# Architectural Support for Fair Reader-Writer Locking

Enrique Vallejo*, Ramón Beivide*, Adrián Cristal†‡, Tim Harris§, Fernando Vallejo*, Osman Unsal†, Mateo Valero†¶
*University of Cantabria, Spain †Barcelona Supercomputing Center, Spain §Microsoft Research Cambridge, UK
‡IIIA - Artificial Intelligence Research Institute, CSIC, Spain ¶Universitat Politècnica de Catalunya, Spain

***Abstract*—Many shared-memory parallel systems use lock-based synchronization mechanisms to provide mutual exclusion or reader-writer access to memory locations. Software locks are inefficient either in memory usage, lock transfer time, or both. Proposed hardware locking mechanisms are either too specific (for example, requiring static assignment of threads to cores and vice-versa), support a limited number of concurrent locks, require tag values to be associated with every memory location, rely on the low latencies of single-chip multicore designs or are slow in adversarial cases such as suspended threads in a lock queue. Additionally, few proposals cover reader-writer locks and their associated fairness issues.**

**In this paper we introduce the Lock Control Unit (LCU) which is an acceleration mechanism collocated with each core to explicitly handle fast reader-writer locking. By associating a unique *thread-id* to each lock request we decouple the hardware lock from the requestor core. This provides correct and efficient execution in the presence of thread migration. By making the LCU logic autonomous from the core, it seamlessly handles thread preemption. Our design offers richer semantics than previous proposals, such as *trylock* support while providing direct core-to-core transfers.**

**We evaluate our proposal with microbenchmarks, a fine-grain Software Transactional Memory system and programs from the Parsec and Splash parallel benchmark suites. The lock transfer time decreases in up to 30% when compared to previous hardware proposals. Transactional Memory systems limited by reader-locking congestion boost up to 3× while still preserving graceful fairness and starvation freedom properties. Finally, commonly used applications achieve speedups up to a 7% when compared to software models.**

***Keywords*-reader-writer locks; fairness; Lock Control Unit;**

## I. Introduction

Critical sections protected with locks limit the performance of parallel programs when their execution has to be serialized. They can be divided in three phases: Lock transfer, load/compute and lock release. The second phase is innate to the parallel algorithm, while the first and third phases are dependent on the implementation of locks and only required for correct execution. The same access pattern applies to data structures protected by read-write locks, which allow for a single writer or multiple concurrent readers. To provide the maximum performance, a parallel system should provide a low-overhead implementation of locks.

Basic single-cache-line locks such as test-and-set (TAS) or test-and-test-and-set (TATAS) generate coherence contention among threads when accessing the shared resource, which degrades performance. Queue-based locks [27] are based on local spinning (which reduces contention) and provide FIFO order (which guarantees fairness) but have a more complex transfer mechanism and incur a high memory usage. While fairness is a desirable property, strict FIFO order leads to starvation if the recipient thread is not active when receiving the lock grant. He *et al.* propose [15] queue-based locks with timestamps so that preempted threads are detected and removed from the queue. Also, biased locks [8], [31], accelerate the execution when taken repeatedly by a single thread.

Other interesting requirements for current workloads are reader-writer (RW) locks and *trylock* support. Reader-writer locks allow for a single lock owner in write mode or multiple concurrent readers. This imposes additional fairness requirements, since some policies (such as readers or writers preference) can generate starvation [28]. Trylocks allow the requestor to abort waiting and perform a different action if it takes too long. For example, many current Software Transactional Memory (STM, [14]) systems use trylocks, and as argued by Dice and Shavit [9] they would benefit from an efficient implementation of RW locking.

Different hardware mechanisms have been proposed to support and/or accelerate lock handling. They will be analyzed in detail in Section II. However, they all incur excessive requirements or suffer from limitations that restrict their use as a generic fine-grain locking mechanism or as the basis to build STM systems.

In this paper, we present a flexible hardware mechanism for multiprocessor systems which is able to support word-level fine-grain reader-writer locks with fairness and fast transfer time. Our mechanism is distributed and completely decoupled from the core or L1, allowing for a modular design. The main contributions of this paper are:

- We introduce the Lock Control Unit (LCU), a hardware mechanism to accelerate reader-writer locking with fairness guarantees. The LCU preserves correct execution in presence of thread suspension, migration or aborted requests (*trylock*) without blocking or suffering a significant performance degradation. Altogether, these characteristics make the LCU suitable for fine-grain RW-locking or for building current STMs.
- We present a proposal for managing resource overflow, providing mechanisms to guarantee forward progress independently of the number of locks being accessed.
- We conduct a detailed evaluation running different

| | | Performance | | | | Flexibility | | | | | Implementation overhead | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Reader-writer | Local spin | Fair | Queue-based | Queue Eviction detection | Trylock support | Scalability | Migration | threads > cores? | Memory/ area overhead | transfer #messages (critical path) | L1 changes |
| SW-only | TAS, contended RW locks | NO/YES | NO | opt | NO | | YES | poor | YES | YES | 1 cache line | 6 | NO |
| | MCS locks [27] | NO | YES | YES | YES | NO | NO | good | YES | YES, slow | O(n) cache lines | 6 | NO |
| | MCS-RW, Krieger, Lev [28,20,24] | YES | YES | YES | YES | NO | NO | good | YES | YES, slow | O(n) cache lines | 6 | NO |
| | Biased locks [8, 31] | NO | opt | opt | opt | NO | YES | good | YES | YES, slow | O(n) cache lines | 6 | NO |
| | Time-published locks [15] | NO | YES | YES | YES | YES | YES | good | YES | YES | O(n) cache lines | 6 | NO |
| HW-supported | fetch&op, AMOs [22, 35, 42] | NO | NO | NO | NO | | YES | good | YES | YES | 1 cache line | 2 | NO |
| | SoC Lock Cache [32] | NO | NO | YES | NO | | NO | bus-based | NO | NO | low | 2 | NO |
| | Lock Table [5] | NO | NO | NO | NO | | YES | bus-based | YES | YES | low | 2 | NO |
| | Tagged memory [1, 3, 6, 7, 17] | NO | NO | NO | NO | | YES | good | YES | YES | Memory tags | 6 | YES |
| | QOLB [13] | NO | YES | YES | YES | NO | NO | good | NO | YES, slow | Memory tags | 1 | YES |
| | SSB [43] | YES | NO | NO | NO | | YES | single-chip | NO | NO | low | 2 | NO |
| | **Lock Control Unit** | **YES** | **YES** | **YES** | **YES** | **YES** | **YES** | **good** | **YES** | **YES** | **low** | **1** | **NO** |

Figure 1. Comparison of different locking mechanisms.

benchmarks and show that our system outperforms previous implementations. We evaluate our proposal using three workload classes: microbenchmarks, a fine-grain Software Transactional Memory system and classical parallel benchmarks. The lock transfer time decreases in up to 30% from previous proposals; STM systems limited by reader-locking congestion boost up to 3×; finally, widely used parallel benchmarks achieve speedups up to 7% from previous software models.

Our system relies on two new architectural blocks: The Lock Control Unit (LCU) for exploiting *locality* and *fast transfer time*, and the Lock Reservation Table (LRT) to manage lock queues. Each core implements a LCU, a hardware table, with entries dynamically reserved for the requested locks. It is responsible for receiving the thread's requests and building the queues. Each memory controller provides an LRT which is responsible for the allocation and control of new lock queues. LCUs and LRTs communicate to each other for requests, transfers and releases of RW-locks, while threads only query the local LCU. To minimize transfer time, lock transfers are direct from one LCU to the next.

Contrary to many hardware locking proposals, we associate locks with logical threads by using a $thread_{id}$ identifier. This decouples the lock from the physical cores, allowing for thread suspension and migration with a graceful performance degradation. We also handle the unavoidable case of resource overflow: The LRT is backed by main memory, while the LCU contains specific entries to guarantee forward progress.

## II. Related Work

Multiple lock implementations, both software-only and hardware accelerated, have been proposed and evaluated. Although we discuss below certain details of the most relevant ones, Figure 1 summarizes their main characteristics. The meaning of some columns of the table follows; the label of other columns are conventional. *Local spin* refers to implementations that wait for the lock iterating on a per-thread private location or structure; it is typical in queue-based locks and has the benefit of not sending remote messages while spinning. *Queue eviction detection* refers to the capability of detecting evicted threads in the queue before they receive the lock, so they can be removed to prevent temporal starvation. *Scalability* refers to the system behavior as the number of requestors increase. Single-line locks present coherent contention and scale poorly, while queue-based approaches remove this problem and scale very well. Regarding hardware proposals, they can be limited to a single-bus or single-chip, what can restrict their scalability for larger systems. Some proposals can fail if one thread *migrates* or if the number of *threads exceeds the number of cores*. This happens typically because locks are assigned to hardware cores instead of threads. *Memory/area overhead* refers to the memory required for the lock, or to the area cost in hardware implementations. Queue-based locks require $O(n)$ memory locations per lock for $n$ concurrent requestors. Some hardware proposals can have high area requirements (for example, tagging the whole memory) or limited structures per core. The number of *Transfer messages* will limit the base transfer time latency of the system. It can depend on the specific coherence protocol, but in general, hardware proposals outperform software-only locks. Finally, requiring *changes to the L1 design* is undesirable, since it implies modifying a highly optimized structure, and possibly, the coherence protocol. As shown, our proposal satisfies all these desirable requirements. Next, we review the details of the most relevant proposals.

Multiple software implementations of locks exist. A good survey can be found on [16]. Mellor-Crumley and Scott developed the MCS lock [27] that implements a queue-based lock with FIFO access. Different reader-writer variants of the MCS lock were developed thereafter. Mellor-Crumley and Scott propose in [28] the first version, denoted as *MRSW* that uses a *reader counter*, allowing concurrent readers to release

the lock in any order. Then, only the last reader to release sees *reader counter = 0* and notifies the following writer in the queue. The *reader counter* becomes a coherence hotspot, which led Krieger *et al.* to propose in [20] a version that does not implement a single counter, but requires a doubly-linked list and a more complex management of reader unlocking. Lev *et al.* proposed in [24] three scalable reader-writer implementations based on the idea of Scalable Non-Zero Indicators (SNZI), a data structure optimized to increase throughput and minimize memory contention. However, their implementation requires more memory accesses and uses a large amount of memory.

Queue-based locks entail the challenge of preventing the lock transfer to a thread that is temporarily not spinning (e.g. preempted), or has abandoned waiting (in a trylock). Scott and Scherer propose in [33], [34] versions of the MCS lock with aborting capability. He *et al.* present in [15] an implementation that detects and skips preempted waiting threads. To the best of our knowledge, no trylock or preemption detection mechanism has been developed for queue-based RW-locks.

Fine-grain synchronization has been supported in hardware with word-level tags (such as Full/Empty bits) in multiple machines such as HEP [17], Tera [3], the J- and M-machines [6], [7], or the Alewife [1]. While these bits provide a simple architectural support for synchronization, they incur a significant area overhead, and do not support RW-locking. The changes required to support RW-locking would imply unaffordable area costs.

QOLB, [13], leverages synchronization bits by building a hardware queue between requestors. QOLB allocates two cache lines for every locked address: the valid one, and a shadow line. One of the lines is used to maintain a pointer to the next node in the queue, if present, and the other one is used for local spinning on the *syncbit*. When the unlock is invoked, the pointer is used to notify the remote core, allowing for fast direct cache-to-cache transfer. As with previous designs, QOLB does not support RW locks and suffers a noticeable performance degradation when evicted threads receive a lock.

The System on a Chip Lock Cache, [32] and the Lock Table, [5] use a centralized system to control the status of several locks. Acquired locks are recorded by leveraging a bus-based coherence mechanism, so their scalability is limited. Moreover, overflow and thread migration are difficult to handle.

Different hardware mechanisms provide direct support for locking on parallel systems. The Stanford DASH [23] leverages the directory sharing bits for this purpose. When a lock is released, the directory propagates the coherence update to a single lock requestor, preventing contention. Other systems use remote atomic operations executed in the memory controller instead of the processing core. This is the case of the *fetch-and-θ* instruction of the Memory Atomic Operations (MAO) in the MIPS-based SGI Origin [22] and Cray T3E [35]. These systems allow for remote updates in main memory, removing the problem of coherence lines bouncing on lock accesses. All remote updates take the same time, which is typically the memory access latency. While they do not support direct cache-to-cache transfer of locks, they do not use L1 memory at all. More elaborate mechanisms such as strided accesses to shared data have been proposed in Active Memory Operations (AMO, [42]) but the locking mechanisms remain the same.

The Synchronization State Buffer (SSB, [43]) avoids whole-memory tagging by using a dedicated hardware structure When a synchronization operation (such as *lock_acquire*) is invoked on a given data address, the shared-L2 memory controller allocates an entry in the SSB to control the locking status of such address. SSB supports fine-grain reader/writer locks. However, these locks are unfair and can starve writers. Same as in MAOs or AMOs, all lock-related operations are remote (in this case, in the on-chip L2 controller).

Finally, multiple proposals focus on accelerating critical section execution. Two prominent examples are Speculative Lock Elision [30], which removes unnecessary locking, and Accelerated Critical Sections [39], which moves their execution to the fastest core. While somewhat related, these approaches are orthogonal to our work. Transactional Memory (TM), [14] aims to simplify the programming model removing the need of locks. However, the implementation of most software TM systems still makes them rely on an efficient lock implementation.

## III. Flexible hardware support for reader-writer locking

Our proposal allows for the acquisition of locks in any memory location, by means of two new ISA synchronization primitives: Acquire (*acq*) and Release (*rel*) [1]. They require three arguments: the *address* to lock (using the TLB to provide virtual- to-physical mappings), a process-local software $thread_{id}$ (provided by the user or the hardware) and the read or write mode, *R/W*. These instructions do not block the core until completion; instead, they return TRUE or FALSE depending on the lock being acquired/released or not, and the software will have to iterate until success.

Figure 2 shows the simplest implementation of the *lock*, *trylock* (based on a fixed number of retries) and *unlock* functions, using the *acq* and *rel* primitives. Alternative implementations can consider the use of timeouts in trylocks or periodically *yielding* the processor to prevent owner starvation as in [15].

The architecture of our proposal is presented in Figure 3. It has been conceived for distributed shared-memory

[1]Additionally, we could consider a third *Enqueue* primitive, similar to QOLB [13], to be used as a prefetch of the lock.

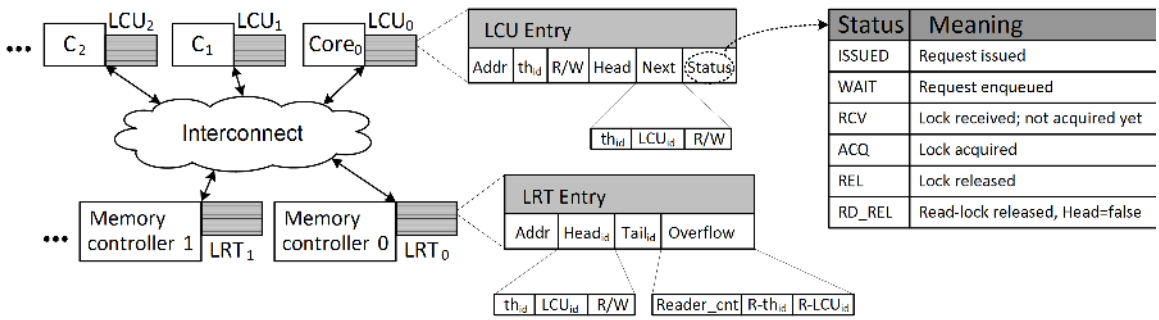



Figure 3. Architectural blocks of the lock handling mechanism and allowed LCU status values.

```
void lock(addr, th_id, read){
  while(!acq(addr, th_id, read)) {}
}

bool trylock(addr, th_id, read, retries){
  for (int i=0; i<retries; i++)
     if(acq(addr, th_id, read)) return true;
  return false;
}

void unlock(addr, th_id, read){
  while(!rel(addr, th_id, read)) {}
}
```

Figure 2. Lock acquisition and release functions.

systems with multiple cores and multiple chips . Each core incorporates a Lock Control Unit (LCU) for implementing distributed lock queues. There is one Lock Reservation Table (LRT) per memory controller. Alternative organizations can be considered, as long as each lock is associated with a single LRT depending on its physical address.

The LCU is composed of a table whose entries record the locking status of a certain memory address, and its associated logic. The LCU is addressed with the tuple *(addr, thread$_{id}$)*, so multiple threads on the same core can request the same lock. LCU entries are allocated on requests from the core. The fields of each LCU entry are presented in Figure 3, requiring around 20 bytes of storage using conservative values. Valid entry *status* values[2] are presented Figure 3. While most of them are self-defining, we will detail the two variants of *released* later. We distinguish between *contended* and *uncontended* locks. The former are locks required by multiple threads that concurrently try to acquire them, and the latter are locks taken by a single thread without further requestors. LCU entries for *uncontended* locks are removed to minimize LCU occupation. Contended locks build a queue of requestors, with LCU entries acting as queue nodes. At a given point in time, only one node will be the queue *Head*. A LCU entry gets *enqueued* when there is a chain of pointers from the *Head* that reaches such an entry.

[2]The actual implementation might require additional transient status values to prevent races, depending on the ordering properties of the network.

Once a LCU entry is enqueued, it will eventually receive the lock "grant", and become the *Head* node. Once built, the queue of LCU entries is never broken or split.

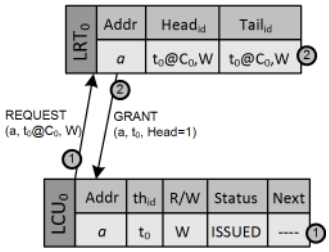
The LRT is the unit that manages the locking status of each memory location. LRT entries are allocated on request, so locks can be taken on any location of the system memory but only locked addresses account for the hardware requirements. LRT entries record the state of the locks and maintain pointers to the queues of waiting threads. The LRT can assign locks to owners (threads) in two modes: the ordinary, queue-based mode, or an "overflow" mode used for readers without available LCU entries. Each LRT entry contains the fields depicted in Figure 3, which serve two main purposes:

- Queue pointers: Tuples *(thread$_{id}$, LCU$_{id}$, R/W mode)* that identify the queue *Head* and *Tail*.
- Support for the "overflow" mode: A *reader_cnt* field, and a tuple *(R-thread$_{id}$, R-LCU$_{id}$)* for a reservation mechanism which will be detailed in Subsection III-D.

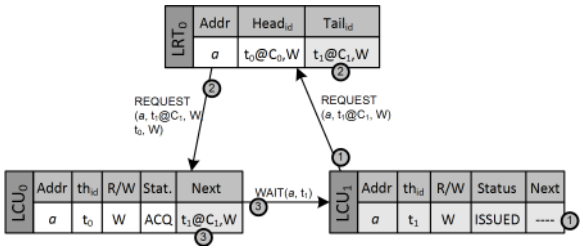
The core communicates with its LCU with the synchronization instructions *Acquire* and *Release*, offloading the tasks of requesting the lock grant, forming a queue of waiting threads and transferring the lock to the next requestor. The ISA instructions and the hardware are related as follows:

- *Acquire*: If a LCU entry for this address is not present, it is allocated and a request message is sent to the corresponding LRT. If it is present with a valid *Status* (*RCV* for writers, *RCV* or *RD_REL* for readers, as explained later), the lock is acquired, returning TRUE. Otherwise, no action takes place.
- *Release*: releases the lock, transferring it to the next requestor or releasing it (sending a message to the LRT). The LCU entry is re-allocated if not present.

The detailed lock management is described next. We begin showing how the LCU and the LRT interact in the simplest write lock enqueue, acquisition, transfer and release operations. Subsequent Subsections will deal with RW-locking, thread migration, suspension, resource exhaustion and memory paging.




(a) Lock is free, request proceeds. The LCU entry will be removed after the local thread acquires the lock.




(b) Lock is taken uncontended. The owner re-allocates the LCU entry with the information sent from the LRT (steps 2 and 3) and sends a WAIT message.

Figure 4. Write lock requests.

### A. Write locking

When a thread requests a lock *acquisition* on a given memory address *a*, the LCU allocates an *ISSUED* entry for the lock and sends a lock REQUEST message to the corresponding LRT. The LRT will grant the lock or add the requestor to the end of the queue, depending on the lock status. However, since the lock can be acquired in uncontended mode, there are three possible cases, described next. Figure 4 shows an example of the first two cases.

(a) *The address is not locked, so the LRT does not contain an entry for address* a. The LRT allocates a new entry, recording the requestor's data in both $head_{id}$ and $tail_{id}$ pointers and returns a GRANT message with a *Head* flag set (Fig. 4a). When received, the LCU switches the status to *RCV* and sets the *Head* flag. On the next *acq* over this lock, it will be taken and the *acq* instruction will return TRUE (according to the code in Fig. 2, the thread iterates until succeeding). This lock is uncontended, taken by a single requestor with null values in the *Next* field of the LCU entry. Since no queue exists, there is no need to maintain a node for such lock. Hence, the LCU entry is automatically removed once the lock is *acquired*, leaving room for other locks taken concurrently. The LRT still records the locking data for new locking requests, not being aware of the LCU entry release.

(b) *The address is locked in uncontended mode*. The LRT forwards the lock request, including the values in the *head* tuple, to the owner LCU. With such information, the owner LCU re-allocates the entry, which had been released on acquisition (case *(a)* above). The status is set to *ACQ*, recording the requestor information in the $next$ tuple and sending a *WAIT* message to the requestor.

(c) *The address is locked in contended mode with an associated queue*. Similar to *(b)*, the LRT forwards the request to the *tail* LCU without having to re-allocate the LCU entry.

Lock *release* is triggered by the owner thread invoking the *rel* instruction. Its behavior depends on the existence of requestors. If the lock is uncontended, the LCU entry will not be present. However, invoking a *rel* implies that

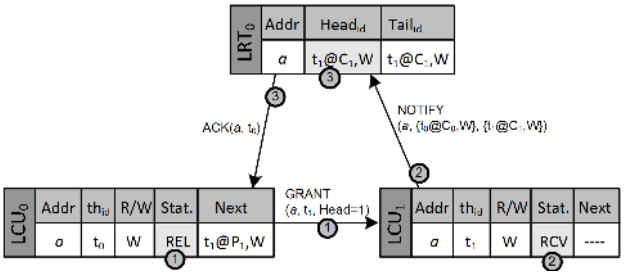



Figure 5. Lock transfer.

the lock has been acquired and the appropriate RELEASE message can be sent to the LRT; otherwise the program would be incorrectly synchronized. To this end, the LCU re-allocates the entry with the parameters from the *rel* instruction and sends a RELEASE message to the LRT. The LRT receives the lock release, removes its entry and sends and acknowledge back that allows the LCU to also remove its entry. The case of not having any free entry in the LCU is covered in Section III-D.

By contrast, if there is a queue, the lock is transferred to the *next* requestor. An example is depicted in Figure 5. The receiver notifies this transfer to the LRT, to update the *head* field. This notification is sent by the receiver LCU for two reasons: First, it takes the remote notification message off the critical path, minimizing the transfer time. Second, it delays the deallocation of the releaser LCU entry until the LRT sends the acknowledgement back. Again, this ensures that the *head* in the LRT always points to a valid LCU entry, as it is updated when the lock has been already received[3].

Finally, RETRY messages are used when data races are detected. For example, an uncontended entry could be released while a REQUEST message is being forwarded from the LRT. In such case, upon reception of the RELEASE, the LRT detects that the releaser is not the only node in the queue (the new requestor has been already recorded

[3]We use a *transfer_cnt* counter, not depicted in the figures, to prevent that consecutive lock transfers have to be serialized by the LRT notification or lead to a wrong result in case of a message race.

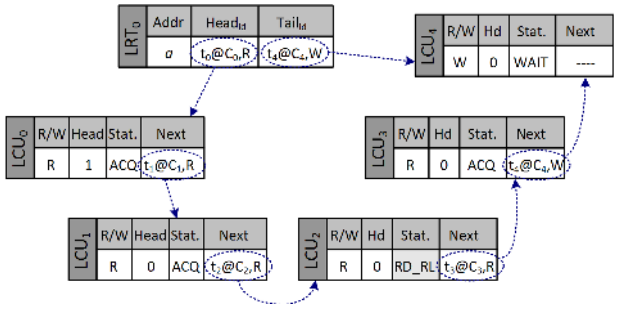



Figure 6. Example of concurrent read locking. Only the first node in the queue contains the *Head* flag set. Thread 2 at LCU 2 has released the lock, but being an intermediate node, the node is preserved in *RD_REL* status. The same thread can re-acquire the read-lock without remote requests.

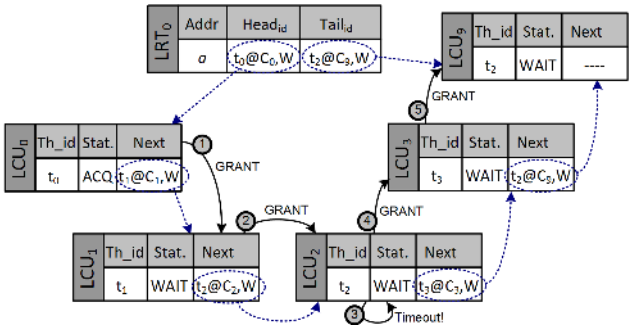



Figure 7. Example of migration. Thread $t_2$ has migrated from $LCU_2$ to $LCU_9$ while waiting. When the lock grant arrives at $LCU_2$ (step 2) the timeout triggers (step 3) and it is forwarded to the next node.

as the tail) and replies with a RETRY message to the RELEASE request. When the LCU with status *REL* receives the enqueue request, it will directly forward the lock to the requestor. This race shows how lock releases require a LCU entry re-allocation even when the lock is uncontended.

### B. Read locking

Read-locking employs the same queueing mechanism, with multiple consecutive LCUs allowed to concurrently hold the lock in read mode. While a single node is the queue *Head*, multiple readers can receive a lock "grant". When a waiting reader LCU entry receives a lock grant, it switches its status to *RCV* and forwards a GRANT message to any subsequent reader in the queue. Similarly, when a read request is forwarded to the tail, a GRANT reply is sent if the tail has the lock taken in read mode. For write-locking, by contrast, the *Head* flag and the lock grant are equivalent.

In this shared mode, all the readers can release the lock in any order. To prevent breaking the queue or granting the lock to a waiting writer when there are active readers, we rely on the *Head* token: When the first node in the queue releases the read-lock, the *Head* token is transferred to its *next* node and the LRT is notified as occurs for write locks in Figure 5. By contrast, when an intermediate node releases its read-lock, it switches to the special *RD_REL* status without sending any message, waiting for the *Head* token. This prevents an entry deallocation that would break the queue. The entry is finally released when the LCU receives the *Head* token, which is bypassed to the next node. An example with 4 concurrent readers and a waiting writer is presented in Figure 6.

While a node is in *RD_REL* status, the local thread can re-acquire it in read mode. Although this breaks the original FIFO order, it does not generate starvation: the advance of the *Head* token along the queue of readers ensures that, eventually, any enqueued writer receives the lock. This behavior prevents the problems of contention on a single memory location (note that the notification to the LRT is out of the critical path) or complex management found on software locks, [20], [28].

### C. Thread Suspension, Migration and Remote Release

Thread eviction becomes an issue in two cases: (i) An enqueued requestor thread is suspended or migrates, so it stops spinning on its local LCU entry (the same happens when a trylock expires), and (ii) A lock owner migrates and the *release* happens in a remote LCU.

In the first case, once received the lock grant, the LCU sets a timer that triggers a *release* if the lock is not taken within a threshold: this prevents starvation if the local thread suspends before *acquiring* the lock, or deadlock if a trylock expires or the lock requestor migrates to other core. If a migration occurs while spinning, execution will resume in the remote node in the same *while* loop presented in Figure 2. The thread will issue a new *enq* request in the new core, becoming the tail of the same lock queue. Then, there could be multiple entries with the same $thread_{id}$ along the queue. As only one will actually acquire the lock, the others will simply pass it through after the threshold without causing any issues. Figure 7 presents an example where thread $t_2$ has migrated, while waiting, from core $C_2$ *(LCU2)* to $C_9$.

The second case (lock owner migration) is detected when a lock is released from a remote LCU different to the one recorded at the *Head* of the LRT. Since the remote LCU contains no allocated entry, the release will be forwarded to the LRT as if it was uncontended-locking. The LRT will detect the migration and, if a queue exists, forward the release to the original queue head node, which will send the lock to the next requestor. This is the reason to maintain always a valid queue head pointer, as described in Subsection III-A. If the lock was in read mode, the migrated thread that releases the lock might not be the queue head. In such case, the message is forwarded through the queue until it reaches the proper LCU. Once the appropriate LCU is found, it acknowledges the remote requestor and behaves as if a local *Release* had been invoked. This mechanism also allows the case of a thread releasing a lock acquired by a different thread by just borrowing its $thread_{id}$.

### D. LCU overflow and forward progress

The LCU entries are the interface with the locking mechanism. They are used in the local spin of the thread for both lock acquisitions and releases. Therefore, when all LCU entries get allocated in queues, the local thread cannot acquire any more locks. To prevent such deadlock, we introduce *nonblocking* LCU entries, which behave in the same way as ordinary entries, but they are not allowed to become part of a queue. When a request from a nonblocking entry is received for a taken lock, a RETRY message is sent back, rather than the corresponding WAIT. To this extent, lock messages include a flag indicating if the request originated in a nonblocking LCU entry.

Nonblocking LCU entries can acquire free locks both in read or write mode. After acquisition, the uncontended LCU entry is deallocated as ordinary, and can be safely reused for a different lock. Alternatively, if a lock is already taken in read mode, the LRT can grant the lock in "overflow" mode to a nonblocking LCU entry. In this case, the LRT responds with a GRANT message, while increasing its *Reader_cnt* value, but without modifying the queue pointers or forwarding the request. The *Reader_cnt* field indicates the number of overflowed readers that have acquired the read lock but are not part of the queue. Overflowed readers receive the lock grant without the need to add themselves to the queue, and remove their LCU entries. When they release the lock, the LRT will determine that they are overflowed readers -since they are not recorded as the queue head when the message is received- and decrease the counter.

This mechanism still suffers from starvation with highly contended locks. A nonblocking LCU entry might never find the lock free (or only accessed by readers, if in read mode) and have to wait forever. To prevent such starvation, the LRT implements a *reservation* mechanism. When a request from a nonblocking LCU entry is replied with a RETRY, the requestor is recorded in the "reservation" tuple in the LRT (the thread and LCU IDs). From this moment, no further requests are served by the LRT, only those iterative requests received from the reservation holder. When the lock eventually becomes free, it can be acquired by the reservation holder. A timeout prevents this reservation from blocking the system, for example after a trylock expiration.

Although a single nonblocking entry per LCU is enough to guarantee forward progress, we implemented in our model three types of LCU entries with different requirements:

- *Ordinary*: Normal entries as considered initially; they can contain lock entries in any status.
- *Local-request*: Nonblocking entries reserved for requests from the local thread.
- *Remote-request*: Nonblocking entries reserved to serve remote releases, i.e. a RELEASE from a migrated thread.

### E. LRT Overflow

Since LRT entries control the status of taken locks, their data can never be cleared until lock release. We use the main memory as a backup for LRT overflow. Each LRT is assigned a preallocated hash-table in main memory for overflowed lines. When a request for a new entry arrives at a full LRT, a victim entry is selected and sent to the overflow table. The LRT contains an *overflow* flag indicating if there are overflowed entries in main memory, a counter for these entries and a pointer to the overflow table.When a request is received with *overflow* = *true*, the LRT logic must check the memory table if the address is not found in the LRT. When an overflowed entry is accessed again, it will be brought back from memory to the LRT, with the corresponding replacement, if required. When all overflowed entries are brought back, the *overflow* flag is cleared. Software exceptions are used in the unlikely event of having to resize the table.

Similar offloading mechanisms have been proposed for other hardware structures [5], [43]. Our simulations show that a 16-way associative LRT with 512 entries did not suffer from significant overflow problems and hash table resizing was never required.

### F. Virtual memory and paging

Our locking mechanism works with physical addresses. Therefore, changes in the virtual memory map might lead to erroneous behavior, in particular when a virtual page with taken locks is paged out and relocated back in a physical location where different locks overlap.

Paging operations with locks involved require support from the OS. When a virtual page with locks is taken out, the OS code that handles the TLB invalidation mechanisms must also handle the lock queues:it is necessary for the OS to invalidate existing lock queues for the given page, removing all of its LCU entries. When a queue is invalidated, the Head lock owner shifts to uncontended mode, with its entry safely cleared. Additionally, if multiple readers share a lock along the queue, they must be recorded as overflowed readers in the LRT when their entries are removed. Finally, the LRT entries should include ordinary process identifiers in order to determine the appropriate mapping when the page is brought back from disk.

Memory paging is a highly implementation-dependant topic, and we leave it open. Our simulated models do not support memory paging operations.

## IV. Evaluation

Our proposal has been implemented in GEMS [26], running on top of the full-system simulator Simics [25]. We used a simple in-order processor model issuing 4 instructions/cycle, with 64KB L1 caches. We implemented the LCU in two different 32-core machine models, labeled A and B.

Model A (*in-order*) implements single-core chips interconnected using GEMS's hierarchical switch topology

| Parameter | Model A | Model B |
|---|---|---|
| Chips | 32 | 4 |
| Cores | 32 $(32 \times 1)$ | 32 $(4 \times 8)$ |
| L1 size (KB)(I+D per core) | 64+64 | 64+64 |
| L2 size (KB per chip) | 1024 | 8 banks $\times$ 256 |
| L1 access latency (cycles) | 3 | 3 |
| L2 access latency (cycles) | 10 | 16 |
| Local mem. latency (cycles) | 186 | 210 |
| Remote mem. latency (cycles) | 186 | 315 |
| LCU entries | 8+2 | 16+2 |
| LCU lat (cycles) | 3 | 3 |
| LRTs | 32 | 8 |
| per-LRT entries | 512 | 512 |
| LRT latency | 6 | 6 |

Figure 8. Model parameters

network with a MESI coherence protocol. Although GEMS does not implement a model for a global bus, this approximates it by guaranteeing a global order for coherence, data, and lock requests at the higher level of the hierarchy of switches.

Model B (*m-CMP*) implements a multi-CMP system based on the Sun T5440 Enterprise Server, [37]. This system is composed of 4 Niagara [19] T2Plus chips. Each of them contains 8 cores with private L1 caches, a shared 8-banked L2 cache with interleaved memory requests, 2 memory controllers, and 4 Coherence Units (CUs) that provide a unified view of the system memory across the multiple chips. The four chips are interconnected via 4 coherence hubs, one for each of the per-chip CUs, which do not provide any global order for coherence or locking requests. In this case, we used GEMS with the Garnet [2] network simulator. We changed the coherence mechanism to a hierarchical MSI-MOSI protocol, and did not implement SMT in the processing cores.

The parameters of each model are presented in Figure 8. The latencies in the Table are presented in absence of network or coherence congestion, with each value including the miss penalty for lower cache levels and the network round-trip latency. Latencies for model A resemble a SunFire E25K server, but this model can also represent the performance on a bus-based single CMP system with private L1's and L2's. Latencies for model B are derived from the 1.4 GHz version of the T5440 server, as presented in [36].

In both systems each core implements its own LCU containing 8 or 16 ordinary entries, plus 1 *local-request* and 1 *remote-request* entries to guarantee forward progress. Our model implemented one LRT module per memory controller, with 512 entries, 16-way associative.

Our evaluation benchmarks include a lock transfer time microbenchmark, benchmarks from lock-based Transactional Memory systems and applications from the Parsec [4] and Splash-2 [41] benchmark suites. The details are presented next.

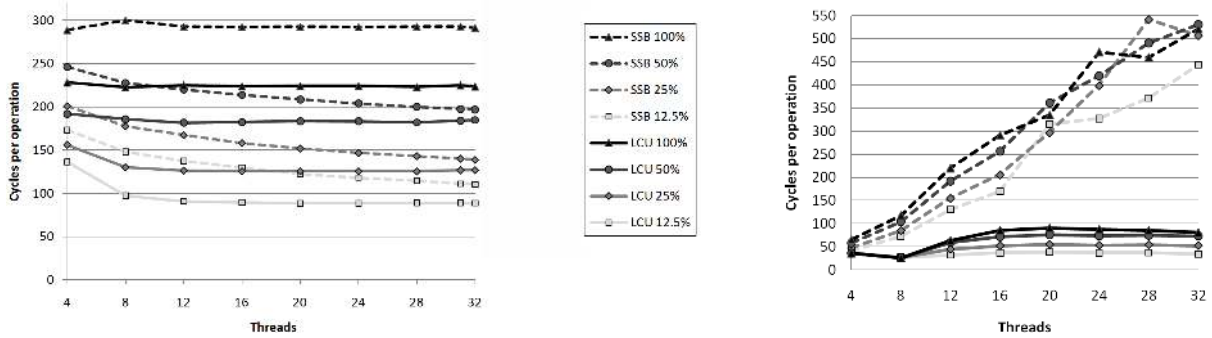
### A. Lock transfer time

Our first set of results use a synthetic benchmark based on ones used in earlier work, [15] and [18]. Multiple threads iteratively access the same critical section (CS), protected by a single lock. The CS is short, it only performs a few arithmetic operations, so the lock handling time dominates the operation. We measure the number of cycles required to run 50 000 iterations, averaged across multiple runs, and calculate the time per CS in cycles. We compare our proposal with different software and hardware lock implementations. Our software locks include *TAS* and *TATAS* locks, the base *MCS* queue-based lock and *MRSW* [27] which is the reader-writer version of MCS. We also evaluated the Synchronization State Buffer [43] (*SSB*) hardware. This model accelerates the lock handling, but does not build a requestors queue to reduce lock transfer time, and does not provide any fairness guarantees. When possible, we considered different rates of readers and writers accessing the CS, with multiple readers being able to access in parallel.

Figure 9 compares our system with the SSB model. The thread count varies from 4 to 32. In the *in-order* model (a) the performance of both systems is close to each other. However, in the mutual exclusion case (100% write locks) the LCU still outperforms SSB in 30.6% on average, due to the longer transfer latency in SSB. When we consider RW-locks, the average critical section access time decreases in both models with the reader proportion, as multiple readers access the CS concurrently. The SSB does not provide fairness, so readers are allowed to take the lock when it is in read mode. This increases the performance with the number of threads, at the cost of starving writers.

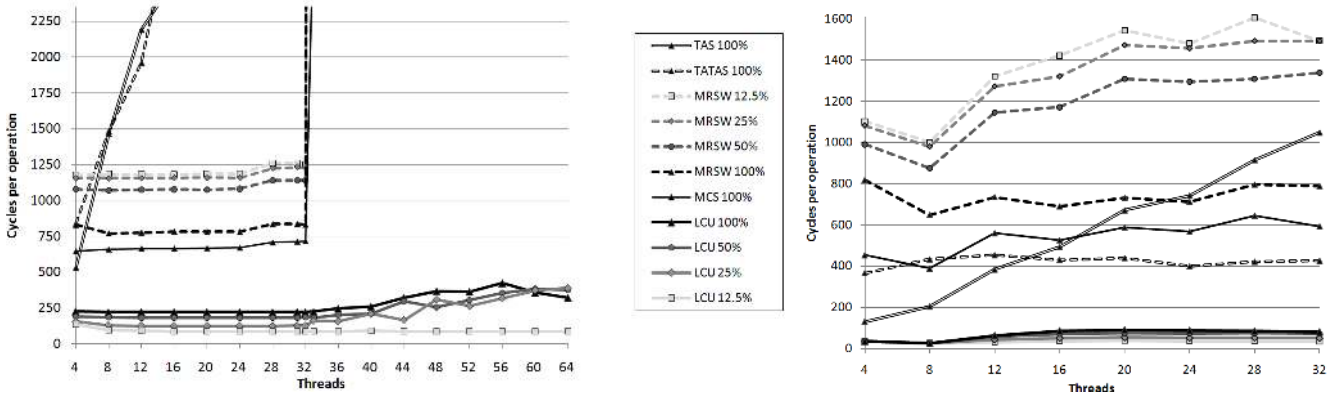
Figure 9b shows a radically different behavior in the multi-CMP model B. The repeated remote-retry mechanism of the SSB saturates the bandwidth of the inter-chip links, leading to poor performance. By contrast, the local-spin of the LCU maintains the desired behavior, with a slight performance drop when the number of threads exceeds the 8 cores in a chip and inter-chip links start to be used.

Figure 10 compares software locks and the LCU. The LCU model behaves smoothly with more threads than cores, due to the integrated starvation detection mechanism. Queue-based locks provide an almost constant performance up to 32 threads but after that, the starvation anomaly described before dramatically increases the lock transfer time. The LCU outperforms the software *MCS* proposal in more than $2\times$ in both models. This is because, even with the software queue, the coherence operations involved in the lock transfer require a remote coherence invalidation, followed by a subsequent request. The reader-writer variant, *MRSW*, even in the 100% writers case does not perform well due to the increased number of required operations. Moreover, as the readers rate increases, the average time per operation increases too. This is due to coherence congestion in the reader counter




(a) Model A, *in-order* (b) Model B, *m-CMP*

Figure 9. CS execution time including lock transfer, LCU vs SSB. Labels indicate the proportion of write accesses, mutual-exclusion is 100%.




(a) Model A, *in-order* (b) Model B, *m-CMP*

Figure 10. CS execution time including lock transfer, LCU vs SW locks. Labels indicate the proportion of write accesses, mutual-exclusion is 100%.

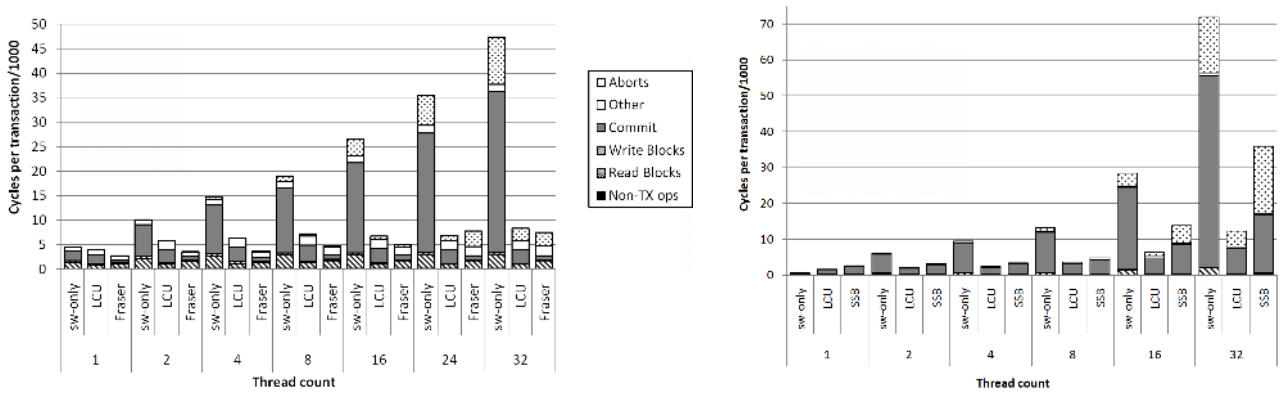
contained in the lock, which has to be modified atomically twice per reader (incremented and decremented). Due to this effect, the LCU obtains an average speedup of $9.14\times$ for the 75% read case. Finally, contended locks (TAS and TATAS) in the model A suffer from strong congestion as the number of threads increases. By contrast, in the model B, the hierarchical coherence protocol favors the unfair lock transfer between threads in the same chip, what maintains throughput (as observed in the figure) but starves remote requestors.

### *B. Fine-grain locking: STM benchmarks*

While Transactional Memory (TM, [14]) liberates the programmer from the complexity of locking, many Software TM (STM) systems are internally based on fine-grain locking ([10], [11], [40], among others). Dice and Shavit discuss in [9] the benefits of RW-locking in STM. They highlight its simplified design, implicit privatization safety ([38]), support for irrevocable transactions and strong progress properties. However, they find an excessive performance degradation on reader-locking, and propose a new lock to reduce it. They propose their model for single-chip systems, since the performance in multi-chip systems is poor. In this work, we perform evaluations in the more adverse multi-chip scenario.

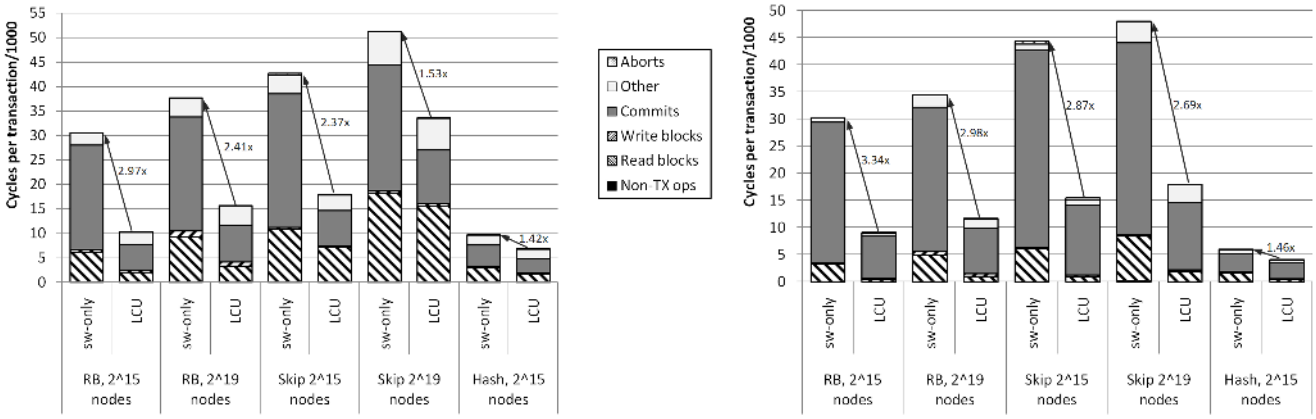
The LCU reduces the performance penalty of RW-locking in STM, and for evaluating it we used a RW lock-based implementation of Fraser's OSTM [12]. We compared the performance of this base system (labeled *sw-only*) against the same model using LCU RW-locks. As a reference, we also provide performance data of Fraser's non-blocking OSTM (labeled *Fraser*). This is a nonblocking system which does not read-lock the read set during commit. Due to this use of invisible readers, it fails to support the privatization idiom. Consequently, its performance cannot be compared directly with the lock-based variants, but it provides some indication of the behavior of basic object-based STM systems.

We use three data-structure micro-benchmarks, frequent in the STM community: *RB-tree*, *skip-list* and *hash-table*. Multiple threads access the shared structure to perform a transactional operation, typically a search (read-only) or an insert/delete. We measure the average time to finish a transaction. Ideally, this value should remain constant as the number of threads increases. Due to the visible readers




(a) Model A, *in-order* (b) Model B, *m-CMP*

Figure 11. Scalability of the transactional system. Transaction cycle dissection of the RB-tree benchmark with $2^8$ maximum nodes.




(a) Model A, *in-order* (b) Model B, *m-CMP*

Figure 12. Transaction execution time, 16 threads and 75% of read-only transactions.

implementation in the lock-based versions, *RB* and *Skip* suffer from reader-locking coherence congestion in the root of the data structure, while the *hash-table* does not present such pathology.

The plots in Figure 11 study the scalability of the system. They show the transaction execution time for different models in the *RB-tree* benchmark with $2^8$ maximum nodes and 75% of read-only transactions, as the thread count increases. With a single thread, the *LCU* improves the performance of the base *sw-only* in a 10.8%. Fraser's nonblocking model outperforms the lock-based one, as its commit phase is much shorter, given the lack of reader locking. As the number of threads increases, the base *sw-only* model gets worse, due to the increase of the commit phase in which locks are acquired. This is specially true in the multi-CMP model with more than 8 threads, due to the large inter-chip latencies. The shared data structure suffers from reader congestion in the root, which affects the overall performance. The *LCU* model scales nicely, almost preserving the execution time. For high thread counts, the *LCU* performance is similar to that of the unsafe Fraser's system, and outperforms the SSB model.

Figure 12 shows the performance of the previous models with larger problems ($2^{15}$ or $2^{19}$ maximum number of nodes) with 16 threads. The *RB* and *skip* benchmarks behave as presented in Figure 11, with reader congestion in the root of the data structure. The faster implementation of reader locking in the LCU provides speedup values from $1.53\times$ to $3.35\times$. The *hash-table* does not have a single entry-point, but still, the speedup is at least $1.42\times$.

### C. Traditional parallel benchmarks

We finally evaluate the performance of the LCU with some traditional lock-based programs that do not rely on reader-writer locking. Figure 13 shows the execution time of the parallel sections of Fluidanimate, Cholesky and Radiosity from the Parsec and Splash benchmarks suites, only for the system in model A. We selected these applications for two reasons: they are lock intensive and present different

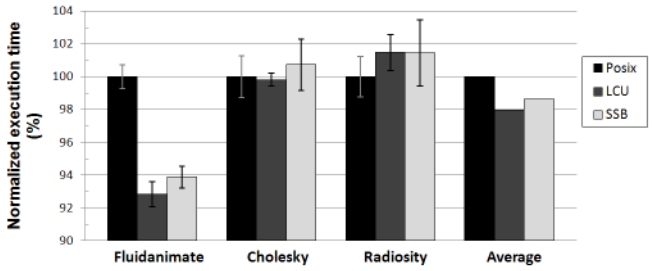



Figure 13. Application execution time.

behavior with respect to the locking pattern. We simulated several runs of each application with 32 (Fluidanimate) or 16 (Cholesky, Radiosity) threads and averaged the result for each one of three models: the original *Posix* mutexes in our Solaris system, the *LCU* and the *SSB*. The plot shows the resulting values, including the estimated error for a 95% confidence interval. Note that these plots show the overall application execution time, not only the lock-related code.

In Fluidanimate the LCU provides a speedup of 7.4% over the base model. This comes from faster lock transfers and finer-grain locking. While the original application uses a lock per each modeled *cell*, our version protects each *value* being updated within a cell with a dynamic lock. The direct lock transfer also allows the LCU to slightly outperform the SSB.

Cholesky is a matrix application which does not seem to be almost affected by the lock model, as all the results are within the estimated error range of the base software model. Notwithstanding, we can still observe that the introduction of the LCU does not harm performance.

Radiosity, by contrast, shows the opposite pattern. We identified the problem in that most lock accesses in the application are to the per-thread private task queues. Only when a thread finishes its work, it accesses remote queues to steal work to do, in a form of load balancing. In this case, the base software version maintains the frequently accessed lock line in the L1 cache, in a sort of *implicit biasing* (in opposition to the explicit biasing mechanisms developed for software locks [31]). By contrast, the LCU is required to repeat remote accesses, with a negative impact on performance. This effect could be also observed in the single-threaded case of Figure 12b. Such implicit biasing is inherent to coherence-based locks, such as software locks or tagged-memory systems, including QOLB.

To deal with this case, we have envisioned an additional hardware unit collocated with each LCU, called Free Lock Table (FLT). The FLT is used to "save" locks frequently accessed by a single thread, without their releases being remotely visible as long as no other requestor exists. Preliminary results, not presented in this paper, show that the FLT improves execution in the presence of such private locks, restoring the *implicit biasing* that our system lacks. Nevertheless, it can reduce the performance when ordinary, shared locks are used. Therefore, this new hardware would require a prediction mechanism or programmer hints to discern private and shared locks. We leave the FLT integration, including the predictor design, as an open area for future work. Despite the lack of the FLT, the geometric mean of the three application results shows an average speedup of 1.98% for the LCU model.

## V. Conclusion

The Lock Control Unit hardware mechanism has been proved to efficiently handle fine-grain reader-writer locking, improving existent software locks and previous hardware proposals. As far as we know, our mechanism is the first proposal that implements a fair reader-writer queue. The LCU provides more flexibility and robustness than previous proposals. In addition, the evaluation of the lock transfer time outperforms previous designs in more than a 30%, and application performance can be improved in more than a 7% from software locking. Moreover, our model makes emerging STM systems based on RW-locking competitive with different, unsafe approaches, despite the traditional cost of reader locking.

Several areas remain as future work. Mainly, a biasing mechanism, as discussed in Section IV-C, with an appropriate predictor to use it. Other interesting research lines would be the use of priorities for real-time systems, and the implementation of hierarchical locks for a multi-CMP system or a SMT multicore. Such locks would exploit the locality of threads in the same chip or SMT threads in the same core.

## Acknowledgment

The authors would like to thank the effort of the anonymous reviewers, who provided very valuable feedback. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2010-21291-C02-02 and TIN2007-60625, by the HiPEAC European Network of Excellence and by the European Commission FP7 project VELOX (216852).

## References

[1] A. Agarwal *et al.* The MIT Alewife machine: architecture and performance. ISCA'95: $22^{nd}$ Int. Symp. on Computer Architecture, 1995.

[2] N. Agarwal, L.S. Peh and N. Jha. GARNET: A detailed interconnection network model inside a full-system simulation framework CE-P08-001, Dept. of Electrical Engineering, Princeton University, Feb 2008

[3] R. Alverson *et al.* The Tera computer system ICS '90: $4^{th}$ Int. Conf. on Supercomputing. 1990.

[4] C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC benchmark suite: characterization and architectural implications. PACT'08: $17^{th}$ Int. Conf. on Parallel Architectures and Compilation Techniques,2008.

[5] M. Chiang. Memory system design for bus based multiprocessors. Ph.D. dissertation. Univ. of Wisconsin-Madison, Sept. 1991.

[6] W. Dally *et al*. The Message-Driven processor: A multicomputer processing node with efficient mechanisms. IEEE Micro, IEEE Computer Society Press, 1992, 12, 23-39.

[7] S. Keckler *et al*. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor ISCA'98: $25^{th}$ Int. Symp. on Computer Architecture, 1998.

[8] D. Dice, M. Moir and W. Scherer III. Quickly reacquirable locks. Available online at http://home.comcast.net/∼pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf.

[9] D. Dice and N. Shavit. TLRW: Return of the read-write Lock. TRANSACT'09: $4^{th}$ Workshop on Transactional Memory, 2009.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. DISC'06: $20^{th}$ Symp. on Distributed Computing, 2006.

[11] P. Felber, C. Fetzer and T. Riegel. Dynamic performance tuning of word-based Software Transactional Memory. PPoPP'08: $13^{th}$ Symp. on Principles and Practice of Parallel Programming.

[12] K. Fraser and T. Harris. Concurrent programming without locks. ACM Transactions on Computer Systems, Vol. 25, Issue 2, May 2007.

[13] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. SIGARCH Comput. Archit. News 17, 2 (Apr. 1989), 64-75.

[14] T. Harris, J. Larus and R. Rajwar. Transactional Memory. Morgan Claypool Synthesis Series, $2^{nd}$ endition. 2010.

[15] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. HPC'05: $11^{th}$ Int. Conf. on High Performance Computing, 2005.

[16] M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufmann. USA, 2008.

[17] H. F. Jordan. Performance measurements on HEP – a pipelined MIMD computer. ISCA'83: $10^{th}$ Int. Symp. on Computer architecture, 1983.

[18] A. Kägi, D. C. Burger, and J. R. Goodman. Efficient synchronization: let them eat QOLB. ISCA'97: $24^{th}$ Int. Symp. on Computer Architecture, 1997.

[19] P. Kongetira, K. Aingaran and K. Olukotun. Niagara: A 32-way multithreaded sparc processor IEEE Micro, IEEE Computer Society, 2005, 25, 21-29.

[20] O. Krieger, M. Stumm, R. Unrau and J. Hanna. A fair fast scalable reader-writer lock. ICPP'93: Intl. Conf. on Parallel Processing, 1993.

[21] J. Kuskin *et al*. The Stanford FLASH multiprocessor. ISCA'94: $21^{st}$ Int. Symp. on Computer architecture, 1994.

[22] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. Comput. Archit. News. Vol 25, 2 (May. 1997), 241-251.

[23] D. Lenoski *et al*. The Stanford Dash multiprocessor. IEEE Trans. on Computer. Vol 25, 2 (Mar. 1992), 63-79.

[24] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. SPAA'09: $21^{st}$ Symp. Parallelism in Algorithms & Architectures, 2009.

[25] P. S. Magnusson *et al*. Simics: A full system simulation platform. IEEE Computer, 35(2):50-58, Feb'02.

[26] M. M.K. Martin *et al*. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset. *Computer Architecture News (CAN)*, Sept. 2005.

[27] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems 9, 1 (Feb. 1991), 21-65.

[28] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. PPoPP'91: Symp. on Principles and Practice of Parallel Programming, 1991.

[29] Z. Radović and E. Hagersten Hierarchical backoff locks for nonuniform communication architectures. HPC'03: Symp. on High-Performance Computer Architecture, 2003.

[30] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. MICRO'01: $34^{th}$ Int. Symp. on Microarchitecture, 2001.

[31] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. OOPSLA'06: Object-Oriented Programming, Systems, Languages & Applications, 2006.

[32] B. Saglam and V. Mooney, III. System-on-a-chip processor synchronization support in hardware. DATE'01: Conf. on Design, Automation and Test in Europe (DATE'01), 2001.

[33] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. PODC '02: $21^{st}$ Symp. on Principles of distributed computing, 2002.

[34] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. PPoPP'01: $8^{th}$ Symp. on Principles and Practice of Parallel Programming, 2001.

[35] S. Scott. Synchronization and communication in the T3E multiprocessor. ASPLOS'96: $7^{th}$ Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996.

[36] D. Sheahan. Memory and coherency on the UltraSPARC T2 Plus Processor Corporate blog entry. *http://blogs.sun.com/deniss/entry /memory_and_coherency_on_the*. April 2008.

[37] Sun Enterprise. Sun Sparc Enterprise T5440 Server Architecture. White Paper. July 2009

[38] M. F. Spear, V. J. Marathe, L. Dalessandro and M. L. Scott. Privatization techniques for software transactional memory. PODC'07: Symp. on Principles of Distributed Computing, 2007.

[39] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. ASPLOS'09: $14^{th}$ Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 2009.

[40] C. Wang, W. Chen, Y. Wu, B. Saha, A. Adl-Tabatabai. Code generation and optimization for Transactional Memory constructs in an unmanaged language. CGO'07: Int. Symp. on Code Generation and Optimization, 2007.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. The SPLASH-2 programs: characterization and methodological considerations ISCA'95: $22^{nd}$ Int. Symp. on Computer Architecture, 1995.

[42] L. Zhang, Z. Fang and J. Carter. Highly efficient synchronization based on Active Memory Operations. IPDPS'04: Int. Parallel and Distributed Processing Symposium, 2004.

[43] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. ISCA'07: $34^{th}$ Int. Symp. on Computer Architecture, 2007.