Fast and Correct Load-Link/Store-Conditional Instruction Handling in DBT Systems

Abstract

Dynamic Binary Translation (DBT) requires the implementation of load-link/store-conditional (LL/SC) primitives for guest systems that rely on this form of synchronization. When targeting e.g. x86 host systems, LL/SC guest instructions are typically emulated using atomic Compare-and-Swap (CAS) instructions on the host. Whilst this direct mapping is efficient, this approach is problematic due to subtle differences between LL/SC and CAS semantics. In this paper, we demonstrate that this is a real problem, and we provide code examples that fail to execute correctly on QEMU and a commercial DBT system, which both use the CAS approach to LL/SC emulation. We then develop two novel and provably correct LL/SC emulation schemes: (1) A purely software based scheme, which uses the DBT system's page translation cache for correctly selecting between fast, but unsynchronized, and slow, but fully synchronized memory accesses, and (2) a hardware accelerated scheme that leverages hardware transactional memory (HTM) provided by the host. We have implemented these two schemes in the Synopsys DesignWare® ARC® nSIM DBT system, and we evaluate our implementations against full applications, and targeted micro-benchmarks. We demonstrate that our novel schemes are not only correct, but also deliver competitive performance on-par or better than the widely used, but broken CAS scheme.

1 Introduction

Dynamic Binary Translation (DBT) is a widely used technique for on-the-fly translation and execution of binary code for a guest Instruction Set Architecture (ISA), on a host machine with a different native ISA. DBT has many uses, including cross-platform virtualisation for the migration of legacy applications to different hardware platforms (e.g. Apple Rosetta, and IBM PowerVM Lx86, both based on Transitives's QuickTransit, or HP Aries) and the provision of virtual platforms for convenient software development for embedded systems (e.g. OVPsim by Imperas).

A popular DBT system is QEMU [2], which has been ported to support many different guest and host architecture pairs, e.g. Arm v7A/v8A on Intel x86, or PowerPC on Arm. Due to its open-source nature, QEMU is often regarded as a de-facto standard DBT system, but there exist many other proprietary systems such as Wabi, the Intel IA-32 Execution Layer, or the Transmeta Code Morphing software.

Atomic instructions are fundamental to multi-core execution, where shared memory is used for synchronization purposes. Complex Instruction Set Computer (CISC) architectures typically provide various *read-modify-write* instructions, which perform multiple memory accesses (usually to the same memory address) with atomic behavior. A prominent example of these instructions is the Compare-and-Swap (CAS) instruction. A CAS instruction atomically updates a memory location only if the current value at the location is equal to a particular expected value. The semantics of the CAS instruction is shown in Figure 1. For example, Intel's x86 processors provide the `CMPXCHG` instruction to implement *compare-and-swap* semantics.

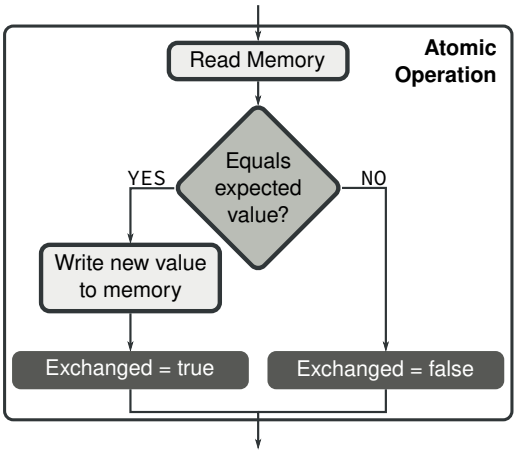

Figure 1: The compare-and-swap instruction atomically reads from a memory address, and compares the *current value* with an *expected value*. If these values are equal, then a *new value* is written to memory. The instruction indicates (usually through a return register or flag) whether or not the *value* was successfully written.

The Reduced Instruction Set Computer (RISC) paradigm avoids complex atomic *read-modify-write* instructions by dividing these operations into distinct *read* and *write* instructions. In particular, *load-link* (LL) and *store-conditional* (SC) instructions are used, and the operation of these instructions is shown in Figure 2. LL and SC instructions typically operate in pairs, and require some kind of loop to retry the operation if a memory conflict is detected.

A *load-link* (LL) instruction performs a memory read, and internally registers the memory location for *exclusive* access. A *store-conditional* (SC) instruction performs a memory write if, and only if, there has been no write to the memory location since the previous *load-link* instruction. Among competing *store-conditional* instructions only one can succeed, and unsuccessful competitors are required to repeat the entire LL/SC sequence.

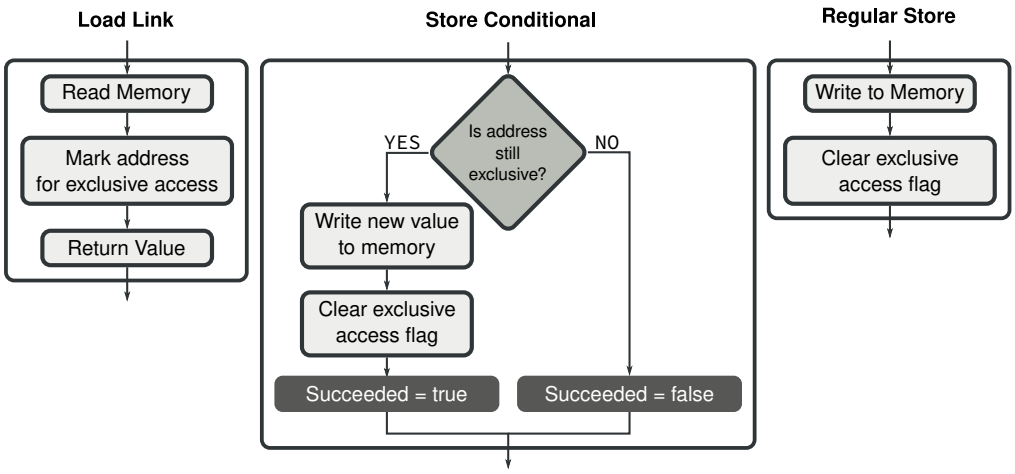

Figure 2: A load-link instruction reads memory, and internally marks that memory for exclusive access. A store-conditional instruction checks to see if a write has occurred by observing if the memory is still marked for exclusive access. If the memory has been written to, the store-conditional fails, otherwise the new value is written and the instruction succeeds. A regular store instruction clears the exclusive access marker.

For DBT systems, a **key challenge** is how to emulate guest LL/SC instructions on hosts that only support CAS instructions. Since LL/SC linkage should be broken by any write to memory, the efficiency of detecting intervening memory writes becomes crucial for the overall performance. A naïve approach would synchronize all store instructions, using critical sections to enforce atomicity when executing the instruction. However, such a naïve scheme would incur a high runtime performance penalty.

A better approach to detect intervening stores efficiently is to use a *compare-and-swap* instruction. In this approach, the guest *load-link* instruction reads memory, and stores the value of the memory location in an internal data structure. Then, the guest *store-conditional* instruction uses this value as the *expected* parameter of the host CAS instruction. If the value of the memory location has not changed, then the *store-conditional* can succeed.

Unfortunately, although fast, this approach does not preserve the full semantics of LL/SC instructions, and in particular suffers from the ABA problem, i.e. a memory location is changed from value A to B and back to A [6]. In this scenario, two interfering writes to a single memory location have occurred, and the *store-conditional* instruction should fail. However, since the *same value* has been written back to memory, this goes unnoticed by the instruction, and it incorrectly succeeds. We refer to this broken approach as the *CAS approximation*, as this emulation strategy only *approximates* correct LL/SC semantics.

1.1 Motivating Example

Figure 3 shows how the *CAS approximation* implements the operation of the *load-link* and *store-conditional* instructions.

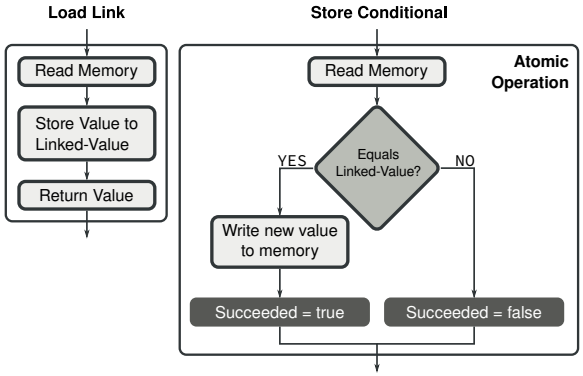

Figure 3: An example of how compare-and-swap semantics can be used to approximate load-link/store-conditional semantics.

The *load-link* instruction records the value of memory (into `linked-value`) at the given address, and returns it as usual. The *store-conditional* instruction performs a compare-and-swap on the guest memory, by comparing the current memory value to `linked-value` (from the load-link instruction), and swapping in the new value if the old values match. If the CAS instruction performed the swap, then the *store-conditional* is deemed to have been successful.

1.1.1 Trivial Broken Example

The following sequence of events describes a possible interleaving of guest instructions, which will cause LL/SC instructions implemented with the *CAS approximation* to generate incorrect results. The sequence of events are:

- **t0:** *Core 1* performs a *load-link* instruction, marking

```
1 ElementType *pop(Stack *stack) {
2   ElementType *originalTop, *newTop;
3   do {
4     originalTop = LoadLink(stack->top);
5     newTop      = originalTop->next;
6   } while (StoreConditional(stack->top,
        newTop) == FAILED);
7   return originalTop;
8 }
```

Figure 4: Implementation of a lock-free stack `pop` operation.

memory address `0x1000` for exclusive access. The value returned from memory is #1.

- **t1:** *Core 2* writes the value #2 to memory, using a regular store instruction.
- **t2:** *Core 2* writes the value #1 to memory, again using a regular store instruction.
- **t3:** *Core 1* performs a *store-conditional* instruction, and since the value in memory is the same from when it was read in **t0**, **incorrectly** performs the actual store, and returns a `SUCCESS` result.

The *CAS approximation* approach violates the semantics of the LL/SC pair at **t3**, as from the point-of-view of the *store-conditional* instruction, the value in memory has not changed, and so the instruction **incorrectly** assumes no changes were made, and hence succeeds. However, this assumption is **wrong**, since two interfering writes (at **t1** and **t2**) have been performed, and thus should cause the *store-conditional* instruction to return a `FAILED` result.

We constructed a simple program to test this behavior on both a real 64-bit Arm machine, and an emulation of a 64-bit Arm machine using QEMU. Our analysis found that the test binary behaved as expected on the real Arm machine, but incorrectly performed the *store-conditional* in QEMU. We configured the commercial Synopsys DesignWare® ARC® nSIM DBT to use the CAS approach and verified it too exhibited the incorrect behavior.

1.1.2 ABA Problem in Lock-free Data Structures

In practice, the trivial broken example described previously can lead to the ABA problem appearing in multi-core data structures that have collaborating operations [6], and that utilize LL/SC as the synchronization primitives.

For example, a lock-free implementation of a linked-list based stack loads its *shared* `top` pointer (the *load-link*), prepares the desired modifications to that data structure, and only updates the shared data provided there has been no concurrent update to the shared data since the previous load (the *store-conditional*). A typical implementation of the `pop` operation, using LL/SC semantics is given in Figure 4.

Table 1 shows a possible interleaving of operations on a lock-free stack, which leads to incorrect behavior when the *CAS approximation* is used to emulate LL/SC instructions. The sequence of events are as follows:

- **t0:** *Core 1* starts executing a `pop()` operation.
- **t1:** *Core 1* performs a *load-link* on the stack top pointer, and stores the resulting element in `originalTop`. In this case, object `A`.
- **t2:** *Core 1* now resolves the `next` pointer, which points to object `B`, and stores it in `newTop`.
- **t3:** *Core 2* now pops object `A` from the stack.
- **t4:** *Core 2* then pops object `B` from the stack.
- **t5:** *Core 2* finally pushes object `A` back on the stack.
- **t6:** At this point, *Core 1* attempts to update the top of the stack pointer, using a *store-conditional* instruction to write back `newTop` (which is object `B`).

Now, with correct LL/SC semantics, the *store-conditional* instruction will return `FAILED`, to indicate that an intervening write has occurred (even though the value in memory has not changed). However, if the *CAS approximation* is used to implement the *store-conditional* instruction, then it will not detect that any writes to `stack->top` have occurred (since the *load-link* in **t1**), and the top pointer will be incorrectly updated to `B`, since the current value of the pointer matches the expected value `A` (i.e. `originalTop == stack->top`). This can lead to undefined behavior, as the correct stack state has been lost. For example, stack element `B` can now point to reused, or invalid/freed memory.

Since the *CAS approximation* detects modifications to memory only through changes of values, other techniques must be used to prevent the ABA problem and guarantee correctness [7, 13, 15].

From this example, it can be seen that it is crucial for DBT systems to correctly emulate LL/SC instructions. Incorrect emulation of these instructions will lead to incorrect behavior of guest programs.

Similar to our QEMU experiment to detect incorrect behavior, we also constructed a test program of an LL/SC based implementation of a lock-free stack. We discovered that in QEMU, interleaving stack accesses as depicted in Table 1 results in stack corruption.

This paper introduces a novel solution to the LL/SC emulation problem and makes the following **key contributions**:

- We show that the use of *compare-and-swap* instructions is insufficient for capturing the full semantics of *load-link*/*store-conditional* instructions, and demonstrate that state-of-the-art emulators (such as QEMU) behave incorrectly.
- We introduce a provably correct scheme for implementing *load-link*/*store-conditional* instructions in a DBT system.

Time	Core 1	Core 2	Stack State
t_0	**Begin `pop()` operation**		✓ `top->A->B->C`
t_1	`originalTop =` **`LoadLink`**`(top) == A`		
t_2	`newTop = originalTop->next == B`		
t_3		`pop() == A`	✓ `top->B->C`
t_4		`pop() == B`	✓ `top->C`
t_5		`push() == A`	✓ `top->A->C`
t_6	**`StoreConditional`**`(top, newTop)` ⇒ **`Compare-and-Swap`**`(top, A, B)`		✓ `top->A->C` ✗ `top->B->?`

Table 1: An example thread interleaving of lock-free stack manipulation exhibiting the *ABA* problem, when the *CAS approximation* is used to implement LL/SC instructions.

- We show that our correct implementation delivers application performance levels comparable to the broken *CAS approximation* scheme.

The remainder of this paper is structured as follows. In Section 2, we describe a number of different approaches for DBT systems to emulate *load-link*/*store-conditional* instructions, and we follow with an evaluation of these schemes in Section 3. Section 4 gives a proof of correctness. Finally, we discuss related work in Section 5, and conclude in Section 6.

2 Schemes for LL/SC Handling

In this section we introduce four schemes for handling LL/SC instructions in DBT systems. These schemes range from a naïve baseline scheme, through to an implementation utilizing hardware transactional memory (HTM).

1) **Naïve Scheme** (Section 2.1): This scheme inserts standard locks around *every* memory instruction for tracking linked addresses, effectively turning memory instructions into critical sections. The scheme is correct, but impractical due to the extensive performance penalty associated with synchronizing on every memory access.
2) **Broken: Compare-and-Swap-based Scheme** (Section 2.2): This scheme is used in state-of-the-art DBT systems such as QEMU. The scheme maps guest LL/SC instructions onto the host system's *compare-and-swap* instructions, resulting in high performance. However, as described in Section 1.1, it violates LL/SC semantics.
3) **Correct: Software-based Scheme** (Section 2.3): This scheme utilizes facilities available in the DBT system to efficiently manage linked memory addresses, by taking advantage of the *page translation cache*.
4) **Correct: Hardware Transactional Memory** (Section 2.4): This scheme exploits the *hardware transactional memory* facilities available on modern processors to efficiently detect conflicting memory accesses in LL/SC pairs.

In general, the handling of LL/SC instructions in DBT systems closely follows the observed hardware implementation of these instructions. Each *load-link* instruction creates a CPU-local record of the *linked* memory location (i.e. storing the memory address in a hidden register). Then, the system monitors all writes to the same memory location. If a write is detected, the linkage of *all* CPUs is broken, to ensure that no future *store-conditional* instruction targeting the same memory location can succeed.

Emulating *store-conditional* instructions requires verifying that the local linkage is still valid. If so, the *store-conditional* can succeed, and invalidate the linkage of other CPUs for the same memory location atomically.

Since emulating an SC instruction comprises several checks and updates that must appear to be atomic, this emulation must be properly synchronized with the emulation of other SC and LL instructions. Furthermore, concurrent *regular* stores that interleave between the LL/SC pairs have to be detected so that future SC instructions cannot succeed. Detecting interleaving regular stores is the main challenge in the efficient emulation of LL/SC instructions.

2.1 Naïve: Lock Every Memory Access

A naïve implementation of LL/SC instructions simply guards all memory operations to the corresponding memory address with the same lock. Conceptually, a global lock can be used to guarantee mutual exclusion of all LL/SC and regular stores. In practice, more fine-grained locking can be used to improve performance, by allowing independent memory locations to be accessed concurrently.

This emulation scheme is presented in Figure 5. A *load-link* instruction enters a critical section, and creates a local linkage under mutual exclusion with respect to *store-conditional*, and regular stores to the same locations.

The *store-conditional* instruction checks the local linkage, and if it matches, then it performs the actual write to the memory address. The linkages of other CPUs corresponding to the same memory address are also invalidated, so that no future *store-conditional* can succeed. The emulation of a regular store simply invalidates the linkage on all CPUs corresponding to the same memory address. The actual write is performed unconditionally.

Although simple, this scheme suffers a significant perfor-

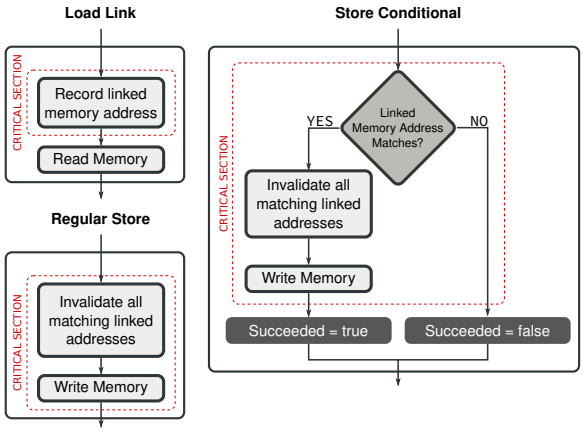

Figure 5: A naïve scheme to emulate *load-link* and *store-conditional* instructions. Critical sections are used to serialize all (including regular) memory accesses.

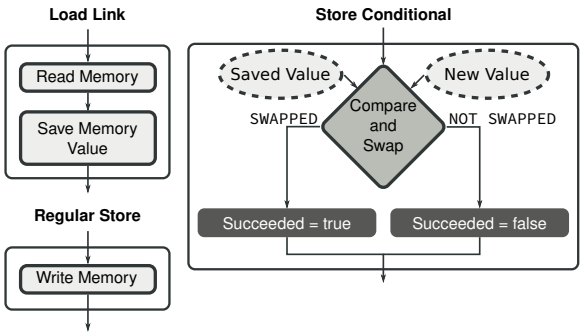

Figure 6: CAS approximation of LL/SC.

mance penalty, due to the critical sections causing a slowdown of all regular store instructions. In this scheme, lock acquisition has to be performed by every regular store, even those that do not access memory locations currently in use by LL/SC pairs. Since in a typical application only a small portion of instructions are LL/SC, the majority of regular stores are slowed down unnecessarily.

2.2 Broken: Using CAS Style Semantics

As previously described, a *compare-and-swap* instruction can be used to approximate the semantics of *load-link*/*store-conditional* pairs. In this scheme, the LL/SC linkage comprises not only the memory *address*, but also the memory *value* read by the LL instruction.

Emulating the *load-link* instructions saves the linked address as well as the linked value (i.e. the result of the corresponding *read*). Then, emulating the *store-conditional* instruction uses the linked value for comparison with the current value of the linked memory location using the CAS instruction. If the current value of the memory at the previously stored linked address does not match the linked value saved from the previous LL instruction, an interleaving memory access has been detected. Since intervening writes to the memory location are detected by changes in *memory value*, no linkage invalidation of other CPUs is required. Similarly, regular stores can proceed without any need of linkage invalidation or synchronization.

This scheme offers great performance, since the emulation of LL/SC and regular stores do not need to synchronize at all. Furthermore, the *compare-and-swap* instruction is a well established synchronization mechanism, and thus its performance is optimized by the host hardware. However, as we have demonstrated, the CAS scheme only approximates the semantics of LL/SC instructions and in particular, utilizing the CAS scheme for this purpose can cause the emulation of guest programs to break.

2.3 Correct Software-Only Scheme

This scheme improves upon the approach taken by the naïve scheme. The **key idea** is to slow down only those regular stores that access memory locations that are currently being used by LL/SC instructions. To achieve this, we take advantage of a software Page Translation Cache (PTC), which is used in the DBT system to speed up the translation of *guest* virtual addresses to *host* virtual addresses. Emulated memory accesses can query this cache to avoid a costly full memory address translation, which may incur walking guest page tables, and resolving guest physical pages to corresponding host virtual pages.

Upon a hit in the cache, the emulation can use a fast-path version of the instruction. For regular stores, fast-path version involves no synchronization with LL/SC instructions. Note that each core has its own private PTC, and that there exists a number of separate per-core PTCs based on memory access type (e.g. read, write, execute).

This scheme allows regular stores that are not conflicting with LL/SC memory addresses to proceed without any slowdown, by only synchronizing if the Write-PTC query misses in its corresponding cache. To achieve this behavior, emulating a *load-link* instruction involves invalidating the Write-PTC entry for the corresponding memory address. Then, future regular stores will be guaranteed to miss in the cache, and will be forced to synchronize with concurrent LL/SC instructions. In other words, no concurrent regular store can enter the fast-path without invalidating all LL/SC linkages for the corresponding memory address.

A similar approach has been found in the literature [20], although this seems not to have been adopted by industry. However, the authors did not provide enough implementation details to avoid a race condition in the PTC manipulation (see Section 2.3.1). Our implementation avoids this particular issue, and also provides further optimizations that improve the performance and scalability of the scheme.

Time	Core 1	Core 2
t_0	**Regular Store**	**Load-link**
t_1	PTC lookup **hits** in cache	
t_2		Create linkage
t_3		Invalidate PTC
t_4		Read data from memory
t_5	Write data to memory	

Table 2: A particular interleaving of operations that leads to a race-condition in the Software-only scheme.

2.3.1 Page Translation Cache Race Condition

Implementing the Software-only scheme without regard for the ordering of operations between cores leaves room for a race condition to appear. This particular scenario is depicted in Table 2.

In this interleaving, on *Core 1*, the emulation of a regular store performs a successful PTC lookup, and enters the fast-path at time t_0. At this point, *Core 2* performs the full emulation of a *load-link* instruction, i.e. the linkage is established, the corresponding PTC entry is invalidated, and the data is read from memory (t_2 through t_4). Then, back on *Core 1*, the emulation of the regular store actually updates the data in memory, at time t_5.

After this execution, a *store-conditional* instruction on *Core 2* can still succeed, as *Core 1* did not invalidate the underlying linkage. However, the data read by *Core 2* at time t_4 is now out of date, since an interleaving write to the memory address (*Core 1* at t_5) has been missed.

This race-condition manifests itself when PTC invalidations performed by LL instructions happen after a successful PTC lookup by a regular store, but *before* the actual data write by the regular store instruction.

To prevent such behavior, we protect the PTC entry during fast-path execution of regular stores by making a cache line tag annotation. In particular, successful PTC lookups atomically annotate the underlying tag value by setting a bit corresponding to an ongoing fast-path execution. This tag bit can be cleared only after the actual data update has happened. While the tag bit is set, PTC invalidation during LL emulation is blocked.

2.3.2 Optimizations

There are a number of optimizations that can be applied to the software-only scheme, to make further improvements.

Firstly, since the linkage comprises a single word, it can be updated atomically. This means that mutual exclusion is not required for LL emulation. However, operations still have to have a particular order to guarantee correctness. In the case of LL emulation, the data read must happen *after* the PTC invalidation, which must happen after the linkage is created.

The required ordering is depicted in Figure 7 using a red dotted line to indicate a memory fence across which operations cannot be reordered. We also provide a proof of correctness of this scheme in Section 4.

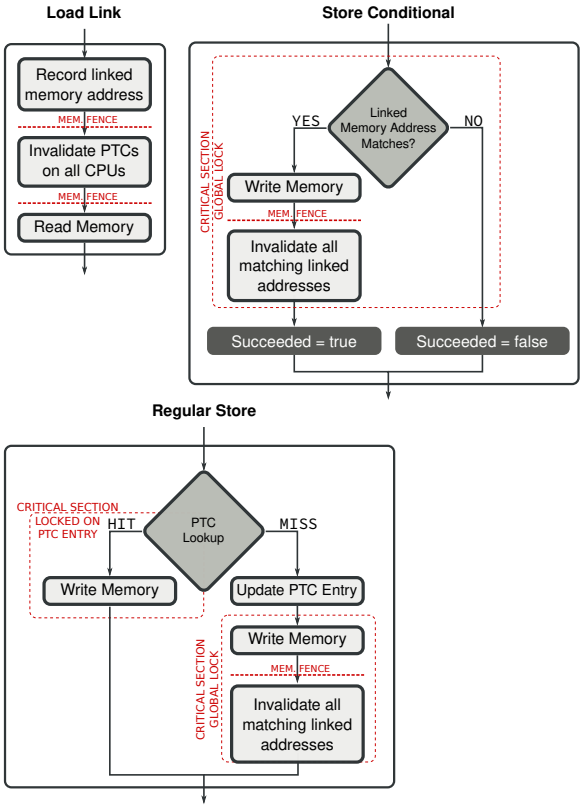

Figure 7: A software-only scheme emulation of LL/SC, utilizing the DBT system's page translation cache.

Secondly, since emulation of LL does not use locks for mutual exclusion, the state of the underlying lock can be used to infer if a concurrent write operation is in flight.

This can be used to immediately fail *store-conditional* instructions without checking whether or not the linkage is present. If the emulation of an SC instruction observes the lock as being already held, then there is already a concurrent update to the memory location in progress, so the SC instruction will fail anyway. In this scenario, the SC instruction fails immediately. In other words, *store-conditional* emulation acquires the underlying lock if and only if it has enough confidence that it is still possible to succeed. Note that the emulation of regular stores *always* acquires the lock, as this write is unconditional.

These optimizations reduce the amount of code that executes within a critical section, and especially in many-core applications, this results in less code having to serialize, thus resulting in much better scalability.

2.4 Correct Scheme Using Hardware TM

Hardware Transactional Memory (HTM) has been used for implementing memory consistency in cross-ISA emulation on multicore systems before [16]. We take inspiration from

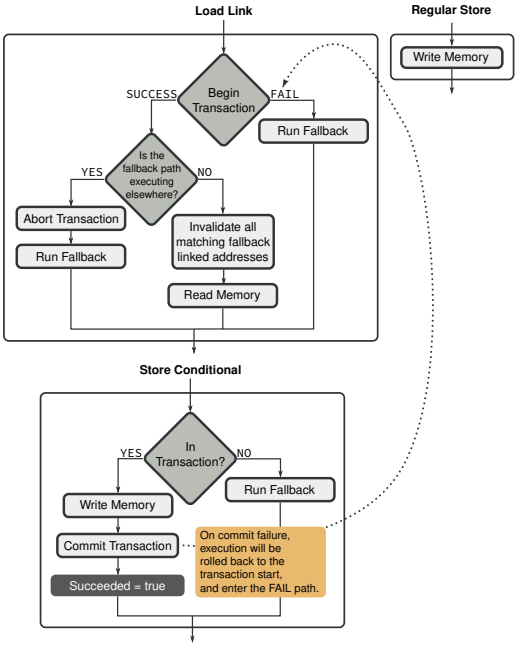

Figure 8: Scheme employing Hardware TM.

this approach, and develop a scheme for implementing LL/SC handling using similar HTM constructs.

HTM allows the execution of a number of memory accesses to be perceived as atomic. This can be exploited for the handling of LL/SC instructions by encapsulating the whole LL/SC sequence in a transaction. In particular, the emulation of a *load-link* begins a transaction (e.g. the `XBEGIN` Intel TSX instruction) and then reads the data. The emulation of a *store-conditional* writes the new data, and then commits the ongoing transaction (e.g. `XCOMMIT`).

Since all memory accesses between LL and SC instructions (inclusive) are part of a transaction, any concurrent transactions (i.e. another LL/SC pair) succeed only if there are no conflicts between any memory accesses. Furthermore, even memory accesses that are not part of a transaction will abort ongoing transactions in the case of a conflict.

This behavior is guaranteed by the strong transactional semantics supported by many modern architectures. As a result, this scheme can avoid any locking or linkage invalidation by relying on transactional semantics to resolve conflicts between concurrent LL/SC pairs, and to detect intervening regular stores.

Using HTM always requires a fallback mechanism to be in place, as HTM is not guaranteed to succeed. In our case, the fallback mechanism is the Software-only scheme, described in Section 2.3.

The emulation of an SC instruction can check if it is executing within a transaction by e.g. using the Intel TSX `XTEST` instruction. For any core, both LL and SC instructions either take the transactional path or the fallback path. However, it is still possible for one core to execute a LL/SC sequence transactionally, while another core concurrently executes LL/SC using the fallback mechanism. Therefore, the transactional path has to correctly interact with fallback path.

Since the transaction is perceived to be atomic, we analyze the interaction from the perspective of the fallback execution, assuming that transactional execution happens instantaneously. The transaction can happen while inside a critical section, during the emulation of a *store-conditional* in the fallback path.

To guarantee correct execution, we use a lock elision technique [19] that aborts (e.g. `XABORT`) the transaction if any cores are currently executing in the fallback path (i.e. within a critical section).

Another scenario is when the transaction happens while the fallback execution emulates instructions between LL and SC pairs. Here, the lock elision technique cannot be used, since there is no lock to elide. Instead, the fallback linkage is invalidated, causing the future fallback SC instruction to fail. This approach allows the transactional execution to continue, to reduce the number of transaction aborts.

3 Evaluation

We compiled applications from EEMBC Multibench [9] and PARSEC [3] benchmark suites for bare-metal parallel execution by linking the binaries against a custom bare-metal `pthread` library. The `pthread` library assigns each application thread to a virtual core. In our evaluation, we use 10-core execution to support eight workload threads for each benchmark. The two remaining cores are necessary to support non-workload threads, e.g. the main application thread that spawns the workload threads.

In addition to the application benchmarks, we constructed our own suite of micro-benchmarks. These are designed to stress the use of LL/SC instructions. The benchmarks vary in the level of congestion in space and time. The micro-benchmarks are described in Table 3 and the details of the host machine used for experimentation are shown in Table 4.

3.1 Key Results: Application Benchmarks

The Multibench suite does not exhibit much change in performance. For most benchmarks, all schemes result in the runtime performance falling within 5% relative to the Naïve scheme. This is due to the infrequent execution of the affected instructions (i.e. LL, SC, and regular store) by these benchmarks. For example, data-parallel benchmarks synchronize using LL/SC instructions only at the beginning and at the end of the execution. In this case, the workload kernel does not contain any synchronization, and thus is not affected by the schemes.

Similar to the Multibench suite, the PARSEC benchmarks do not heavily use LL/SC instructions. However, the use of regular store instructions is much more frequent. This results in the PARSEC suite showing a significant performance

Benchmark	SC %	Description
`space_<x>`	3.84	atomically increment random counter in an array of size `x`
`space_indep`	16.67	atomically increment a thread-private counter
`time_<x>_<y>`	variable	workload loop performs `y` instructions, `x` of which are inside LL/SC sequence
`stack`	2.32	alternate pushing and poping element in a lock-free stack [1]
`prgn`	16.67	generate random numbers by a lock-free random number generator

Table 3: Characterisation of Micro-benchmarks. To indicate heavy use of LL/SC, the percentage of executed SC instructions as a proportion of all executed instructions is displayed. These numbers were generated in single-threaded versions of the benchmarks. In multi-threaded versions, all threads perform the same workload. However, the dynamic SC count can differ as LL/SC contention can force LL/SC sequences to be repeated.

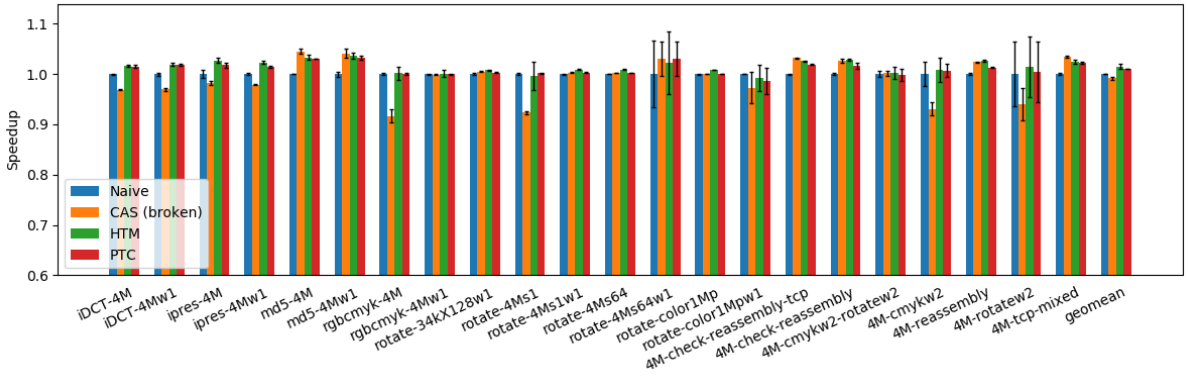

Figure 9: Results for EEMBC Multibench suite with 10 cores.

System	Dell® PowerEdge® R740xd		
Architecture	x86-64	*Model*	Intel® Xeon® Gold 6154
Cores/Threads	36/72	*Frequency*	3 GHz
L1 Cache	I$128kB/D$128kB	*L2 Cache*	1MB
L3 Cache	10 MB	*Memory*	512 GB

Table 4: DBT Host System

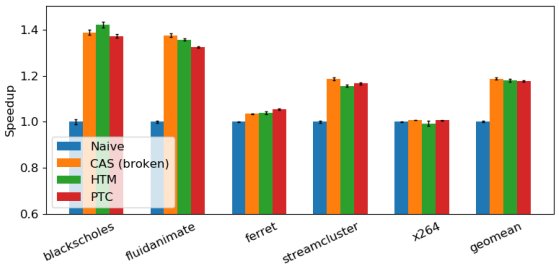

Figure 10: PARSEC suite with 10 cores.

improvement by using any other scheme compared to the Naïve scheme. Since the Naïve scheme incurs a synchronization overhead for all regular stores, its performance degrades compared to the other schemes that do not synchronize independent regular stores. The performance of the other schemes is comparable, with the PTC scheme achieving a speedup of 1.18× over Naïve on average.

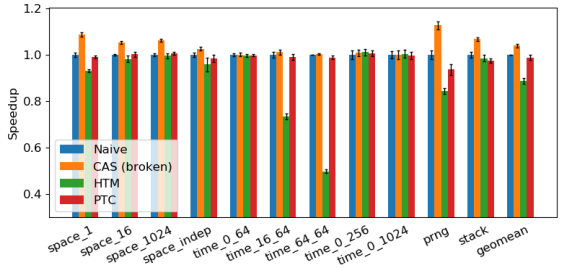

Figure 11: Micro-benchmarks with 1 core.

3.2 Drilling Down: Micro-Benchmarks

These benchmarks exhibit much more frequent use of LL/SC instructions. Therefore, the performance implications of the scheme used is much more pronounced.

First, we evaluate the implementation overhead of each scheme by running a single-core version of the benchmarks. In this version, there is no congestion, as there are no concurrent cores. As a result, all SC instructions succeed and no LL/SC sequence is repeated. The single-core performance results are shown in Figure 11.

The CAS scheme results in the best performance with the average speed-up of 1.04 over the naïve scheme. This is due to implementation simplicity and lack of explicit synchronization. However, this scheme does not preserve the LL/SC semantics.

The performance of the PTC scheme is on par with the naïve scheme. This indicates that the additional cache invali-

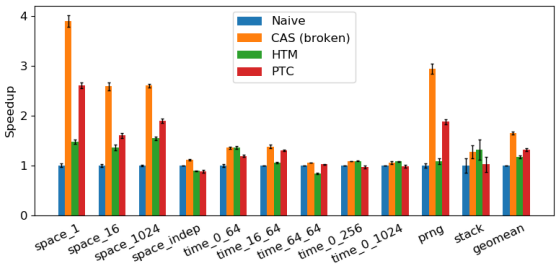

Figure 12: Micro-benchmarks using eight cores.

dation and lookup present in the PTC scheme incurs negligible overhead.

The HTM scheme shows performance comparable to the other schemes only if the transactions are small, i.e. there are few instructions between LL and SC. A significant drop in performance is observed for benchmarks that perform substantial work inside an LL/SC sequence. This causes the underlying transactions to become relatively large, and they require more hardware effort (cycles) to successfully execute and commit.

Next, we evaluate the performance of the proposed schemes in a multi-core context with eight cores. In multi-core execution, each core must synchronize and communicate the LL/SC linkage, resulting in different LL/SC success rates for different schemes. The efficiency of linkage communication affects the overall performance beyond the single-core overhead. The results are shown in Figure 12.

In the case of high space congestion (`space_1`), PTC is able to outperform the naïve scheme by 2.6×. The broken CAS scheme results in the best performance due to its efficient handling of LL/SC instructions. Fast LL/SC handling lowers the chances of interleaving concurrent LL/SC pairs, resulting in lower SC failure rates. Using the CAS scheme, each SC instruction fails three times on average before succeeding, while the PTC scheme results in six failures for each successful SC instruction.

If there is no congestion and all LL/SC pairs update independent memory locations (`space_indep`), the naïve scheme is able to outperform both PTC and HTM. Since the benchmark has few regular stores, the naïve scheme does not suffer the overhead of unnecessary synchronization of all regular stores. At the same time, the naïve scheme emulates LL instructions much more efficiently, as it does not need to handle the PTC invalidation.

In the case of time congestion (`time_64_64`), large frequent LL/SC sections result in low HTM performance. The poor HTM performance in this case has also been observed for single-core execution. If the LL/SC sections are kept small (`time_0_64`), the HTM scheme outperforms the PTC scheme with a speed-up over Naïve of 1.36× for HTM and 1.19× for PTC. This is because HTM can rely on transactional semantics to avoid data races and thus, if most transactions succeed, it can avoid explicit synchronization with other cores.

3.3 Scalability

In this section, we evaluate the effect of varying the number of cores on the performance of the schemes. We vary the number of cores from one to eighteen in two core increments. The upper number of cores is selected such that all simulation threads, i.e. virtual cores, can be scheduled at the same time, on the same processor. This configuration results in the greatest level of parallelism without incurring any NUMA overhead caused by inter-processor communication.

All micro-benchmarks are constructed such that a constant amount of work is performed overall. Effectively, the benchmarks split the same workload evenly between all available cores. We expect to see two types of ideal scaling characteristics. First, data parallel benchmarks with a low level of congestion (such as `space_indep`) should exhibit runtimes inversely proportional to the number of cores. Second, high-congestion benchmarks (such as `space_1`) need to sequentialize almost all execution. In this case, adding more cores should result in only marginal runtime improvements, i.e. the runtime is expected to remain constant for all number of cores.

Without much congestion (`space_indep`), the schemes scale almost ideally. For a greater number of cores, the performance degrades most significantly for the HTM scheme. On the other hand, CAS exhibits little performance degradation even for high number of cores. This is because CAS is a long established synchronization mechanism that has been highly optimized by the hardware designers, whereas hardware transaction technology is relatively new. The scalability characteristics of hardware transactions are only expected to improve with new architecture designs.

High space-congestion benchmarks (`space_1`) exhibit near ideal scaling only in the case of CAS. HTM scales poorly and above ten cores becomes the worst performing scheme. For this many cores, very few transactions are able to complete without conflicts, resulting in almost all LL/SC instructions taking the fallback path. The PTC scheme scales significantly better than the Naïve scheme, especially at the medium number of cores (up to ten). The Naïve scheme shows no improvement even when moving from one-core to two-core execution. We attribute this scalability improvement to the optimizations introduced by PTC, as discussed in Section 2.3.2. In particular, using a lock to handle both LL and SC instructions forces the execution (in the case of Naïve) to sequentialize unnecessarily.

The time-congestion benchmarks show similar behavior to the space-congestion benchmarks. CAS scales almost ideally for all levels of congestion. The PTC scheme scales similarly to CAS, and outperforms all other correct schemes. The HTM scheme shows poor and unstable performance. Especially for a small number of cores, the performance is sensitive to the hardware implementation, which affects the efficiency of conflict detection, and transaction failures. For higher numbers of cores, few LL/SC sequences succeed on the transactional path, resulting in most executions taking the fallback path.

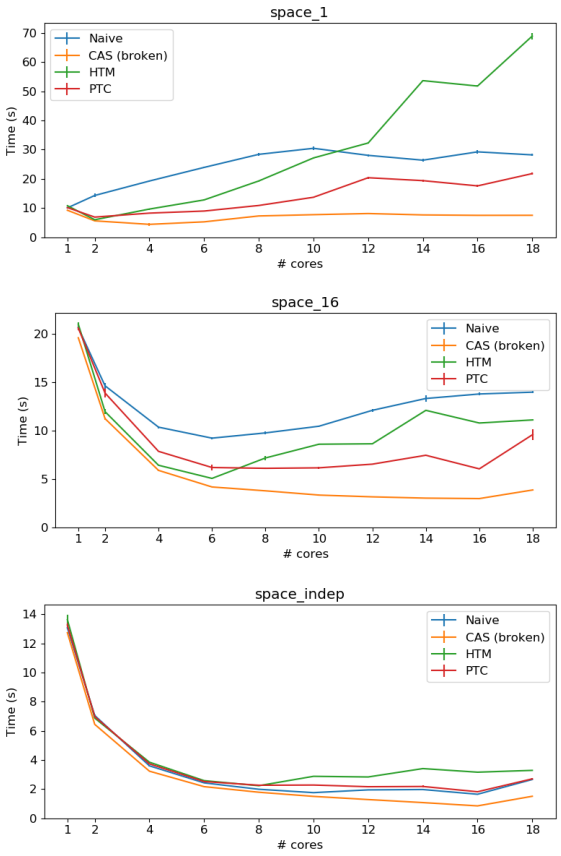

Figure 13: Scalability of space micro-benchmarks. High-congestion benchmark at the top.

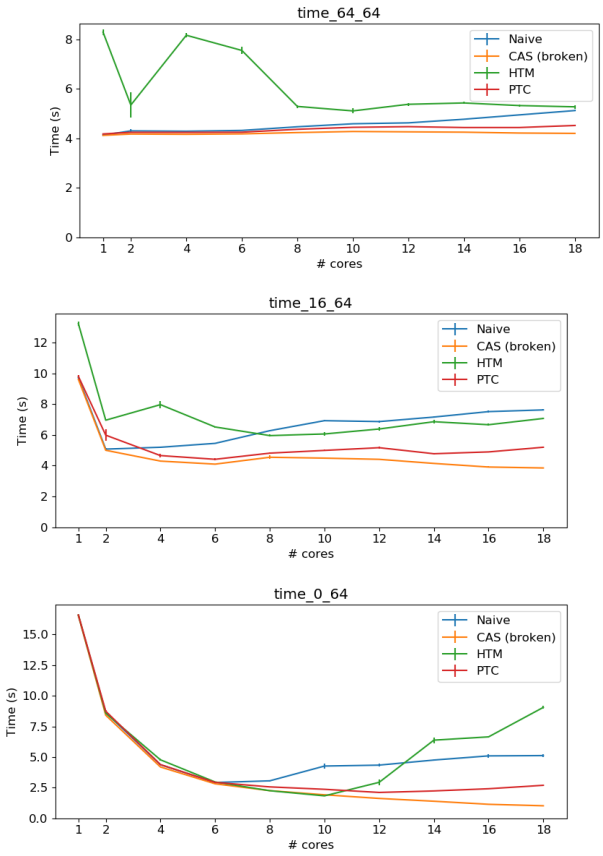

Figure 14: Scalability of time micro-benchmarks. High-congestion benchmark at the top

4 Proof of Correctness

To ensure that our emulation schemes respect the desired LL/SC semantics we prove that the naïve and software-only schemes' behavior is allowed. We do not consider the HTM scheme here for space reasons. We assume that the host has a Total-Store-Order memory model like x86, where each core's writes appear in memory in program order. However, guest architectures with LL/SC such as Arm will often have a weaker memory model, so first we define our expectations for LL/SC in such a model.

Axiomatic Definition of Atomicity. In an axiomatic memory consistency model such as ARMv8 [18], we assume a coherence order relation (co) between all Writes to the same address, and a reads-from relation (rf) between a Write and a Read that reads its value. A from-reads relation (fr) can be derived from these as between a Read and a Write that is later (in co) than the Write the Read reads-from. We can also consider parts of these relations that relate events from different external threads, and call these coe, fre, rfe.

The atomicity condition for LL/SC is then that there is no fre followed by coe between any LL and a successful SC. Expanding this definition, since the LL incorporates a Read, and there cannot be a Write from a different thread after (in coherence order co) the one that reads-from and before the Write from a successful SC. This captures ARMv8-like LL/SC correctness. For architectures which allow no Writes (not even same-thread Writes) between the LL and SC, the condition can be stated as there is no fr followed by co between any LL and a successful SC.

Axiomatic Definition of Properly Locked Executions. For locks we have two events, lock acquire and lock release. The definition of Properly Locked executions says that any successful lock acquire is followed by the corresponding lock release, and preceded by the previous release (or no lock events for the initial acquire). This order must be common to all threads. Further, same thread events (Writes, Reads) after the lock acquire are globally ordered after the acquire, and same thread events before the release are globally ordered before the release.

Naïve Scheme from Section 2.1 Here all LL, SC and Writes are guarded by locks, and furthermore the same lock is used for any particular address.

Proof: In this case we have a global order for any properly locked executions. Since all LL, SC and Write events are

between successful lock acquires and releases on the same thread, they are globally ordered by the lock ordering. We can read off coherence order co (and coe) as a subset of this global order, and similarly for from-reads order fr (and fre). Note that the place of a LL in this lock order corresponds to when its address is saved, not when the read is done. This is safe because if any write (SC or normal) from other threads intervenes in the lock order between the LL and a SC, the lock address is guaranteed to be invalidated and thus the SC must fail. The correctness of the atomicity condition is thus ensured by the check done for successful SC (within a locked region) that no other thread has done a Write since the last LL.

Software-only Scheme from Section 2.3 In this scheme, the SC and slow path Writes are still guarded by locks, but Writes on the fast path used when the page table cache lookup succeeds are guarded by a different lock (tag bit) in the cache entry. An LL invalidates the cache entries for the matching virtual address on all other threads (within a locked region), spinning on the lock in the old cache entry to avoid races with ongoing fast path Writes, *before* invalidating all other locked addresses, *and then* reading the value. Plain Writes then check whether the virtual address is valid on that thread. If so, the Write action is done immediately (fast path). If not (slow path), the thread acquires the lock, performs the Write action, and *then* clears matching locked addresses on all other threads (and releases the lock). An SC checks whether the locked address is still held, and if so then does the Write and succeeds, otherwise it fails.

Proof: In this case, if all Writes go through the slow path then the proof is analogous to the naïve version (use the lock ordering guaranteed to exist for properly locked executions to find the co and fr relations). The only wrinkle is that the LL is not locked and therefore does not participate in the lock ordering. Consider however that the LL only does the underlying read of the value after saving the locked address, and the locked address is invalidated by any such write, *after* the write. So either the LL reads the old value, and then the write (and possible invalidation) occurs, or the LL reads the new value, but if so the locked address will get invalidated before the writing thread releases the lock. Both cases are safe. Now let us consider the case that one or more Write goes through the fast path. We show that no such Write can come in between the LL and a successful SC (in the time order implied by the ordering between the LL and SC's acquire). Indeed, using the fast path means the virtual address is valid on the Write's thread. Moreover, it must be valid *when the Write happens* because it is protected by the cache tag entry lock. Since the LL invalidates virtual addresses on all threads before doing the Read, this means the Write must be ordered before the invalidate which is part of the LL, or else the write cannot go on the fast path until after the SC mutex release. In the first case, since the LL does the invalidate while spinning on the lock in the cache entry, the Write must have been done before the invalidate completed, i.e. definitely before the underlying read of the LL. In the second, such a fast-path Write is by definition not between the LL and SC. Of course, when the LL has invalidated the Writer's page table entry, the write can still go on the slow-path, but then we are back to the previous case.

5 Related Work

Rigo et al. [20] highlight the issues around the correct handling of LL/SC instructions in QEMU, and a possible solution is provided. While somewhat similar to our software-only scheme, it does not explain (or even proves) how correctness is achieved. The paper does not offer any evaluation or comparison with other schemes. Prior solutions developed in PQEMU [8] and COREMU [22] are shown to suffer from the ABA problem, i.e. they incorrectly implement the LL/SC semantics as demonstrated in our motivating example. Pico [5] introduces support for hardware transactional memory for the emulation of LL/SC synchronization, while an intermediate software approach uses extensive locking. In contrast, our approaches employ sparse locking and we deliver a formal proof-of-correctness. Qelt [4] is a recent development based on QEMU. While fast thanks to its floating-point acceleration, the paper does not offer any details on the implementation of their LL/SC emulation approach. Jiang et al. [12] employs a lock-free FIFO queue for LL/SC emulation, but the paper is hard to follow and lacks a convincing correctness argument. XEMU [21] considers guest architectures with native LL/SC support. Gao et al. [11] presents a method to adapt algorithms using LL/SC to a pointer-size CAS without the ABA problem by using additional memory for each location, which would not be suitable for an emulator. The implementation presented assumes sequential consistency, but does include a formal proof mechanized in the PVS proof assistant. A more theoretical approach to LL/SC implementation without performance evaluation is taken in [14]. A wait-free multi-word Compare-and-Swap operation is the subject of [10]. A non-blocking software implementation of LL/SC, which makes use of atomic k-word-compare single-swap operations, is developed in the patent in [17].

6 Summary & Conclusions

In this paper, we have shown that existing state-of-the-art DBT systems implement an approximate version of *load-link/store-conditional* instructions that fails to fully capture their semantics. In particular, we showed how these implementations can cause bugs in real world applications, by causing the ABA problem to appear in e.g. lock-free data structures, in what would otherwise be a correct implementation. We presented software-only and HTM assisted schemes that correctly implement LL/SC semantics in the commercial Synopsys DesignWare® ARC® nSIM DBT system. We evaluate our schemes and show that we can maintain high simulation throughput for application benchmarks for provably correct LL/SC implementations.

References

[1] Liblfds. URL http://www.liblfds.org.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[4] Emilio G. Cota and Luca P. Carloni. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 74–87, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6020-3. doi: 10.1145/3313808.3313811. URL http://doi.acm.org/10.1145/3313808.3313811.

[5] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-ISA machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 210–220, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. URL http://dl.acm.org/citation.cfm?id=3049832.3049855.

[6] D. Dechev. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 158–167, Oct 2011. doi: 10.4108/icst.collaboratecom.2011.247161.

[7] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[8] J. Ding, P. Chang, W. Hsu, and Y. Chung. PQEMU: A parallel system emulator based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 276–283, Dec 2011. doi: 10.1109/ICPADS.2011.102.

[9] EEMBC Multibench EEMBC. 1.0 multicore benchmark software, 2016.

[10] Steven Feldman, Pierre Laborde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *Int. J. Parallel Program.*, 43(4):572–596, August 2015. ISSN 0885-7458. doi: 10.1007/s10766-014-0308-7. URL http://dx.doi.org/10.1007/s10766-014-0308-7.

[11] Hui Gao, Yan Fu, and Wim H. Hesselink. Practical lock-free implementation of ll/sc using only pointer-size CAS. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, pages 320–323, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3887-7. doi: 10.1109/ICISE.2009.841. URL http://dx.doi.org/10.1109/ICISE.2009.841.

[12] Xiao-Wu Jiang, Xiang-Lan Chen, Huang Wang, and Hua-Ping Chen. A parallel full-system emulator for risc architure host. In Hwa Young Jeong, Mohammad S. Obaidat, Neil Y. Yen, and James J. (Jong Hyuk) Park, editors, *Advances in Computer Science and its Applications*, pages 1045–1052, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-41674-3.

[13] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[14] Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In Rachid Guerraoui, editor, *Distributed Computing*, pages 144–158, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30186-8.

[15] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[16] Ragavendra Natarajan and Antonia Zhai. Leveraging transactional execution for memory consistency model emulation. *ACM Trans. Archit. Code Optim.*, 12(3):29:1–29:24, August 2015. ISSN 1544-3566. doi: 10.1145/2786980. URL http://doi.acm.org/10.1145/2786980.

[17] Oracle America, Inc. Method and apparatus for emulating linked-load/store-conditional synchronization, 01 2011. URL https://patents.google.com/patent/US7870344.

[18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. *PACMPL*, 2(POPL):19:1–19:29, 2018. doi: 10.1145/3158107. URL https://doi.org/10.1145/3158107.

[19] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE*

international symposium on Microarchitecture, pages 294–305. IEEE Computer Society, 2001.

[20] Alvise Rigo, Alexander Spyridakis, and Daniel Raho. Atomic instruction translation towards a multi-threaded QEMU. In *Proceedings 30th European Conference on Modelling and Simulation*, pages 587–595, United Kingdom, 2016. European Council for Modeling and Simulation.

[21] Huang Wang, Chao Wang, and Huaping Chen. XEMU: A cross-ISA full-system emulator on multiple processor architectures. *Int. J. High Perform. Syst. Archit.*, 5 (4):228–239, November 2015. ISSN 1751-6528. doi: 10.1504/IJHPSA.2015.072853. URL http://dx.doi.org/10.1504/IJHPSA.2015.072853.

[22] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941583. URL http://doi.acm.org/10.1145/1941553.1941583.