.

Initially, there are three data elements, A, B, C on the stack, with A at the top of the stack. Thread 1 first tries to execute POP. It records that the *old_top* is pointing to A, and both *ret* and *new_top* are pointing to B. Assume it is then interrupted by a context switch and Thread 2 pops A. Afterwards, Thread 3 pops B and Thread 2 pushes A back, leaving the *top_ptr* pointing to A again. Thus, when Thread 1 continues its execution and executes CAS, it incorrectly succeeds and makes the *top_ptr* point to B, even though the stack has been updated by other threads, which may lead to a wrong result or even program crash.

The root cause of the ABA problem is that the CAS instruction only checks the target value, and *"value unchanged"* is considered as *"nothing has been changed"*. The top two boxes in Figure 3 show an implementation of a lock-less stack using the LL/SC instructions. Notice that the checking of the old value in line 2 is replaced by the LL instruction, and then checked by the SC instruction in line 5. In this way, if the value has been modified and restored, it will be detected by the nullified exclusive flag.

Although the LL/SC instructions can deal with the ABA problem, it is not correctly translated by the current QEMU. Figure 3 shows its translation in pseudo code. Hence, even though the program runs correctly on a native ARM machine, it is likely to crash on QEMU running on Intel x86 platforms due to the ABA problem. Our experimental studies show it always crashes in 2 seconds.

D. Strong Atomicity vs. Weak Atomicity

The concept of *strong atomicity* and *weak atomicity* describes how strictly the atomicity is enforced. For *strong* atomicity, a transactional semantic guarantees the atomicity is enforced between transactions and non-transactional codes. while *weak* atomicity guarantees the atomicity is enforced only among transactions, but not with the non-transactional code regions [19] [12].

Based on the above definition, the LL/SC instructions have the requirement of strong atomicity. Any change to their atomic variable[1] either by other LL/SC instructions or store instructions in non-transactional code regions in other threads will break the atomicity. As atomic variables are rarely updated by the regular store instructions in non-transactional code regions, which means only enforcing weak atomicity without monitoring other store instruction could provide adequate results with reduced overheads. Current QEMU only enforces weak atomicity for the emulation of LS/SC to avoid substantial overhead.

[1]We will call the operand in the LL/SC instructions *synchronization variable* and *atomic variable* interchangeablely in the paper.

III. Two Proposed Schemes: HST and PST

In this section, we present two main schemes with some of their optimized variations to address the ABA problem. In general, they can be classified as software-based schemes and hardware-based schemes, respectively

A. Hash Table-Based Store Test (HST) Scheme

The *hash table-based store test scheme* (HST) is a software-based scheme. In order to correctly emulate the LL and SC instructions, the violation of their atomicity must be detected. A straightforward way is to track the access of the atomic variable of each LL/SC instruction, and instrument each store instruction in non-transactional code regions to check for potential conflicts with any of those atomic variables. Their codes are shown in Figure 5. The atomic variable is recorded in LL, updated in store and checked in SC.

It is worth noting that since store instructions happen more frequently than atomic instructions, it is critical to implement them in a highly efficient manner. Hash table lookups and updates are in the critical path of all the instrumented store operations. It is thus a major consideration to minimize hash table operations in our design.

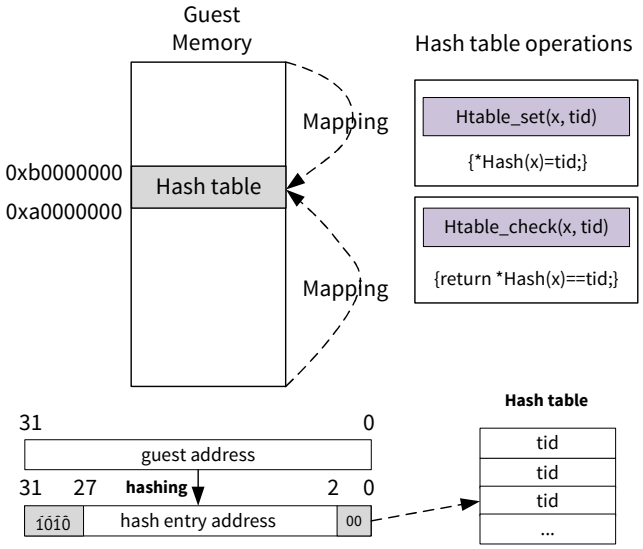

Fig. 4: Hash table design in HST.

As shown in Figure 4, a very simple and efficient hashing algorithm is used in our implementation. It maps the guest virtual space to a hash table in an unused memory region (from 0xa0000000 to 0xb0000000 in this example). The hash function takes in a guest virtual address and outputs an address points to a hash entry by setting the highest half byte to the start address of the hash table (0xa in this example) while zeroing the lowest 2 address bits to provide a 4-byte alignment for each hash entry. To reduce the access overhead, we provide a *non-blocking* hash table that has only one field (instead of multiple fields), and allows it to be updated with a single instruction. The index of the hash table is embedded in its memory address, and the value of the entry is its thread ID. In this way, the *Htable_set* and the *Htable_check* operations

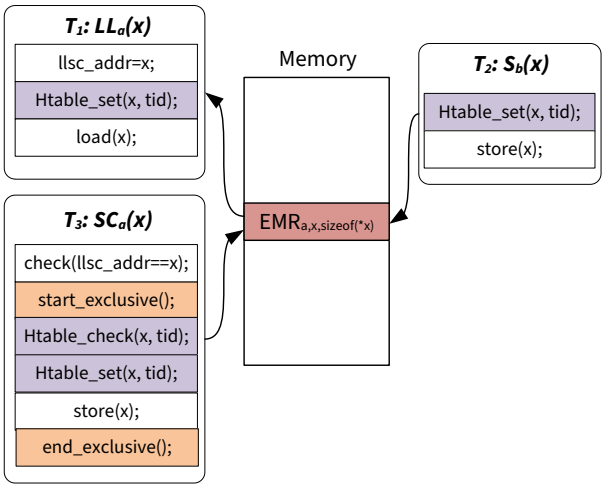

Fig. 5: `LL/SC` and `store` in HST.

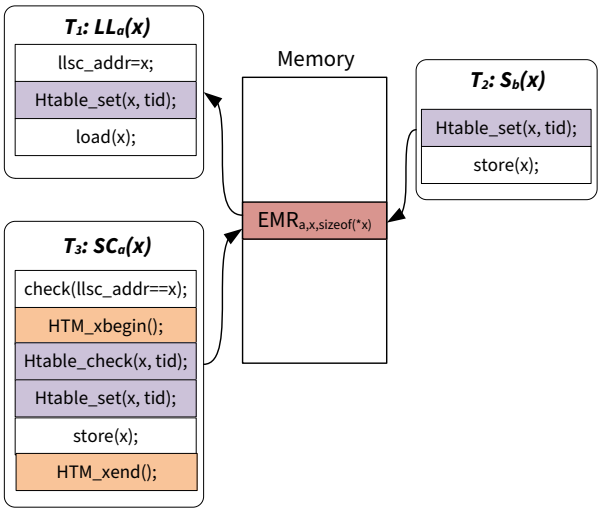

Fig. 6: `LL/SC` and `store` in HST-HTM.

shown in Figure 5 can be done with a single store and load, respectively, without the need for atomic operations. Also, this simple design makes it possible to implement them in DBT IR, instead of using *helper_functions* whose calls require a context switch to QEMU and will incur much higher overhead.

As shown in Figure 5, each `LL` and `store` instruction is instrumented (at the IR level) to set the corresponding entry value to the thread ID (*tid*), while `SC` checks the entry value to see if it has been modified and the *tid* is matched. These operations have to be done exclusively. As shown in Figure 5, `start_exclusive` and `end_exclusive` (the two functions provided by QEMU) are used to ensure the atomicity of `SC`.

Note that there could be conflicts at the hash table. These conflicts don't affect the correctness, and happen rarely, i.e. only 2.4% in PARSEC. For example, between `LL` and `SC` instructions, if another thread issues a *store* instruction that has the same hashed index as the `LL/SC` but with a different address, the store instruction will modify the hash entry to its thread ID and cause the `SC` instruction to fail. However, the `SC` will perform a retry, and its atomic semantic is not affected.

B. Improving HST with HTM

HST can be further optimized if there is additional hardware support, such as hardware transactional memory (HTM), to implement the critical section for the `SC` emulation. As shown in Figure 6, the check and the update operations of the hash table in `SC` can be implemented with an HTM transaction to replace the slow critial section, i.e. replacing `start_exclusive` with `HTM_xbegin` and `end_exclusive` with `HTM_xend`.

Note that this approach is different from the PICO-HTM scheme [10] mentioned earlier. PICO-HTM adds `HTM_xbegin` before `LL` and `HTM_xend()` after `SC`. However, Pico-HTM can cause "livelocks" if QEMU needs to emulate or translate guest code between LL/SC, i.e. QEMU code becomes part of transaction in addition to code between LL/SC. It *significantly* inflates code size within HTM and can cause repeated HTM aborts [18]. To address this issue, HST-HTM uses a hash table to limit the transaction code and emulates "only" SC as described in Figure 6, instead of code "between" LL/SC as in PICO-HTM.

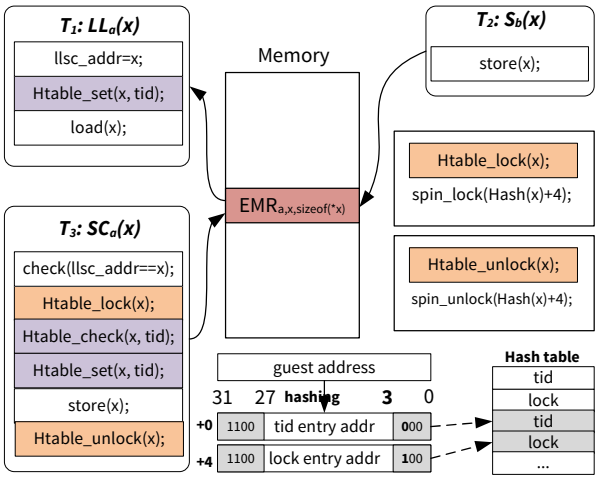

Fig. 7: Hash table design, `LL/SC` and `store` translation for HST-WEAK.

C. Improved Hash Table-Based Store Test (HST-WEAK)

To look for opportunities to further improve the performance of HST, a code analysis is conducted on the generated binary code of PARSEC [20] benchmark suite and the Linux kernel. Based on the analysis, the shared data can be modified by multiple threads using atomic instructions such as `LL/SC` during lock contention, and only be updated by the lock-owner thread with normal `stores`.

Based on the observation, we simplify the atomic instruction emulation with a weak atomicity model, i.e. only the effects of atomic instructions(`LL/SC`) among threads are considered. The `store` instructions are not instrumented to check for conflicts

with LL/SC. We also assume there is no race condition between LL/SC instructions and store instructions. Hence, the HST can be simplified as shown in Figure 7. We call this optimized scheme *HST-WEAK* scheme.

Obviously, HST-WEAK is not a fully correct solution since potential conflicts with store instruction are neglected. Similar to PICO-CAS, it may thus lead to the ABA problem. It is different from PICO-CAS, which is used in current version of QEMU. In HST-WEAK, it will produce a correct result if LL/SC instructions are used following their intended programming convention. This is because, in HST-WEAK, we still track the thread ID of LL/SC while PICO-CAS only guarantees the update operations are atomic (i.e. using CAS). Hence, any conflict among LL/SC in different threads with the same atomic variable can be captured by HST-WEAK, while PICO-CAS may not, if the timing of an LL/SC pair is overlapped with another in a different thread (see Section IV-A for more details).

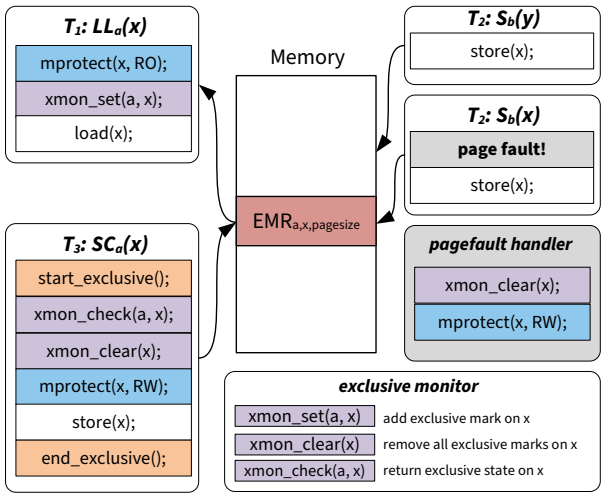

Fig. 8: LL/SC and store translation for PST.

D. Page Protection-Based Store Test (PST) Scheme

To monitor the changes to the atomic variable of LL/SC by the *store* instructions in non-transactional code regions, we can also employ the page protection mechanism in OS. When emulating the LL instruction, in addition to access the hash table entry, it will set the page that contains its atomic variable to "*read-only*". Any store operation that tries to access the atomic variable will be captured and triggers a page fault. In the page-fault handler, we check whether the store address matches the address of the atomic variable or not. If the store address matches, the atomicity of the LL/SC is broken and a retry of LL/SC will be needed. Otherwise, it will be executed as a regular store operation without breaking the atomicity. We call it the *Page Protection-Based Store Test (PST) Scheme*, and its procedure is shown in Figure 8.

In the PST scheme, before SC operations are to be carried out, the page protection need to be changed from "*read-only*" back to "*writable*". After the SC operations are completed, the page protection should be changed back to "*read-only*" again. This process could incur high overhead because a context switch to OS kernel mode and change page protection may require all threads to be suspended.

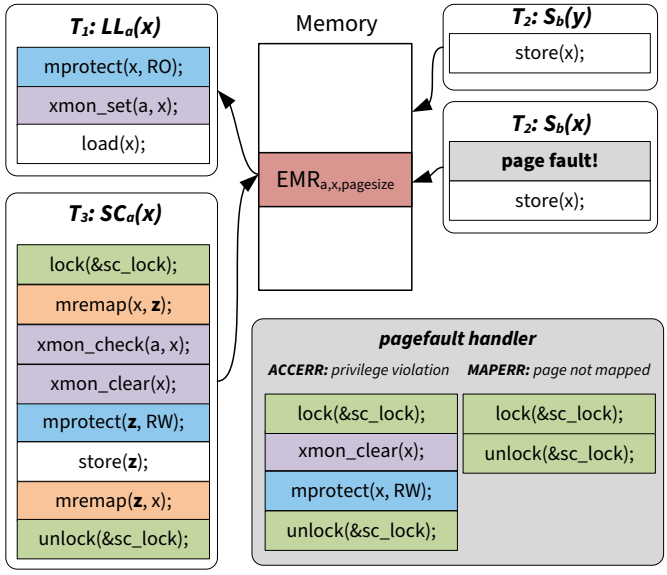

Fig. 9: LL/SC and store translation for PST_REMAP.

Compared to the HST scheme, PST eliminates the need to instrument all of the store instructions in the non-transactional code regions. However, page granularity may be too large for monitoring the atomic variable of LL/SC in some applications. The store instructions to the same page of the atomic variable may have the effect of "*false sharing*" and trigger unintended page faults.

E. Optimization of Remap

The critical section in PST is to avoid writing from other threads, while only allowing the SC thread writing to the page. The crux of the problem is giving threads with different privileges to one page. We can therefore employ the sys_mremap [21] feature to map one physical page to multiple virtual addresses, so that one page could have different page privileges. For example, in Figure 9, during SC, we remap the page x to a new address z with write privilege, leaving the original address x unmapped. Then all the operations inside emulated SC is redirected to address z. This prevents other threads reading or modifying the page since all the operations to the ummaped address x triggers pagefault with erro number MAPERR(mapping error). Therefore, the pagefault handler of mapping error simply waits the completion of SC by locking and unlocking. In SC, after the critical operations, we remap the page at z back to original address x and performs unlock to wake the threads blocked by pagefault handler. We will discuss the performance details in Section IV.

IV. Experimental Results and Analysis

To study the efficiency and the effectiveness of our schemes, We have implemented them on QEMU-4.1, which is the latest version of QEMU when we started this project. As QEMU-4.1

can be considered as an implementation of PICO-CAS, we also implemented PICO-ST and PICO-HTM faithfully to the best of our knowledge. To measure the overhead and scalability, We conducted our experiments on a bare-metal cloud server from Alibaba [22], which has 52 physical cores and 187GB memory. Due to the lack of the HTM support on the cloud server, we also employed a local workstation with 10 cores (20 threads), 2.20GHz Intel(R) Xeon(R) Silver 4210 CPU and 64GB memory to evaluate the HTM-based schemes. The operating system is Ubuntu-18.04LTS with linux-4.15.0-65 kernel and the benchmark suite is PARSEC 3.0 [20] with `simlarge` input set. *Canneal* has only 30% parallelism so it is not an appropriate benchmark to study "scalability", and thus is excluded in the scalability analysis section but is included in the overhead analysis section. In all of our experiments, we use ARM as the guest ISA and Intek x86_64 as the host ISA. For all of the experiments, we run each case 3 times and take their average to account for potential variations.

A. Atomicity and the ABA Problem

To simplify our presentation, we introduce the following symbols to represent different events.

Symbol	Note
$LL_a(x(c))$	Thread *a* performs an `LL` to address *x*, with an initial value of c at x.
$SC_a(x(c,d))$	Thread *a* performs an `SC` of a value d to address *x*, with an initial value of c at x.
$S_a(x(c))$	Thread *a* performs a `store` of a value c to address *x*.

There are 4 basic execution sequences of *LL, SC* and *S* as shown below (using the symbols defined above) that can cause the ABA problem in PICO-CAS, which is used in current QEMU. The symbol "#" in `SC` means "don't care".

(Seq_1) $LL_a(x(c)) \rightarrow S_b(x(d)) \rightarrow S_b(x(c)) \rightarrow SC_a(x(c,\#))$

(Seq_2) $LL_a(x(c)) \rightarrow LL_b(x(c)) \rightarrow SC_b(x(c,d)) \rightarrow LL_b(x(d)) \rightarrow SC_b(x(d,c)) \rightarrow SC_a(x(c,\#))$

(Seq_3) $LL_a(x(c)) \rightarrow LL_b(x(c)) \rightarrow SC_b(x(c,d)) \rightarrow S_b(x(c)) \rightarrow SC_a(x(c,\#))$

(Seq_4) $LL_a(x(c)) \rightarrow S_b(x(d)) \rightarrow LL_b(x(d)) \rightarrow SC_b(x(d,c)) \rightarrow SC_a(x(c,\#))$

More complicated ABA scenarios can be derived from these basic sequences. According to the original `LL/SC` semantic, all above execution sequences will cause *SC* to fail. Based on the execution sequences, the *strong* and *weak* atomicity as defined in Section II-D can be described as follows.

weak $LL_a(x(c)) \rightarrow [LL_b(x(\#))] \rightarrow SC_b(x(\#,d)) \rightarrow SC_a(x(d,\#))$ (failed)

strong $LL_a(x(c)) \rightarrow S_b(x(c)) \rightarrow (failed)SC_a(x(c,\#))$ (failed)

The square brackets around $LL_b(x)$ means it can happen either before or after $LL_a(x(c))$.

Note that strong atomicity covers the cases of weak atomicity, i.e. if weak atomicity fails, strong atomicity also fails. If weak atomicity is supported, Seq_2, Seq_3, and Seq_4 will fail because there is a $[LL_b(x(\#))] \rightarrow SC_b(x(\#,\#))$ between `LL/SC`, which is not allowed in weak atomicity. In addition, if strong atomicity is supported, Seq_1 will also fail because $S_b(x(\#))$ is not allowed between `LL` and `SC`. Based on the above conditions, we can verify the strong and weak atomicity associated with each scheme as follows.

a) HST - strong atomicity: Initially, the hash table entry is set by Thread *a* with an $LL_a(x(c))$. Then `store` and `SC` from thread b are guaranteed to complete before $SC_a(x(c,d))$ since it is protected by an exclusive region. Thus, if the entry is changed by Thread b before the $SC_a(x(c,d))$, it will fail.

b) HST-WEAK - weak atomicity: Note that, in HST-WEAK, there is no instrumentation for stores. So, Thread a is not aware of any $S_b(x(c))$ between `LL/SC`. However, $SC_b(x(\#,c))$ is guaranteed to complete before $SC_a(x(c,d))$ due to the lock in the hash table. Hence, only weak atomicity is guaranteed.

c) HST-HTM - strong atomicity: Different from HST-WEAK, HTM tracks all store instructions. According to the atomicity provided by HTM, $SC_a(x(c,d))$ can be completed atomically. $S_b(x(c))$ can thus be completed before the $Htable\_check$ and the $SC_a(x(c,d)))$ will fail. This provides strong atomicity.

d) PST - strong atomicity: Same as HST, $SC_b(x(\#,c))$ and $S_b(x(c))$ will complete before the exclusive protection on $SC_a(x(c,d))$, which clears the exclusive state before $SC_a(x(c,d))$ and results in a failure.

To validate our proposed schemes, we also implement a micro-benchmark with a lock-free stack as described in Figure 3. Following the steps shown in Figure 2, when the pointer *next* of an entry in the lock-free stack points to itself, it is a sign that an ABA scenario has occurred. We use 16 threads to run the micro-benchmark program on an Intel x86 platform, and execute `POP` and `PUSH` operations 1048575(0xFFFFF) times. When all threads have completed, we go through the lock-free stack to check whether there exists an entry whose pointer *next* points to itself. We have tested the QEMU-4.1, PICO-ST, PICO-HTM, HST, HST-WEAK, HST-HTM, PST and PST-REMAP. The results show that among all schemes, only QEMU-4.1 has an average of 4% of the entries having the ABA problem, while all other schemes have none - even in HST-WEAK scheme.

B. Performance Evaluation

1) Scalability: Figure 10 shows the scalability of HST, HST-WEAK, PST and PICO-ST from one thread to 64 threads. We normalize the speedups over their own single-thread performance. HST-WEAK scales the best for all of the programs. HST, PST and PICO-ST also scale well for all of the programs except *fluidanimate* and *swaptions*. As PICO-CAS ignores the strong atomicity requirement of `LL/SC`, it avoids the runtime overhead of instrumentation and synchronization. HST-WEAK, on the other hand, uses a weak atomicity model

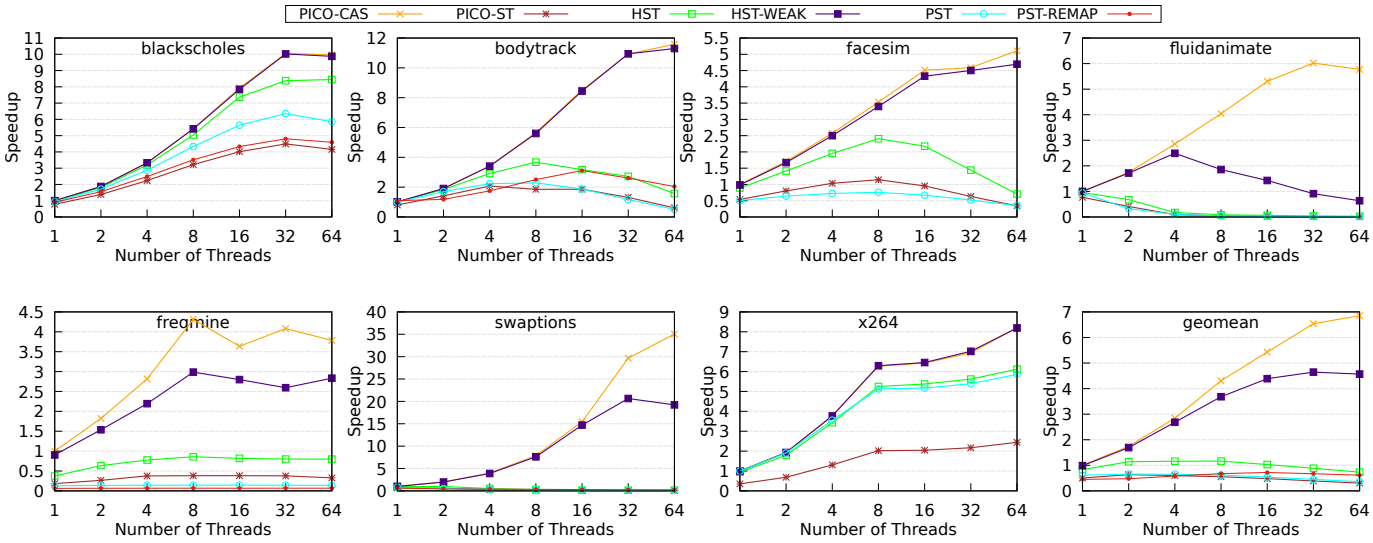

Fig. 10: Normalized Speedup over single-thread on Pico-CAS for arm-PARSEC.

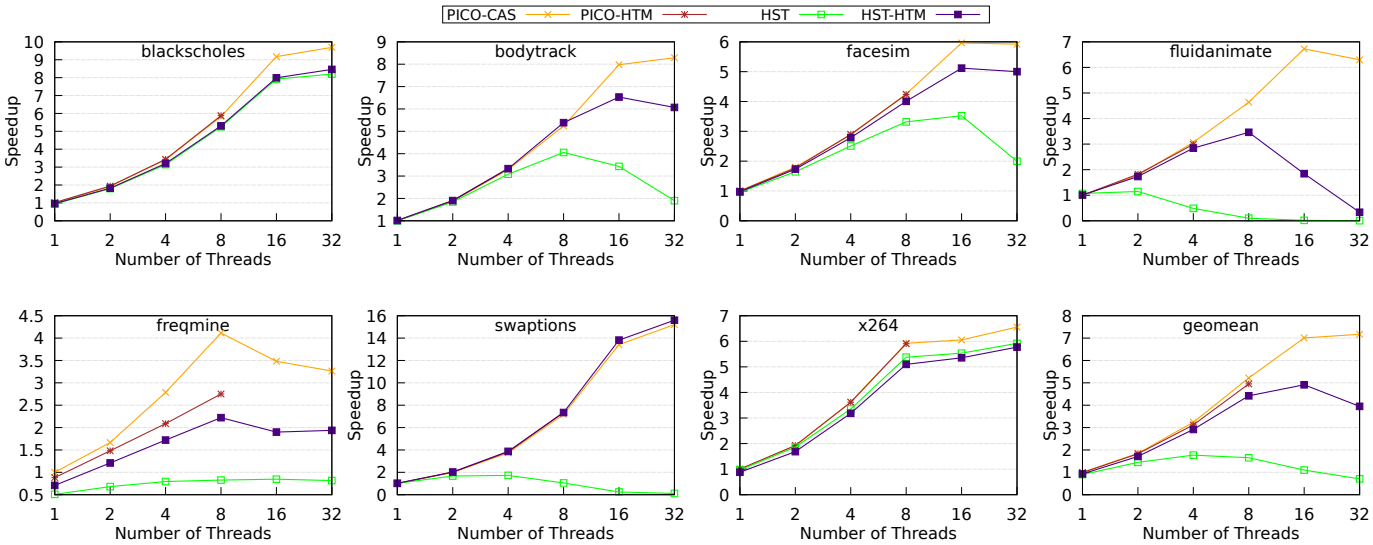

Fig. 11: Normalized Speedup over single-thread on Pico-ST for arm-PARSEC (Correct schemes).

and only enforces the exclusion during the `SC` emulation. It shows that HST-WEAK actually has a similar performance compared to that of PICO-CAS. Due to a more optimized implementation that includes IR-level instrumentation and a non-blocking hash table, HST outperforms PICO-ST in almost every benchmark program. As for PST, it delivers the worst performance for many benchmarks.

To better understand their scalability, each program is profiled to get more details on the overhead of instrumentation, instruction composition and other runtime overhead. The results are shown in Figure 12. In Figure 12, each program has four bars for each thread configuration (from 1 thread to 32 threads). From left to right, they are PICO-ST, HST, PST and PST-REMAP. Because of some implementation issues, PST-REMAP currently cannot run on all of the applications. So, for some programs, the data for PST-REMAP is missing.

Each bar is further broken down into several components. The execution time of the basic QEMU functions is shown as the "*native*" component. The "*exclusive*" component is the time spent on the locking support. It includes the time spent on enforcing the exclusion for `SC` in HST and PST, and locking overhead on the hash table in PICO-ST. The "*instrument*" component comes mainly from `store` instrumentation in HST and `LL/SC` emulation in PICO-ST. The "*mprotect*" component is the time spent on system calls in `LL/SC` for PST. It can be seen that, comparing to PICO-ST whose major overheads are in store instrumentation and synchronization, *mprotect* overhead from system calls in PST has largely offset the benefit of eliminating the store instrumentation for PST.

We have also measured the performance of HTM-based

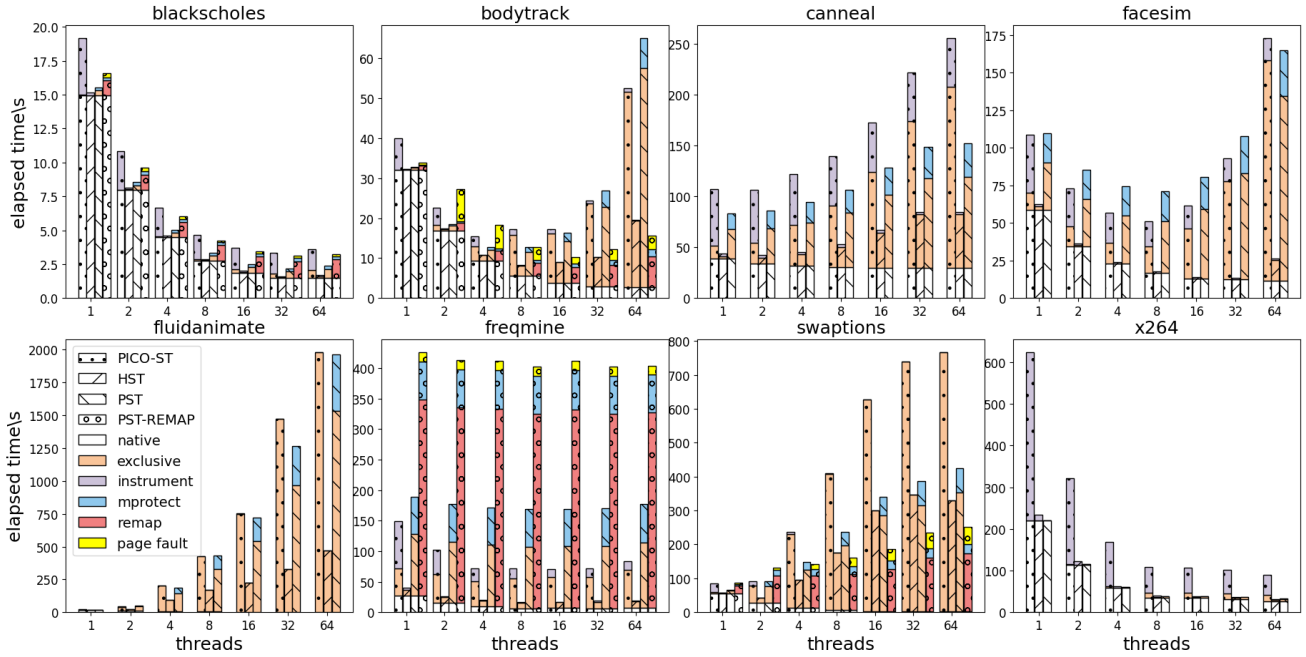

Fig. 12: Overhead breakups for selected benchmarks of PARSEC.

Application	LL/SC(%)	store(%)
blackscholes	0.00506	6.56
bodytrack	0.00142	5.38
canneal	0.09168	20.42
facesim	0.08322	13.86
fluidanimate	0.08374	5.26
freqmine	0.22046	19.46
swaptions	0.01759	6.70
x264	0.00003	1.00
geomean	0.01207	7.15

TABLE I: The dynamic counts (in percentages) of LL/SC and store in PARSEC with 4 threads and *simlarge* input using QEMU-4.1.

solutions, and the results are shown in Figure 11. It can be seen that PICO-HTM outperforms HST-HTM when there is only a small number of threads that do not stress the HTM supporting hardware, and it is not burdened by the store instrumentation overhead. However, as the thread number goes beyond 8 threads, the PICO-HTM will crash frequently, while HST-HTM can still scale up to 32 threads with good performance.

Overall, the best solution is HST-based schemes. They are scalable with good performance, and are portable to any platform without requiring special hardware support. Comparing to the fast but slightly incorrect PICO-CAS scheme, it introduces an overhead of 2.9% and can be up to 555.0% when the number of threads increases. The overhead increases dramatically, especially for applications with intensive atomic operations. Comparing to the PICO-ST scheme, it has a min/max speedup of 1.25x and 3.21x, respectively. The geometric mean speedup of all applications is 2.03x.

2) Overhead Analysis: Generally, to correctly emulate LL/SC, it it crucial to detect the interference among LL, SC and store. The software-based solutions, such as HST, implement it by instrumenting memory stores, while hardware-based solutions try to mitigate the runtime overhead with the help of the hardware such as MMU and system support such as *mprotect* and page-fault handler. To understand the trade-off between the hardware-related overhead vs. software-related overhead such as instrumentation overhead, a program profiling analysis is conducted and the result is shown in Table I.

PICO-ST carefully inserts locks in both store and LL/SC to avoid race condition, but introduces large instrumentation overhead as shown in *blackscholes* and *x264*. If we only instrument the stores and leave alone LL/SC, PICO-ST is slowed by 20%∼45%, which indicates that the instrumentation overhead comes mainly from store instrumentation. This can be explained by the amount of store and LL/SC shown in Table I. The store instructions are 88x∼3000x more frequent than LL/SC.

As HST implements lock-free hash table at the IR level, it introduces less than 5% overhead while *helper_functions* can introduce about 20%∼45% overall overhead. That explains why HST has a better performance than PICO-ST in most cases. Comparing to HST, PST is an aggressive strategy that tries to shift the overhead from store to LL/SC, i.e. using less frequent system calls vs. instrumenting a large number of store instructions. However, Figure 12 shows that PST is actually worse than HST. This is because the aggregated system call overheads far exceed the aggregated instrumentation overheads.

HST-HTM uses HTM to support strong atomicity. However, PST cannot take advantage of HTM because system calls inside an HTM transaction will cause it to abort. Remapping a page to another location is a better alternative for PST. It can be seen that, for *blackscholes*, *bodytrack* and *swaptions*, PST-REMAP can benefit from the page remapping. But in *freqmine*, the remapping overhead is more dominant, which is

another evidence that careful trade-offs is essential for good performance.

Another major overhead is supporting *exclusive* execution for critical code sections. All of the schemes avoid race conditions by utilizing fine-grained locking, or suspending all the other threads when one thread enters a critical code section, to support exclusive execution. The percentages of `LL/SC` vary and they are used in different scenarios, so their overheads can vary. In *blackscholes* and *x264*, there are very few atomic instructions so they have good scalability. However, for *bodytrack* and *facesim*, the execution time shows a "U" shape as the number of threads increases. As the programs scale up beyond 8 threads the synchronization overheads, such as thread barriers, become more dominant. For programs such as *canneal*, *fluidanimate*, *freqmine* and *swaptions*, there are more extensive atomic instructions (see Table I), which cause such overheads to increase as the number of threads increases.

Store instructions to the same page of the atomic variable called "false sharing" results in false alarm and therefore can hurt the performance of PST. False alarm grows from 0.2% to 17% when threads increase from 2 to 64. Take *bodytrack* as an example, the amount of false sharing grows from 0.29% to 16.57% as thread number increase from 1 to 64, respectively. Consequently, its execution time also shows a "U" shape as the number of threads increases.

V. Related Work

Synchronization is one of the most critical parts when extending DBTs to multiple threads. PQEMU [8] proposes a weak atomicity model that protects `LL/SC` by a host *mutex*. It serializes the `LL` and `SC` instructions that have the same atomic variable, while the impact of `store` is ignored. COREMU [9] uses multi-word compare-and-swap [23] to handle multi-processor synchronization with lightweight memory transactions, which could still suffer the ABA problem. PICO [10] emulates atomic instructions by instrumenting stores (in *PICO-ST*) or using HTM (in *PICO-HTM*). DQEMU [24] implements a hierarchical synchronization scheme, and a distributed share-memory protocol is used for exclusive accesses among multiple threads on a multi-node, multi-core system.

Rigo [25] has proposed a soft-MMU based scheme similar to PST. It extends the soft-MMU of QEMU to detect conflicts on atomic variables during address translation. It introduces little additional overhead compared to the system-mode emulation that has soft-MMU support. But it slows down the user mode to a larger extent because soft-MMU is already a large portion of the overhead in user mode. Hui Gao [26] and Maged M. Michael [27] present an algorithm to implement the semantics of `LL/SC` using `CAS`. It makes the synchronization variables free from the ABA problem. But, it is based on an strong assumption that these variables can only be modified by atomic but not normal writes.

Even though there are many schemes proposed that can handle `LL/SC` correctly, they either lead to large overheads or need additional hardware support. Consequently, QEMU decides to use the fastest way at the cost of some correctness. Beside QEMU, there are many DBTs that need to deal with `LL/SC` translation, such as Dolphin [28], TinyEmu [29] and ArcEm [30]. However, they either emulate one thread at a time in a round-robin manner, or ignore the ABA problem.

VI. Discussion

a) Optimization using Intel MPK: Theoretically, Intel MPK can improve the performance with new hardware support, since it supports *thread-local* control of the page protection on groups of pages without requiring changes to the global page tables [31]. In this way, different permissions can be set to the threads that are running `LL/SC` instructions concurrently without the need to switch to the kernel-mode and suspend all other threads. However, synchronization could be a challenge since waiting for other threads to set up the protection introduced large overheads. Due to the limited number of protection keys available (only 16 keys on recent Intel hardware), it is challenging to support multi-threaded applications with a large number of threads.

b) Rule-based Code Trasnlation: Rule-based code translation improves performance [32]. Most of `LL/SC`s are generated automatically by compilers, so there is a fixed pattern that can be recognized. If so, replacing the `LL/SC` pairs with a C call to the standard atomic builtin function [33] avoids large overheads and is free from ABA problem.

VII. Conclusion

In this paper, we propose two new schemes, HST and PST, to address the ABA problem [13] that may happen when emulating atomic instructions from a RISC guest machine to a CISC host machine. As described in TABLE II, existing Pico-CAS is an incomplete and incorrect scheme, while PICO-ST is correct but with heavy overhead. Our proposed HST is faster, fully correct, and portable. HST-WEAK and HST-HTM can deliver better performance, while HST-WEAK sacrificing strong atomicity and HST-HTM portability. PST is a page protection based approach to avoid instrumenting stores. However, its system call overheads far exceed store instrumentation overheads. PST-REMAP allows concurrent and different page privileges on the same page to mitigate PST overhead. Prototypes of these schemes have been implemented on QEMU-4.1. Experimental results on a set of benchmarks in PARSEC show that all of the proposed schemes can provide correct emulation and, the best scheme, HST, can achieve an average of 2.03× speedup

Approaches	Speed	Atomicity	Portability
HST	fast	strong	portable
HST-WEAK	fast	weak	portable
HST-HTM	fast	strong	HTM
PST	slow	strong	portable
PST-remap	varies	strong	portable
Pico-ST	slow	strong	portable
Pico-CAS	fast	incorrect	portable
Pico-HTM	fast	incorrect	HTM

TABLE II: A summary of proposed schemes showing trade-off between atomicity, efficiency, and portability.

over the best existing software-based scheme with a similarly capability.

ACKNOWLEDGMENT

This work is partially supported by the National Key Research and Development Program of China (2018YFB1003405), the National Natural Science Foundation of China (61702286), the Natural Science Foundation of Tianjin, China (18JCYBJC15600), the CERNET Innovation Project (NGII20190514), and a faculty startup funding of the University of Georgia. We would also like to thank anonymous reviewers for their constructive comments and suggestions.

APPENDIX

A. Abstract

We provide source codes of all the approaches mentioned in the paper, including HST, HST-weak, PST, PST-remap, HST-HTM, Pico-HTM, and Pico-ST, which are already intergrated in the QEMU emulator version 4.1.0. For the benchmarks, we provide source codes of our Arm lock-free stack for correctness evaluation. As for the PARSEC 3.0 benchmark, you can either compile from source or download the configured PARSEC with compiled Arm binary directly. To run the experiments, we provide linux shell scripts for dependency installation, correctness and performance evaluation, as well as data collection and result analysis.

B. Artifact check-list

- **Program: QEMU-4.1.0 with various approaches integrated; PARSEC 3.0; Arm lock-free stack**
- **Data set: We use *simlarge* input with PARSEC**
- **Run-time environment: linux-5.4, Ubuntu-20.04LTS**
- **Hardware: We recommend using x64 processors suppporting more than 40 threads. Note that verifying HTM based solutions needs Intel TSX.**
- **Disk space required: 40GB**
- **Time needed to prepare workflow: Less than an hour if you use PARSEC binaries instead of compiling from source.**
- **Time needed to complete experiments: Around 3 days**
- **Publicly Link: https://github.com/NKU-EmbeddedSystem/ABA-LLSC**
- **Code licenses : The GNU General Public License (GPL)**
- **Experiment Workflow: Linux shell scripts**

C. Installation

In this section, we prepare executable binaries for QEMU with Arm guest, PARSEC in Arm, and Arm lock-free stack.

a) Prepare PARSEC-3.0 for Arm: If you want to compile PARSEC from the very beginning, you can download the source of PARSEC and follow the instructions here https://github.com/arm-university/arm-gem5-rsk/wiki to enable cross-compiling to Arm.

However, we strongly recommend you to use our configured PARSEC to avoid trivial setup of cross-compile, which is not our focus here.

```
git clone https://github.com/NKU-EmbeddedSystem/
    parsec-3.0-arm.git
```

b) Fetching Code: Here we fetch the source codes of QEMU and lock-free stack.

```
git clone https://github.com/NKU-EmbeddedSystem/ABA-
    LLSC.git
cd ABA-LLSC
git clone https://github.com/NKU-EmbeddedSystem/QEMU
    -ABA.git
git clone https://github.com/NKU-EmbeddedSystem/lock
    -free-stack-arm-asm.git
sudo bash installDep.sh
```

c) Build: To build portable approaches,

```
cd ABA-LLSC
bash build.sh
```

If Intel TSX is supported, you may also build HTM based approaches,

```
cd ABA-LLSC
bash build-HTM.sh
```

D. Experiment workflow

In the workflow we evaluate the **correctness** and **performance**. **Make sure the binaries have been compiled.**

a) Correctness: We use an arm lock-free stack to evaluate if the solution suffers ABA problem. If the solution suffers ABA problem, it can't pass the test.

```
cd ABA-LLSC/experiment
bash correctness.sh
```

If TSX supported

```
bash correctness-HTM.sh
```

b) Performance: We use parsec-3.0 benchmark to profile performance. Configure parsec and then we can start the script. It may take about 3 days, so we'd better run it in backend.

```
cd path-to-parsec
source env.sh
cd path-to/ABA-LLSC/experiment
nohup bash scalibility.sh & # run the script in
    backend
```

If TSX is supported, you may also evaluate HST-HTM.

```
cd path-to-parsec
source env.sh
cd path-to/ABA-LLSC/experiment
nohup bash scalibility-HTM.sh
```

Pico-HTM leads to potential livelock and PST-remap can only run four applications in parsec 3.0, so they're **not involved** in performance scripts. PST-remap only supports blackscholes, bodytrack, freqmine, and swaptions in parsec benchmarks.

You can also run them manually using Linux commands.

```
cd path-to-parsec
source env.sh
cd path-to/ABA-LLSC/experiment
parsecmgmt -a run -p <program> -i simlarge -n <
    thread number> -s "time $(pwd)/../bin/$1"
# Example run with blackscholes, 4 threads:
# parsecmgmt -a run -p blackscholes -i simlarge -n 4
    -s "time $(pwd)/../bin/Pico-HTM"
```

E. Evaluation and expected result

1) Correctness: When testing Pico-CAS, it shows "Stack is smashed" in logs while other solutions show "ABA problem test passed!".

2) Performance: We provide scripts to calculate speedup normalized to single thread execution time of Pico-CAS of the solutions.

```
cd experiment
python3 ./speedup.py
cat speedup.csv
```

You can draw figure using draw.py, and the result is saved as speedup.pdf.

```
python3 ./draw.py
```

REFERENCES

[1] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu, "Hermes: a fast cross-isa binary translator with post-optimization," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 246–256. [Online]. Available: https://doi.acm.org/10.1109/CGO.2015.7054204

[2] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere, "Evaluation of dynamic binary translation techniques for full system virtualisation on armv7-a," *Journal of Systems Architecture*, vol. 65, pp. 30–45, 2016. [Online]. Available: https://doi.acm.org/10.1016/j.sysarc.2016.03.001

[3] E. Cota and L. Carloni, "Cross-isa machine instrumentation using fast and scalable dynamic binary translation," 04 2019, pp. 74–87. [Online]. Available: https://doi.acm.org/10.1145/3313808.3313811

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005. [Online]. Available: https://doi.acm.org/10.1145/1065010.1065034

[5] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[6] T. Garnett, "Dynamic optimization if ia-32 applications under dynamorio," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[7] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996.

[8] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung, "Pqemu: A parallel system emulator based on qemu," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 276–283. [Online]. Available: https://doi.acm.org/10.1109/ICPADS.2011.102

[9] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "Coremu: a scalable and portable parallel full-system emulator," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 213–222, 2011. [Online]. Available: https://doi.acm.org/10.1145/1941553.1941583

[10] E. G. Cota, P. Bonzini, A. Bennée, and L. P. Carloni, "Cross-isa machine emulation for multicores," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 210–220. [Online]. Available: https://doi.acm.org/10.1109/CGO.2017.7863741

[11] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: defending against memory consistency model mismatches in heterogeneous architectures," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 388–400, 2016. [Online]. Available: https://doi.acm.org/10.1145/2749469.2750378

[12] C. Blundell, E. C. Lewis, and M. Martin, "Deconstructing transactional semantics: The subtleties of atomicity," in *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*. Citeseer, 2005, pp. 48–55.

[14] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01. New York, NY, USA: ACM, 2001, pp. 134–143. [Online]. Available: http://doi.acm.org/10.1145/378580.378611

[13] Wikipedia, "The aba problem," accessed 21 November 2019. https://www.brainkart.com/article/The-ABA-Problem_9525/.

[15] M. M. Michael, "Hazard pointers: safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, June 2004.

[16] Wikipedia, "Load-link/store-conditional," accessed 21 November 2019. https://en.wikipedia.org/wiki/Load-link/store-conditional.

[17] C. Wellons, "C11 lock-free stack," 2014(Accessed 2020). [Online]. Available: https://nullprogram.com/blog/2014/09/02/

[18] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, "Efficient and scalable cross-isa virtualization of hardware transactional memory," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 107–120. [Online]. Available: https://doi.acm.org/10.1145/3368826.3377919

[19] M. Abadi, T. Harris, and M. Mehrara, "Transactional memory with strong atomicity using off-the-shelf memory protection hardware," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009, pp. 185–196. [Online]. Available: https://doi.acm.org/10.1145/1504176.1504203

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81. [Online]. Available: https://doi.acm.org/10.1145/1454115.1454128

[21] L. manual page, "mremap," accessed 21 November 2019. https://man7.org/linux/man-pages/man2/mremap.2.html.

[22] Alibaba, "Alibaba cloud server description," Accessed 2020. [Online]. Available: https://help.aliyun.com/document_detail/108491.html?spm=a2c4g.11186623.6.593.2313b1478Qjd8u#section-wiv-kqq-u7o

[23] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," *Lecture Notes in Computer Science Distributed Computing*, p. 265–279, 2002.

[24] Z. Zhao, Z. Jiang, X. Liu, X. Gong, W. Wang, and P.-C. Yew, "Dqemu: A scalable emulator with retargetable dbt on distributed platforms," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3404397.3404403

[25] A. Rigo, A. Spyridakis, and D. Raho, "Atomic instruction translation towards a multi-threaded qemu," in *ECMS*, 2016. [Online]. Available: https://doi.acm.org/10.7148/2016-0587

[26] H. Gao, Y. Fu, and W. H. Hesselink, "Practical lock-free implementation of ll/sc using only pointer-size cas," in *2009 First International Conference on Information Science and Engineering*. IEEE, 2009, pp. 320–323. [Online]. Available: https://doi.acm.org/10.1109/ICISE.2009.841

[27] M. M. Michael, "Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas," in *Distributed Computing*, R. Guerraoui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 144–158. [Online]. Available: https://doi.acm.org/10.1007/978-3-540-30186-8_11

[28] Dolphin Emulator Project, "A GameCube and Wii emulator," 2020, https://dolphin-emu.org.

[29] T. E. Project, "A system emulator for the risc-v and x86 architectures," Accessed 2020, https://bellard.org/tinyemu/.

[30] A. A. Project, "A portable open-source emulator for the acorn archimedes," Accessed 2020, http://arcem.sourceforge.net/.

[31] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 241–254.

[32] J. Jiang, R. Dong, Z. Zhou, C. Song, W. Wang, P. C. Yew, and W. Zhang, "More with less – deriving more translation rules with less training data for dbts using parameterization," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 415–426.

[33] "Built-in functions for atomic memory access," Accessed 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html