CaCAO: Complex and Compositional Atomic Operations for NoC-Based Manycore Platforms

Sven Rheindt(✉), Andreas Schenk, Akshay Srivatsa, Thomas Wild, and Andreas Herkersdorf

Chair for Integrated Systems, Technical University Munich, Munich, Germany
{sven.rheindt,andreas.schenk,srivatsa.akshay,thomas.wild, herkersdorf}@tum.de

Abstract. Tile-based distributed memory systems have increased the scalability of manycore platforms. However, inter-tile memory accesses, especially thread synchronization suffer from high remote access latencies. Our thorough investigations of lock-based and lock-free synchronization primitives show that there is a concurrency dependent crossover point between them, i.e. there is no one-fits-all solution. Therefore, we propose to combine the conceptual advantages (no retries and lock-free) of both variants by using dedicated hardware support for inter-tile atomic operations. For frequently used and highly concurrent data structures, we show a speedup factor of 23.9 and 35.4 over the lock-based and lock-free implementations respectively, which increases with higher concurrency.

Keywords: Atomic operations · Remote synchronization
Compare-and-swap · Distributed shared memory · Network-on-Chip

1 Introduction

In the last decade, the power wall limited the increase of processor frequency. With the advent of mainstream multicore platforms, this technological problem was tackled by distributing applications over multiple cores that still used one common memory. Further scalability was introduced by transitioning to distributed shared memory architectures to lower memory access contention and hotspots [1]. An example is our hybrid tiled architecture depicted in Fig. 1, with Network-on-Chip (NoC)-based interconnect and several bus-connected cores per tile, sharing a tile local memory.

Most distributed memory platform make use of the Message-Passing-Interface (MPI) programming model, but there is still a demand for shared memory programming due to its ease of use [1,2]. But, multicore architectures in combination with shared memory programming introduce the challenge of providing atomic memory accesses to local/remote shared data structures. Thread

© Springer International Publishing AG, part of Springer Nature 2018
M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 139–152, 2018.
https://doi.org/10.1007/978-3-319-77610-1_11

synchronization is even more challenging for distributed shared than for purely shared memory systems, since the widely used NoC interconnect does not inherently allow for atomic memory accesses. Additionally, distributed shared memory architectures exhibit non-uniform memory access (NUMA) properties. Application performance therefore highly depends on data-to-task locality and efficient synchronization primitives.

Synchronization can be categorized into three classes of atomic primitives. Lock - based primitives atomically lock the critical section. Classical locks are often implemented using hardware support in the form of test-and-set (TAS) or compare-and-swap (CAS). Software based lock-free mechanisms use general-purpose atomic operations like CAS or linked-load/store-conditional (LL/SC), which are lock-free and provided by the underlying hardware often as ISA extensions. Hardware based primitives use so called special-purpose atomic operations to implement the whole critical section in dedicated hardware without using locks. An example is the class of fetch-and-ops [3–6].

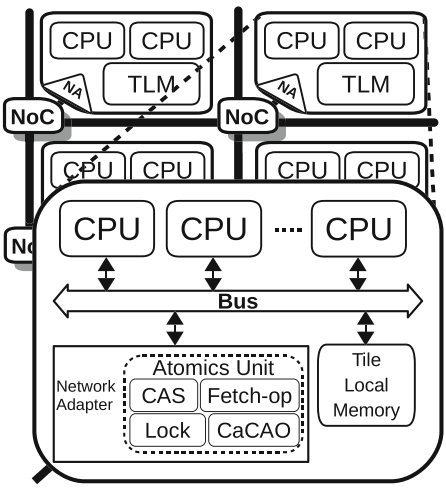


Fig. 1. Hybrid interconnect distributed shared memory platform with atomics unit

Due to the NUMA properties of distributed shared memory architectures, different aspects of synchronization get a new weight. If, for example, an application loops over a CAS until it is successful, the retry penalty/NoC travel time for a failed CAS is much higher for remote than for local operations. Even though a purely lock-based or lock-free software implementation might be favorable for a given concurrent data structure in a conventional bus-based system, totally different results might be true for distributed shared memory architectures.

In this paper, we therefore investigate the effects of lock-based and lock-free software synchronization primitives on a distributed shared memory architecture. We further propose special-purpose hardware implementations for efficient remote atomic operations. We couple the advantages of both lock-based and lock-free primitives with remote execution of the critical section in dedicated hardware to tackle the challenge of NUMA operations on distributed shared memory platforms. We call this combination complex and compositional atomic operations (CaCAO). We compare our dedicated hardware implementation to lock-based and lock-free software based variants that use partial hardware support.

The rest of the paper is organized as follows. We describe the related work in Sect. 2. In Sect. 3, we analyze and compare existing synchronization primitives and propose the use of complex and compositional atomic operations.

Architectural details of our implemented hardware prototype are given in Sect. 4. In Sect. 5, we present and discuss our experimental results, before we finally conclude this paper and give an outlook to future work in Sect. 6.

2 Related Work

On the one hand, the trend of lock-based synchronization leads towards efficient lock implementations [1,7–9] without support for general purpose atomic primitives. On the other hand, many - but not all - multicore platforms provide lock-free synchronization capabilities [2–4,6,10]. For example Mellanox, earlier Tilera, - who provides one of the modern tile-based architectures - supports the CAS primitive only for their GX platform [2], not for the Pro platform [11].

Authors in [12] developed the MCS-lock to overcome the performance bottleneck and other limitations of simple, ticket and various queue based spinlocks [4,7,10,13]. Through spinning on local variables only, they require $\mathcal{O}(1)$ network operations for acquiring a remote lock. The MCS-lock can be efficiently implemented in software, but needs an atomic swap operation for basic functionality and the CAS primitive to provide full features like FIFO ordering and starvation freedom. This was adopted by the authors of [7,8] in their two consecutive works on efficient lock-based synchronization for NoC-based distributed shared memory systems. They transitioned from optimized simple and ticket spinlocks [8] to MCS-locks [7]. Their lock implementations use a hardware loop for local polling until acquisition. An atomic fetch-and-inc/dec unit is used to integrate the ticket spinlock and semaphores. They purely focus on optimizing locks by performing these atomic operations exclusively on special globally addressable registers in their synchronization unit and not on arbitrary memory locations. Apart from the swap instruction, they do not support atomic operations on memory, especially no lock-free primitives. Authors in [9] basically adopt the same idea of just optimizing lock implementations for distributed memory. Through optimized lock queue handling, they avoid head-of-line blocking of independent synchronization requests.

With the lock-free universal primitives CAS and LL/SC, it is possible to emulate all other atomic primitives or transform lock-based mechanisms into lock-free ones [3–6]. Authors in [4] convert operations into purely lock-free variants using the universal primitive LL/SC and the fetch-and-op primitives, that are common in modern multicore systems. However, they admit that in general many lock-free mechanisms can get quite complex. Authors in [3] use a standard 2D-mesh NoC interconnect with one core per tile. They suggest a rather complex combination of CAS as in-cache implementation together with a write-invalidate coherence policy and a load-exclusive coherence policy extension to minimize the CAS operations on memory. Furthermore, they recommend a hardware based serial number extension for the CAS primitive to tackle the ABA problem and also suggest a fetch-and-add primitive for efficient counters.

To our knowledge, the related work either optimizes lock-based synchronization with efficient lock implementation or investigates purely lock-free variants [14,15]. Few provide the special purpose fetch-and-increment primitive for

remote operations on NoC-based systems [1,3]. However, no dedicated hardware support for more complex special purpose atomic operations are provided by state-of-the-art distributed shared memory systems.

3 Complex and Compositional Atomic Operations

We classify synchronization primitives into three main categories, that use different amounts and kinds of hardware support:

(α) Software lock-based: using (efficient) hardware lock support
(β) Software lock-free: using hardware CAS or LL/SC
(γ) Dedicated hardware for whole critical section (wait-free)

In this paper, we first describe and compare (α) and (β). Then we propose to combine their conceptual advantages by using dedicated hardware support (γ) for complex and compositional atomic operations (CaCAO approach).

A main attribute of synchronization primitives is the number of retries. An operation has zero retries, if the read-modify-write cycle is non-conditional. This means, if there is no interfering concurrency on the data structure that makes a retry of the operation necessary. This holds true, e.g. for the fetch-and-ops, but not for the CAS, since the latter only writes back if the read value did not change in the meantime.

3.1 Comparison of the Synchronization Primitives (α) and (β)

(α) A **lock-based software** implementation of a given function locks the critical section (CS) that has to be performed atomically. It uses hardware support like test-and-set to acquire the lock. Whereas - by design - the critical section inside the lock has zero retries, the lock acquisition itself does not and is not even wait-free, since no upper bound for the number of retries until lock acquisition can be given. However, much research has already been done to provide efficient lock implementations [1,7–9]. In the following, when we refer to lock-based software primitives, we therefore use a variant with efficient hardware look support.

(β) **Lock-free software** implementations of a given function can be achieved with the lock-free universal primitives CAS or LL/SC, that need to be provided by the hardware. As their names already suggest, their read-modify-write cycle is conditional and they therefore are not retry-free. A software loop is needed to repeat the operation until it is successful. The best-case execution time of a lock-free software primitive can be one try, if no other party interfered in the meantime. However, the average execution time is heavily dependent on the concurrency and the worst-case can even lead to starvation. Therefore, these lock-free implementations are not guaranteed to be wait-free.

A theoretical comparison between (α) and (β) shows that the best-case execution time of a lock-free software variant is approximately equal to the execution time of the critical section of a lock-based variant without the time for acquiring and releasing the lock. The average time of the lock-free variant is

dependent on the interfering concurrency and the thereby necessary retries (as well as other factors like run-time background traffic). Whereas for the lock-free variant the average execution time is linear in the number of retries, the average lock-based execution time is linear in the number of concurrent contenders for the lock.

This comparison shows that - whereas in the best-case a lock-free implementation is always better - in the average case there can be a cross-over point between the lock-based and lock-free implementations. If the concurrency dependent retry rate is greater than a threshold, the lock-based implementation yields better performance and vice versa. A design time decision between (α) and (β) would be necessary by the programmer. In Sect. 6 we talk about a more dynamic decision making as future work.

3.2 CaCAO Approach (γ)

If one only has (α) and (β), no one-fits-all solution would be available. But we overcome the deficiencies of both software lock-based (α) as well as software lock-free (β) implementations by combining their conceptual advantages: zero retries and lock-freeness.

We propose a **dedicated hardware** (γ) implementation that outsources and atomically executes the whole critical section in a dedicated hardware module (near the memory where the shared data is stored), thereby guaranteeing zero retries by design. Since an upper bound for the execution time can be given, this approach is not only lock-free, but also wait-free.

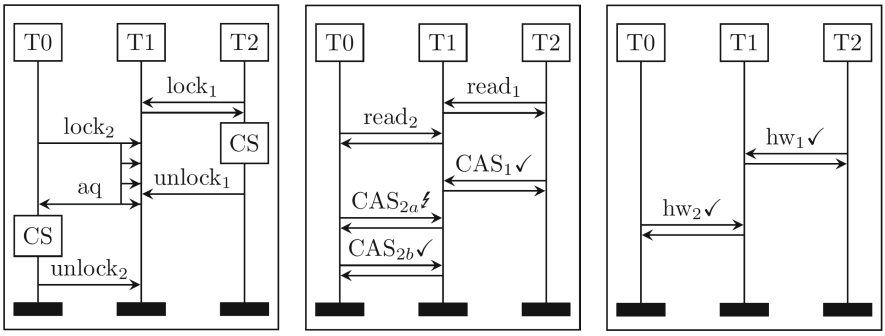
In the best-case, the whole execution (NoC travel time plus atomic read-modify-write cycles) of the lock-free primitives (β and γ) is approximately as small as the minimal time for lock acquisition of the lock-based variant (α). More importantly, the average and worst case times for the proposed dedicated hardware solution (γ) do not rise much, since no interfering concurrency is possible and therefore no retries are necessary. Especially for remote accesses the atomic read-modify-write cycles with constant duration after bus grant are much shorter than the travel time over the NoC. Even if there are several concurrent contenders, they cannot interfere one another due to the atomic read-modify-write cycles in hardware, thereby guaranteeing wait-free operation.

This approach has general validity and outperforms the software lock-based as well as lock-free variants by design. This local or remote site execution in a dedicated hardware module (CaCAO approach) helps to tackle the data-to-task locality problem of distributed shared memory architectures.

In contrast to software based lock-free implementations that might get quite complex [4], CaCAO does not need atomic operations inside the critical section, since the atomicity is intrinsically provided by the dedicated hardware module.

However, the disadvantage of this approach is its very application specific nature due to the need of implementing each needed functionality in dedicated hardware.

In future work, we plan to further extend the functionality and complexity of CaCAO. Because of the compositional nature of this approach, various already



(a) lock-based (b) lock-free (c) dedicated hardware

Fig. 2. Message sequence charts for three types of shared counter implementations

implemented as well as future functionalities of our dedicated hardware module do and can reuse the same hardware blocks.

For the validation of the concept, we implemented and investigated the widely used fetch-and-add operation in the use case scenario of a shared counter as well as dedicated hardware implementations of atomic linked-queue enqueue/dequeue operations. For both scenarios, we implemented and evaluated all three variants (α), (β) and (γ) on our FPGA prototype as described in Sect. 5. In the following, these scenarios are briefly explained and used to show the key differences of the three synchronization variants.

Shared Counter Scenario. The pseudo source codes for the three synchronization types for the shared counter for N cores on several tiles are given in Fig. 3. In Fig. 2 the message sequence charts are given.

(α) The lock-based software implementation first acquires a lock (line 1 in Fig. 3) over the NoC, then remote reads the counter value (2), increments it (3) locally and writes the value back to memory (4) before unlocking (5) the critical section. As can be seen in Fig. 2(a), the critical section (CS) inside the lock has zero retries, whereas the lock acquisition itself may require several retries. An efficient lock implementation, as shown for $lock_2$, lowers these through local polling in hardware at remote site. For (α), hardware support is only provided for efficient locks.

```
shared_counter_locked(*lck,*cnt)
// HW support for lock
1  lock(lck);
2  tmp = *cnt
3  tmp++;
4  *cnt = tmp;
5  unlock(lck);

shared_counter_lock_free(*cnt)
// HW support for CAS
1  tmp = *cnt;
2  do{
3    old = tmp;
4    tmp = CAS(cnt, old, old+1);
5  }while(tmp != old);

shared_counter_HW(*cnt)
// HW support for whole CS
1  fetch_and_inc(cnt);
```

Fig. 3. Pseudo code of shared counter implementation for the three synchronization types

(β) The lock-free software based implementation reads the counter value non-atomically (1). Then it performs a compare-and-swap on the counter with the incremented old value (4). It is successful if the counter value did not change in the meantime (5). Otherwise, the compare-and-swap is reissued with the updated old value (2–4). This can be seen in Fig. 2(b). Between the read_2 (1) and the CAS_{2a} (4) from T0 to T1, another CAS_1 from T2 to T1 happens and updates the shared value, so that the CAS_{2a} fails and needs a retry (CAS_{2b}).
(γ) The dedicated hardware implementation uses the atomic fetch-and-increment mechanism. It is retry-free by definition since the read-modify-write operation is performed atomically in hardware at remote site by sending the whole operation there (Fig. 2(c)). It is clear that the fetch-and-add operation is standard in many CPU ISAs. However, these mostly support only local and no remote memory operations.

Linked Queue Scenario. A more advanced example is the enqueue and dequeue operation of a linked-list queue. Without loss of generality, we limit this scenario to tail-enqueue and head-dequeue operations. The tail-enqueue operation has to atomically update the tail.next pointer only if the read value of tail.next is still NULL, meaning no other enqueue operation happened in the meantime. The second step is to set the tail pointer to the new element non-atomically. Until this operation is finished, all other enqueue attempts result in a fail and retry.

The head-dequeue operation has to atomically update the head pointer, which is successful if no other dequeue operation happened in the meantime.

Analog to the shared counter example, (α) only uses hardware support for efficient lock implementation. (β) is in need for hardware CAS support and (γ) needs dedicated hardware for the whole enqueue/dequeue operation.

4 Implementation Aspects

We implemented a configurable, resource reusing hardware module for local and remote atomic operations. As depicted in Fig. 1, it is inside of a modular network adapter connected to the tile local bus. Besides the atomics unit, the network adapter provides several other functionalities, like e.g. remote reads and writes, direct memory accesses, etc. The extension towards atomic operations of the network adapter consists of submodules for sending (TX) and receiving (RX) atomic operation requests and the atomics unit containing several operations.

Architectural details. As we use a hybrid NoC-bus-based architecture, the atomics unit is - as part of the network adapter - connected to the bus. To perform any atomic operation on memory, it (1) first blocks the bus for any other accesses, (2) executes the (conditional) read-modify-write cycle on the memory connected to the bus, before (3) unlocking the bus again. The modify or conditional write operations of step (2) are performed or evaluated in the hardware unit to minimize calculation and network time by processing at the remote tile. This approach basically allows for arbitrary complex operations in

step (2), that are completely atomic through the exclusive bus usage between (1) and (3). However, in this paper, we limit ourselves to the set of atomic primitives described below.

These network triggered atomic operations concur with atomic operations triggered by local processors and are sequentialized through exclusive usage of the bus. This modular design with standard interfaces therefore ensures high adaptability and integrability into existing systems. On our platform as described in Sect. 5, only support for local atomic swap and CAS is given as ISA of the cores. However - as for most systems - no support for inter-tile remote atomic operations is given.

As the atomic primitives described in the next paragraph require a response or acknowledgement, we implemented them in a synchronous manner. Therefore each CPU can have one pending request. However, since we have several CPUs per tile, several pending requests per tile are possible. The maximum number of pending requests per atomics unit is therefore the total number of CPUs in the system. These are buffered in the FIFOs of the virtual channel based packet-switched NoC which are served in a round-robin fashion.

Atomic Primitives. To support the three types of synchronization primitives (α, β and γ), we implemented the following set of atomic operations:

(a) efficient spinlock implementation
(b) fetch-and-op operations, with op = {Add, Sub, And, Or}
(c) compare-and-swap primitive
(d) CaCAO: linked queue enqueue/dequeue

(a) The efficient spinlock has an integrated hardware loop until lock acquisition to ensure an $\mathcal{O}(1)$ network utilization. Although acquiring a spinlock is in itself not retry-free and has to be repeated until it is successful, outsourcing the retry attempts into a remote site hardware loop minimizes the retry-penalty. Instead of going back and forth over the NoC, even up into system software - costing several hundreds of clock cycles - the retry penalty of the hardware loop is only a few cycles due to bus arbitration. A back-off retry threshold with accompanying "lock not acquired" response is also implemented.

(b) The fetch-and-op primitives follow the same (1)(2)(3) steps. Between (1) locking the bus and (3) unlocking the bus, the hardware unit performs the (2) step by (2a) reading/fetching data, (2b) executing the {op}-operation in hardware (2c) writing back the modified data and finally (2d) sending the fetched data back to the requesting processor. The fetch-and-op primitive has zero retries since the write-back is non-conditional. Strictly speaking, this primitive can already be classified as CaCAO, even though the critical section in step (2) is not very complex.

(c) The compare-and-swap instruction is similar to (b) with the difference of a conditional write-back, only if the read value is equal to the old value argument of the CAS. This additional comparison is handled in the hardware module, whilst reusing the read and write logic already present for (a) and (b).

The CAS can be in need of retries, since between reading the old value by the CPU (which then issues the CAS) and checking the read value against the old

value inside of the CAS unit. An interfering write accesses can happen, which would lead to a unsuccessful CAS. Therefore, the CPU will have to repeat the procedure until it is eventually successful, leading to increasing network load.

A solution to this problem can be given in hardware, if the operation to be performed on the data can be outsourced to some dedicated logic in hardware. This possibility is function specific, but we show, that it well serves for certain frequently used methods, especially for remote atomic operations. We call these complex and composed atomic operations, as discussed in (d).

(d) CaCAO: Complex and compositional atomic operations. The same (1) (2) (3) steps are followed. However, step (2) basically could be of arbitrary complexity and functionality, even though in this paper we only provide enqueue/dequeue operations into a linked queue. But also the fetch-and-op primitive as discussed in (b) could be classified into this category, since the whole critical section is outsourced into dedicated hardware.

The proposed dedicated hardware module has a compositional nature since the various atomic primitives reuse the same building blocks. The memory read (1) and write (3) step is part of every primitive and there is therefore no waste of resources. E.g. the spinlock reuses the CAS building block with hard-coded old and new values 0 and 1, respectively. These building blocks compose the whole module and a future extension to more functionality can build upon them.

5 Experimental Setup and Results

Our measurements were carried out on our distributed shared memory architecture synthesized onto a FPGA prototype. We used a 2×2 tile design with up to 8 Leon3 cores per tile and a tile local memory, which are connected by a shared bus. The tiles are interconnected with a 2D-Mesh NoC. The timing analysis of our design with the tool Xilinx Vivado revealed, that the proposed atomics unit itself is able to operate at 419 MHz. Together with its TX and RX interface, it still reaches 285 MHz. It is integrated into the network adapter, which is currently able to run at 100 MHz. The complete design with CPUs, NoC, Bus and other modules of our complete project limits the frequency to 50 MHz, since one single clock domain is used so far. Further, due to resource constraints our FPGA prototype limits us to a 2×2 tile design. We tried to compensate this by increasing the core count to 8 cores per tile.

Before running actual stress tests, we first obtained cycle accurate minimal duration simulations using an RTL simulator. The results are shown in Fig. 4(a) for the various atomic primitives implemented in our atomics unit. The whole duration is split into trigger-time (triggering the network adapter by the cores), NoC-time (time for flit generation on sender side, reception on receiver side and travel time over the NoC for both request and response messages) and atomics-time (actual time for carrying out the atomic operation in the atomics unit). It is made clear, that while the trigger-time only depends on the bus arbitration, the NoC-time can increase drastically for higher network load. However, the atomics-time is constant after bus grant.

function	$T_{trigger}$	T_{NoC}	$T_{atomics}$
spinlock	7	47	10
spinlock retry	0	0	10
CAS	14	50	10
Fetch-and-op	11	49	10
hw_enqueue	14	49	37
hw_dequeue	7	48	30

(a) Cycle accurate minimal duration of individual and standalone atomic operations in number of clock cycles

Module	LUTs	Register
Atomics	501	316
TX & RX	1031	687
$\sum$	1532	1003
% NA	11.85%	12.35%
% Tile	1.44%	2.00%

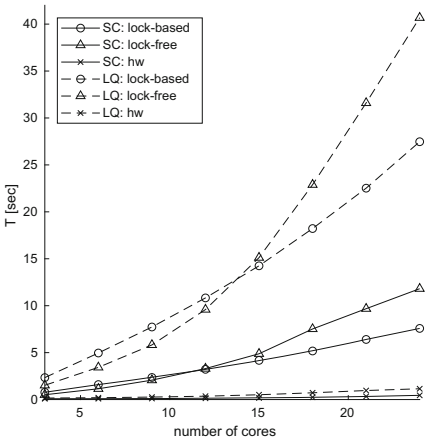
(b) Resource utilization of the atomics unit

Fig. 4. Minimal duration simulations and synthesis results

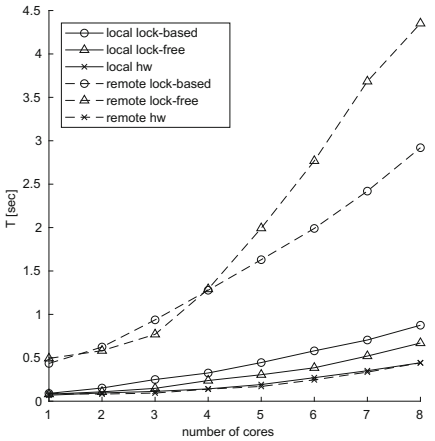
Further, the synthesis of our module has the resource usage given in Fig. 4(b). It is part of a network adapter that additionally has load/store, DMA as well as task spawning support. The overall resource utilization of the atomics unit with around 12% of the network adapter and only maximally 2% of the whole tile, is comparatively low.

Besides these minimal duration simulations, we investigated several stress test measurements on our FPGA platform using the scenarios described earlier. In all the following micro benchmarks, each used core performs 10k iterations of the given scenarios, i.e. either 10k increments to the shared counter (SC) or 10k enqueue/dequeue operations to the linked queue (LQ). The tests are always done for all three synchronization types (lock-based, lock-free or dedicated hardware). We want to note, that for x cores, the overall workload is x-times as high. Alternatively, if the overall workload was kept constant with increasing core counts, the resulting graphs - from a purely visual perspective - would not be as easily distinguishable as shown in Fig. 5.

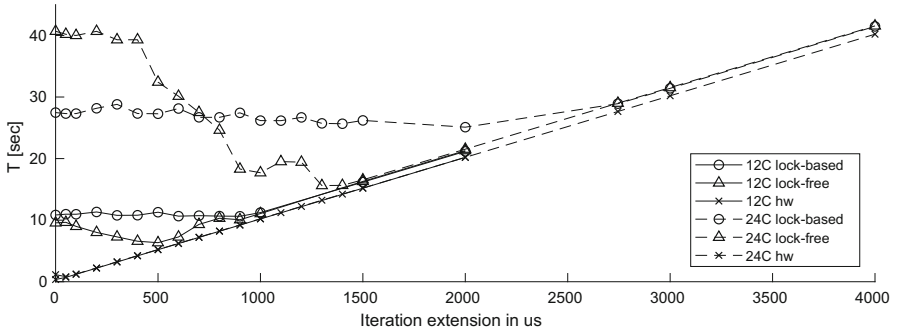
The results for the first stress test are depicted in Fig. 5(a) and show the execution time for both the shared counter (SC, solid lines) and the linked-queue update (LQ, dashed lines) for all three types of synchronization classes each. In this scenario, we investigate remote accesses to one tile from three other tiles with 1 to 8 cores each, totaling to up to 24 cores. Due to the higher complexity and therefore longer iteration duration of the linked-queue scenario, the dashed lines are always above the corresponding solid lines. Apart from that, the two scenarios behave similar. We make four key observations: (1) For no and low concurrency on the data structure, the lock-free variant is preferable over the lock-based one, since it does not suffer from (many) retries and the corresponding re-execution of the critical section. (2) Although not shown in the graphs, but underlined by our measurements, the retry rate rises with increasing concurrency, i.e. core count. We further did not depict but measured, that the execution time of the lock-based variants is linear in the number of cores, while the execution time for the lock-free variants is linear in the number of retries. (3) There is a concurrency depended cross-over point between the lock-based and lock-free variants (intersection of the lines). A concurrency depended decision



(a) Execution time for the three synchronization variants (lock-based, lock-free, hw) for the shared counter (SC) and the linked queue (LQ) for different core counts



(b) Comparison of purely local vs. purely remote execution of the linked queue (LQ) scenario for variable core counts per tile



(c) Variable iteration duration for fixed critical section size for the three variants for a 12 and 24 core scenario, respectively

Fig. 5. Stress test measurement results

for one or the other could be investigated and made at design and/or run-time as possible performance optimization. (4) In all cases, the dedicated hardware implementation fetch-and-inc/CaCAO outperforms the other two variants by far. For the list-queue, the speedup rises from 9.5 (3×1 core) to 35.4 (3×8 cores) and from 14.6 to 23.9 compared to the lock-free and lock-based variant, respectively. The dedicated hardware implementation almost does not suffer from rising concurrency. The additional time is due to serialized execution in the atomics unit.

In the second stress test, we compare the execution time of LQ for purely local vs. purely remote access to the shared data structure. The results are depicted in Fig. 5(b) for varying core count between 1 and 8. We make 3 further

key observations: (5) As expected, the purely local execution outperforms the remote operation in general. (6) Whereas for remote operation, there again is a cross-over point between the lock-based and lock-free variants (intersection of dashed lines), for local operations this behavior is not observed. The concurrency in combination with the much lower retry penalty explains this. (7) The relative advantage of the dedicated hardware implementation is much higher for remote than for local operations due to the higher retry penalty of the lock-free and the higher iteration duration of the lock-based variant. The advantage of the dedicated hardware over the lock-free variant is 6.5 times higher for remote compared to local operations. The advantage over the lock-based variant is 3.3 times higher. In both cases, we considered 8 local vs. 8 remote cores.

In our final measurements, we mimic different ratios of the non-critical part to the critical section of an application. This is done by keeping the critical section size constant, whereas we extend the whole base iteration by some iteration extension (iteration = base iteration + iteration extension). In Fig. 5(c), the results are depicted for the three variants of the linked-queue scenario for 12 and 24 cores. For an extension of 0 µs, the critical section in our scenarios is e.g. around 5% of the unextended base iteration for the SC in the 24 core variant. With this said, we make further key observations: (8) The lower the percentage of the critical section compared to the whole iteration, the lower the retries for the lock-free version and the corresponding total time. (9) A minimum can be found at a delay of around 1400 µs for the 24 core variant and at 500 µs for the 12 core variant (these times equal the average base iteration times for the lock-free variant). The retry rate drops to almost zero at these points. From then on, the execution time is dominated by the iteration extension, i.e. the additional time of the non-critical section. Similarly the lock-based variants start to be dominated by this extension after their average unextended base iteration times are reached, which are 800 µs and 2700 µs, respectively. (10) If the iteration extension dominates the whole iteration, i.e. if the percentage of the critical section gets very small, all variants converge. Even the dedicated hardware variants are dominated by the non-critical part. (11) At 500 µs, were the 12 core variant reaches the zero retry point, shows that the higher concurrency of 24 cores still has a high number of retries. An extrapolation of this principle would yield similar behavior for more than 24 cores at the 1400 µs mark, etc.

6 Conclusion and Future Work

To tackle the scalability issue of future manycore platforms, efficient remote operations and synchronization primitives are a key.

Our investigated scenarios show that there are application specific usecases for lock-based, as well as lock-free synchronization. Modern distributed shared memory platforms should therefore provide both types of synchronization (efficient inter-tile lock implementation, as well as general purpose atomic primitives like CAS or LL/SC) to allow flexibility for the programmer.

We further showed, that especially remote operation highly profit from dedicated hardware implementations to overcome the disadvantages of both lock-based and lock-free software implementations. Future systems should implement often used and highly concurrent tasks as dedicated hardware. Near memory acceleration could be a further improvement.

As future work, for systems without CaCAO support, one could explore the potential of situation based usage of the lock-based or lock-free variants, which would require a concurrency dependent decision. A heuristic, with which the runtime system could choose between the lock-based and lock-free variant would be needed.

Further, we plan to extend the CaCAO approach to more complex functionalities based on its compositional nature. We want to identify useful functions and investigate their potential in real applications.

Acknowledgement. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center Invasive Computing [SFB/TR 89]. The authors would also like to thank Christoph Erhardt, Sebastian Maier and Florian Schmaus from FAU Erlangen, as well as Dirk Gabriel from our chair for the helpful discussions.

References

1. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A., Hennessy, J., Horowitz, M., Lam, M.S.: The stanford dash multiprocessor. Computer **25**(3), 63–79 (1992)
2. Mellanox: Ug130-archoverview-tile-gx. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf
3. Michael, M.M., Scott, M.L.: Implementation of atomic primitives on distributed shared memory multiprocessors. In: 1995 Proceedings of First IEEE Symposium on High-Performance Computer Architecture, pp. 222–231. IEEE (1995)
4. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In: Proceedings of the 3rd International Workshop on Software and Performance, pp. 55–67. ACM (2002)
5. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
6. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **15**(5), 745–770 (1993)
7. Wei, Z., Liu, P., Sun, R., Ying, R.: High-efficient queue-based spin locks for Network-on-Chip processors. In: 2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pp. 260–263. IEEE (2014)
8. Wei, Z., Liu, P., Zeng, Z., Xu, J., Ying, R.: Instruction-based high-efficient synchronization in a many-core Network-on-Chip processor. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2193–2196. IEEE (2014)
9. Chen, X., Lu, Z., Jantsch, A., Chen, S.: Handling shared variable synchronization in multi-core Network-on-Chips with distributed memory. In: 2010 IEEE International on SOC Conference (SOCC), pp. 467–472. IEEE (2010)
10. Schweizer, H., Besta, M., Hoefler, T.: Evaluating the cost of atomic operations on modern architectures. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp. 445–456. IEEE (2015)

11. Mellanox: Ug101-user-architecture-reference.pdf. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG101-User-Architecture-Reference.pdf
12. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. (TOCS) **9**(1), 21–65 (1991)
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington (2011)
14. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. J. Parallel Distrib. Comput. **51**(1), 1–26 (1998)
15. Tian, G., Hammami, O.: Performance measurements of synchronization mechanisms on 16PE NoC based multi-core with dedicated synchronization and data NoC. In: 16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2009, pp. 988–991. IEEE (2009)