Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations

Eduardo José Gómez-Hernández and Juan M. Cebrian and Rubén Titos-Gil and Stefanos Kaxiras and Alberto Ros

University of Murcia, Spain and Uppsala University, Sweden

UPPSALA UNIVERSITET

UNIVERSIDAD DE MURCIA

Introduction

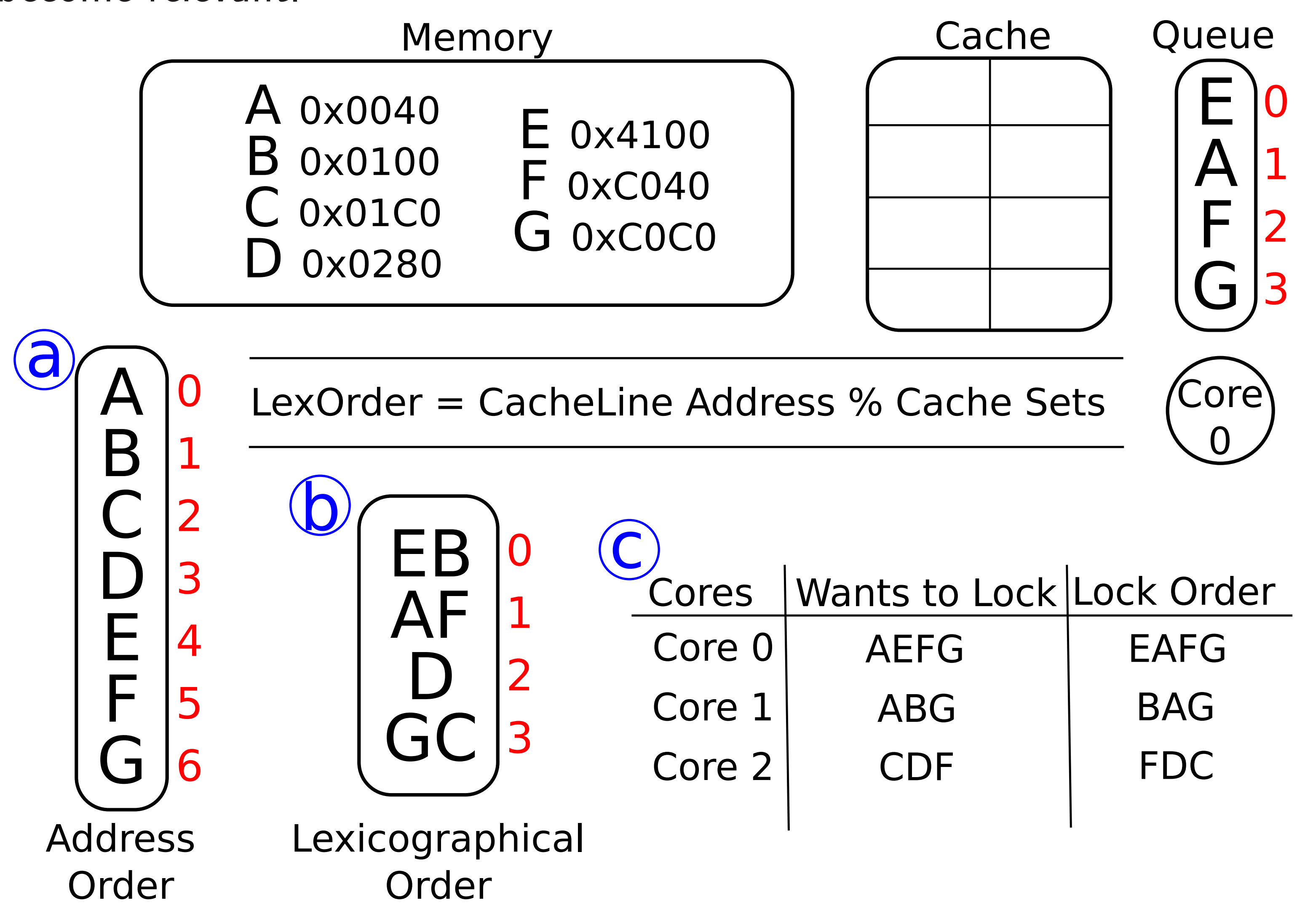
Synchronization, critical sections, atomicity, race conditions, among others, are some of the needs that current multi-threaded applications running on top of multi-core processors have. These needs slowdown programs that otherwise could run very fast, decreasing their scalability.

Motivation

- While prior work in multi-address atomicity exists, they did not consider several deadlock scenarios when distributively locking several cachelines, nor provide a safe, non-speculative, solution.
- Atomic RMW instructions are the most efficient way to perform an atomic update of a variable since they are genuinely hardware operations.
- Non-blocking algorithms rely on directly using atomic read-modify-write (RMW) primitives natively provided by the hardware. Commonly, a compare-and-swap(CAS) instruction.
- A vector instruction computes several data elements exploiting data-level parallelism. Therefore, without multi-address atomics, it is not possible to provide vector atomic operations in a SIMD processor.

Lexicographical Order

Address order does not take into account some hardware structures like the cache. This is not a problem when using software locks. However, when using the cache itself to implement the lock system, its size and structure become relevant.



Cores	Wants to Lock	Lock Order
Core 0	AEFG	EAFG
Core 1	ABG	BAG
Core 2	CDF	FDC

Figure 1: Address Order vs Lexicographical Order

This order forces locks to be held from the top part of the cache to the bottom. In the example (Figure 1), cores will crash in the upper part of the cache.

Lexicographical reOrder Unit

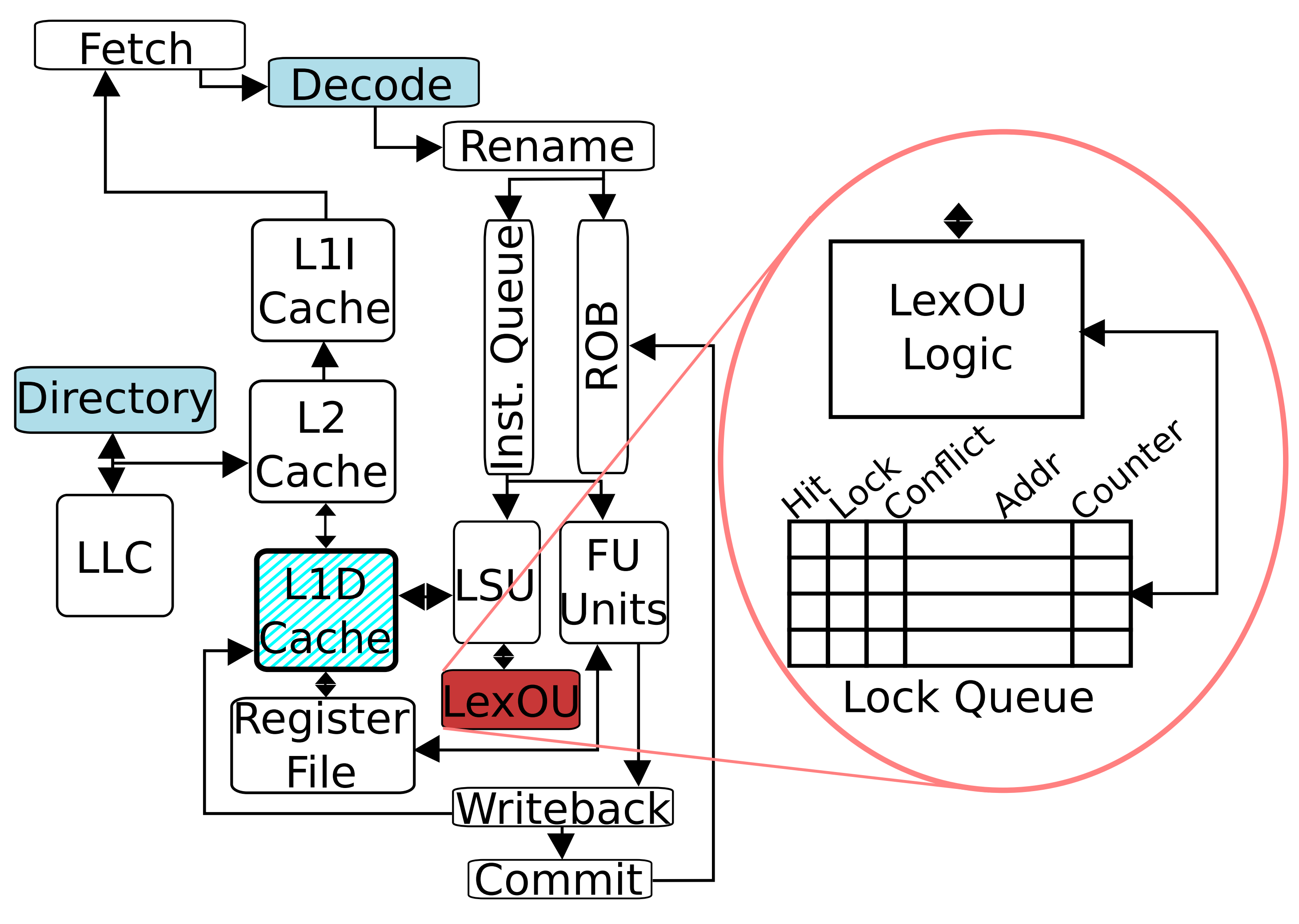


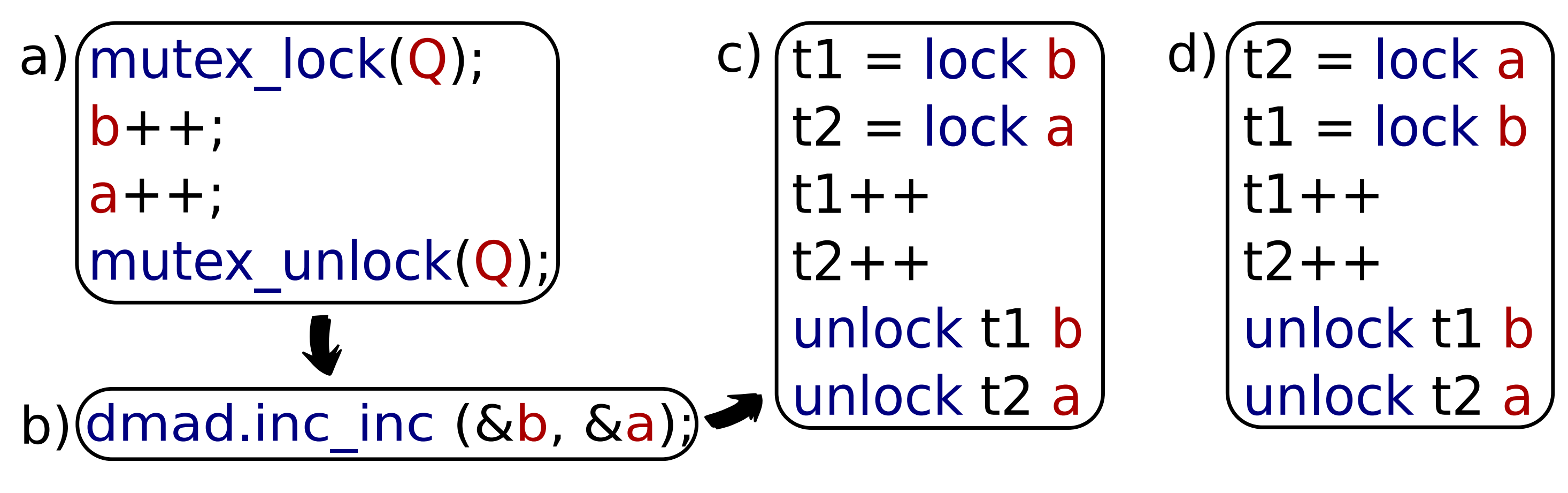
Figure 2: Microarchitectural changes

The Lexicographical reOrder Unit (LexOU), is the main structure that manages the new locking system for multiaddress atomics (MAD). Built from a small memory that acts like a sorted queue (Figure 2), it handles the reordering of the locks to guarantee progress.

Small changes are required to the directory and the L1 cache to allow the hardware locking mechanism and prevent deadlocks between different cores.

MAD Atomics

Multiaddress atomics (MAD Atomics) are very similar to current x86 atomics but allowing multiple addresses per instruction. All the addresses and values are loaded into general-purpose registers to prevent accessing memory after requesting the lock.



a)
```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

b)
```
dmad.inc_inc (&b, &a);
```

c)
```
t1 = lock b
t2 = lock a
t1++
t2++
unlock t1 b
unlock t2 a
```

d)
```
t2 = lock a
t1 = lock b
t1++
t2++
unlock t1 b
unlock t2 a
```

Figure 3: Example of a MAD atomic and its microcode

In Figure 3, a typical mutex lock with two increments is replaced by a MAD atomic that would execute the increment atomically in both variables. Assuming that address ‘a’ has a smaller lexorder than address ‘b’, the lock of address ‘a’ will be performed earlier than address ‘b’. Note that this is only from the memory viewpoint, no instruction reordering is happening.

Results

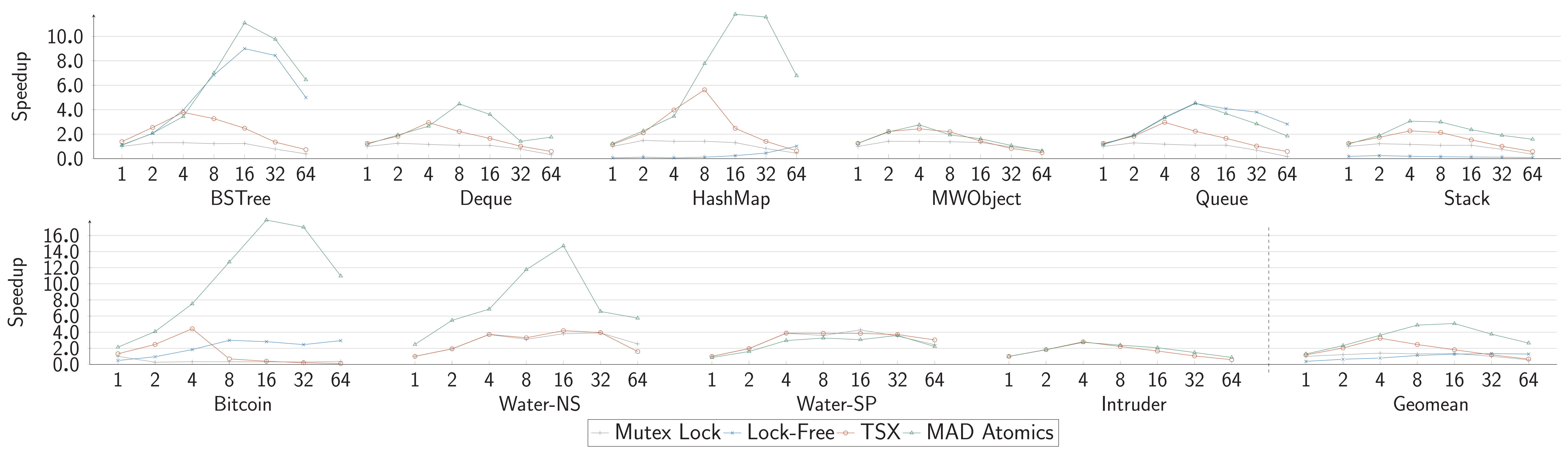


Figure 4: Scalability of the benchmarks (1 to 64 cores) . Each version is normalized to the lock version running a single thread.

ECHO, ERC Consolidator Grant (No 819134)   Accepted in MICRO-54 2021   Mail: eduardojose.gomez@um.es