LRSCwait: Enabling Scalable and Efficient Synchronization in Manycore Systems through Polling-Free and Retry-Free Operation

Samuel Riedel* Marc Gantenbein* Alessandro Ottaviano* Torsten Hoefler† Luca Benini*‡
*IIS, ETH Zürich †SPCL, ETH Zürich ‡DEI, University of Bologna
*{sriedel,ganmarc,aottaviano,lbenini}@ethz.ch †htor@inf.ethz.ch

***Abstract*—Extensive polling in shared-memory manycore systems can lead to contention, decreased throughput, and poor energy efficiency. Both lock implementations and the general-purpose atomic operation, load-reserved/store-conditional (LRSC), cause polling due to serialization and retries. To alleviate this overhead, we propose LRwait and SCwait, a synchronization pair that eliminates polling by allowing contending cores to sleep while waiting for previous cores to finish their atomic access. As a scalable implementation of LRwait, we present Colibri, a distributed and scalable approach to managing LRwait reservations. Through extensive benchmarking on an open-source RISC-V platform with 256 cores, we demonstrate that Colibri outperforms current synchronization approaches for various concurrent algorithms with high and low contention regarding throughput, fairness, and energy efficiency. With an area overhead of only 6%, Colibri outperforms LRSC-based implementations by a factor of 6.5× in terms of throughput and 7.1× in terms of energy efficiency.**

***Index Terms*—atomics, synchronization, manycore, RISC-V**

I. Introduction

Manycore systems are becoming increasingly popular due to the growing demand for computing power. However, the parallel execution of tasks introduces synchronization and atomicity issues that can lead to race conditions and unpredictable results. To ensure exclusive access to critical sections (CSs), atomic operations and locks can be used. However, locks also block cores that try to acquire them when they are not free, leading to busy waiting and polling. Polling, or constantly checking a shared resource for changes, can become an issue in concurrent algorithms. It leads to high core utilization and reduces overall system performance and energy efficiency as the cores compete for shared resources [1]. In the worst case, it can lead to livelocks or starvation, where cores are blocked from making progress because others continuously block them.

Non-blocking algorithms avoid locks by updating atomic variables directly with atomic read–modify–write (RMW) operations. Specific arithmetic operations, like *add, and, or*, are often supported through specialized instructions. However, most concurrent algorithms require more complex modifications of atomic variables, such as conditional updates. For generic RMW operations, the compare-and-swap (CAS) operations or load-reserved/store-conditional (LRSC) pair are typical primitives designed to ensure that the operation is *atomic*, i.e., without interference from other cores [2]. For example, RISC-V's load-reserved (LR) instruction loads a value from memory and places a reservation. The core can perform operations with the loaded value and store the result back conditionally with a store-conditional (SC). The latter instruction will only succeed if the reservation is still valid, meaning the memory location was not modified in the meantime. If the SC succeeds, the RMW sequence appears atomically. However, cores that fail an SC must retry the LRSC sequence pair until it succeeds. Variables outside CSs can also cause polling, where cores wait for changes in shared variables, leading to inefficiencies in core communication, like producer/consumer interactions.

To eliminate retries and polling, we propose a novel, general-purpose atomic RMW instruction pair called LRwait and SCwait. They extend the standard RISC-V LRSC pair by moving the linearization point, the point where the atomic operations of different cores get ordered, from the SC to the LRwait. The LRwait and SCwait are used in the same way as the LRSC pair. However, instead of returning the memory value immediately, the LRwait instruction only responds to one core at a time to set it up for a successful SCwait. This prevents failing SCs and retry loops. Furthermore, LRSCwait allows implementing polling-free locks. To eliminate polling even for non-atomic variables, we propose the Mwait instruction, which enables cores to sleep until a specific memory address changes its value.

While cache-based systems often rely on the coherency protocol to implement such behavior, manycore accelerators scaling to hundreds of cores often rely on software-managed, multi-banked scratchpad memories (SPMs). Examples include commercial chips like GAP9 [3] and RC64 [4], as well as large-scale research prototypes like MemPool [5]. While LRSCwait can be applied to cache and cache-less systems, in this work, we focus on cache-less, SPM-based manycore systems since they pose the design challenge of the memory controllers having to keep track of outstanding LRwait instructions to send their responses at the right time. However, duplicating large hardware queues for each bank is costly and scales poorly.

As a scalable implementation of the proposed instructions, we present *Colibri*. Its concept is similar to linked-list-based software queues. It does not allocate a full array of entries for each queue but just a head and tail pointer per queue as illustrated in Fig. 1. Each core is equipped with a queue node that can be linked to any queue. For Colibri, this means that instead of equipping each memory controller with a

979-8-3503-4859-0/DATE24/© 2024 EDAA
Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.

hardware queue that can hold an entry for each core, each memory controller is extended with a parameterizable number of head and tail registers to form linked lists. Each core is then equipped with one hardware queue node, and when issuing an LRwait, the core inserts itself in the corresponding queue. We implemented Colibri on the open-source, manycore MemPool system, consisting of 256 cores sharing 1 MiB of L1 memory [5]. Colibri provides a scalable solution that can be easily integrated into existing RISC-V systems. The LRSCwait solution can be used as a drop-in replacement for LRSC or as a powerful extension, making it a desirable option for high-performance computing systems. We evaluate the performance of Colibri against various hardware and software approaches. The results indicate that Colibri outperforms other approaches in all experiments, with a throughput increase of up to 6.5 times in high-contention situations and a 13% increase in low-contention scenarios. Additionally, Colibri reduces polling, allowing other applications to be unaffected by concurrent atomic accesses. Our key contributions are the following:

- The LRwait extension consisting of three novel instructions (LRwait, SCwait, and Mwait), which enable atomic access and monitoring memory locations with a minimal amount of polling (Section III).
- A scalable implementation for LRwait named Colibri leveraging a distributed reservation queue (Section IV).
- An implementation and evaluation of Colibri on the MemPool platform that outperforms other approaches in throughput, fairness, polling, and energy per atomic access. Colibri scales linearly on the MemPool platform by introducing an area overhead of just 6% while being 8.8x more energy efficient than locks (Section V).

II. Related Work

A common approach to mitigate polling is using a backoff after a failed atomic access [2]. Existing backoff schemes, such as exponential backoff, where each failed attempt increases the backoff time, can reduce the overhead on shared resources but still make the cores busy-waiting and performing sub-optimally.

The Mellor-Crummey, Scott (MCS) lock [6] relies on a software queue for contending cores to enqueue in and spin on their respective node in the queue. This guarantees that each core spins on a unique location to mitigate contention on the lock variable itself. This approach works well in cache-based systems since each core can spin on its own L0 cache. However, in this work, we focus on systems with software-managed memories.

While software approaches to locks are general and platform agnostic, their performance can not keep up with hardware locks. A study of two software locks and four hardware locks shows that hardware locks consistently outperform the software approaches by 25%-94%. However, hardware locks such as Hardlocks [7] do not scale well, as the locks are managed by a centralized locking unit accessible to all cores. Accessing this unit quickly becomes the bottleneck in large systems. Furthermore, the number of locks is fixed at implementation time. Similarly, Glaser et al. present a synchronization unit where each core has a private direct connection to each hardware lock [8]. While this solves the contention issue, it prevents scaling beyond a few tens of cores. GLock suffers from a similar scalability issue [9]. It is based on a dedicated on-chip network consisting of lock managers and local controllers that synchronize to acquire a lock. Monchiero et al. propose a synchronization-operation buffer implemented as a hardware queue in the memory controller to resolve the lock accesses [10]. However, this approach only implements locks and has a hardware cost that is proportional to the number of cores. Furthermore, each memory controller would require such a buffer to manage locks.

While locks are a common solution for protecting critical sections, their blocking nature often limits performance. Lock-free algorithms, on the other hand, allow for much more concurrency. They often rely on instructions like CAS or the LRSC pair. This section focuses on the latter, specifically, RISC-V's implementation. For example, the ATUN is a unit that can be placed in an Advanced eXtensible Interface (AXI) bus to support LRSC instructions to the downstream memory [11]. The table allows a reservation for every core, thus implementing a non-blocking version of LRSC. Furthermore, each bank would require its own ATUN adapter in a multi-banked system, introducing significant hardware overhead in large manycore systems. The Rocket chip features a similar implementation [12]. However, the number of reservations is limited.

MemPool implements a lightweight version of LRSC by only providing a single reservation slot per memory bank [5]. However, this sacrifices the non-blocking property of the LRSC pair. The GRVI multiprocessor, on the other hand, modifies the granularity at which LRSCs operate by locking the complete memory bank [13]. This reduces the hardware overhead to one bit per core per bank, albeit the approach is still affected by retries due to spuriously failing SC operations.

All those solutions implement the standard RISC-V LRSC instruction, leveraging the freedom of the official specification to achieve different trade-offs. However, none of them solve the polling and retry issue of failing SC operations. On the contrary, they sometimes worsen it. The Request-Store-Forward (RSF) synchronization model proposed by Liu et al. is similar to LRwait [14]. Synchronization requests are stored in a hardware-managed memory and handled in order by a synchronization controller. However, this approach leads to a high memory footprint, and the hardware needs to be replicated for each memory bank. Furthermore, it is infeasible for software-managed memories as the synchronization controller will interfere with the allocated data when adding the queue to the memory.

Our LRwait approach and the efficient implementation through Colibri scale well to hundreds of cores and banks while completely eliminating polling without sacrificing granularity.

III. LRwait and SCwait

RISC-V defines the load-reserved/store-conditional (LRSC) instructions to implement generic, atomic RMW operations. The LR instruction reads a value from memory and places a reservation, which remains valid until the specified memory address is changed. The core can then modify the value and write the result back with an SC instruction. The latter will succeed only if the reservation is still valid. If the SC fails,

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.

the LRSC sequence has to be retried. The linearization point between contending cores is thus at the SC.

LRwait eliminates the wasteful retry loop by moving the linearization point to the LRwait instruction, i.e., atomic accesses of competing cores are ordered at the LRwait instruction. Instead of immediately returning the value, the memory controller withholds the response such that only one core gets a response at a time, guaranteeing it to be the only core issuing an SCwait to the same address. The LRSCwait and LRSC instructions share similar semantics. The SCwait stores a value conditionally and returns a success or failure code analogous to the SC. Likewise, the LRwait matches the LR instruction, but its response is delayed. The sequence of an atomic RMW operation with LRSCwait is the following:

1) The core issues the LRwait and waits for the response.
2) The memory buffers the request until it is the next outstanding LRSCwait pair to that address.
3) Once the LRwait is the next in line, the memory serves the request with the current memory value and monitors it. A store to the same address clears the reservation.
4) The core modifies the value and writes it with an SCwait.
5) The memory accepts the value if a valid reservation still exists and issues the response.

While the memory guarantees that only one core proceeds with an LRSCwait pair, it cannot eliminate the possibility of another core overwriting the atomic variable, leading to a failing SCwait. One constraint of the LRSCwait instruction pair is that every LRwait must eventually be followed by an SCwait. While RISC-V does not have this constraint for LRSC, our extension requires the matching SCwait to yield the queue of outstanding LRwait instructions and allow progress on the atomic variable. Albeit LRSCwait can be used as a drop-in replacement for LRSC, it removes the lock-free progress guarantee that the LRSC instructions have. Since only one core can issue an SCwait, a malicious core could block the resource indefinitely and obstruct progress. However, LRSCwait still gives strong progress guarantees under the following constraints:

a) Mutual exclusion: Just as the LRSC pair, the SCwait only succeeds if a valid reservation is present, meaning there was no write between the LRwait and the SCwait, which guarantees mutual exclusion and, therefore, atomicity.

b) Deadlock freedom: To prevent circular dependencies between cores, every core must have at most one outstanding LRwait operation. RISC-V does not impose this requirement on LRSC. However, only the innermost LRSC pair is guaranteed to progress. Therefore, this requirement for deadlock freedom is a requirement for livelock freedom already. Furthermore, each core's LRwait must eventually be followed by an SCwait to close the CS. We impose the same constraints as the RISC-V standard to allow only a finite and limited set of instructions between LRwait and SCwait.

c) Starvation freedom: Starvation freedom guarantees that all cores eventually make progress. LRSC only guarantees that one core makes progress because an unlucky core could always lose the SC to a faster core. In our work, this scenario is prevented by handling the LRSCwait pairs in order, thus enabling all cores to eventually execute the LRSCwait pair and, therefore, guaranteeing starvation freedom.

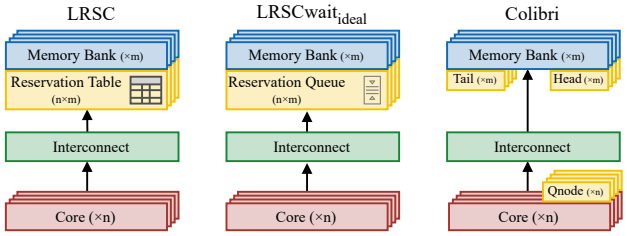

Fig. 1. Difference between LRSC architecture with a reservation table, LRSCwait with a reservation queue, and Colibri with a linked-list-like structure.

Overall, while the blocking nature of the LRSCwait makes a core's progress depend on other cores correctly executing and leaving the LRSCwait blocks, these constraints can easily be adhered to in bare-metal systems, which are fully under the programmer's control. LRSCwait can provide very strong progress guarantees, enabling each core to progress. However, hardware failure or software bugs can become blocking.

A. Ideal Hardware Implementation

A straightforward hardware implementation of LRSCwait requires tracking all outstanding reservations in order to ensure fairness and starvation freedom. As shown in Fig. 1, this can be achieved by an LRSCwait adapter placed in front of each memory bank, consisting of (i) a queue-like data structure of capacity n, where n is the number of cores in the system, and (ii) some additional logic to monitor memory accesses and invalidate reservations when the target address is overwritten. The overhead of this implementation in a system with m memory banks is $O(n \log_2(n) m)$, where $\log_2(n)$ represents identifier size per core. Assuming that m scales linearly with the number of cores, this implementation's overhead scales quadratically with the system size: $O(n^2)$, a non-negligible hardware complexity.

B. Optimized Hardware Implementation

To reduce the hardware complexity, we can decrease the queue's capacity by assuming that only a subset of cores can access a specific address simultaneously. Our implementation supports a parametrizable number of reservation slots q. The case with $q = n$ falls back to the ideal LRSCwait pair described in Section III-A. We call this implementation *LRSCwait$_{ideal}$*. If $q < n$, we trade hardware overhead with performance. In these implementations, *LRSCwait$_q$*, cores executing an LRwait to a full queue will fail immediately.

C. Mwait

To allow efficient monitoring and notification of a memory location from a core in the system, we introduce *Mwait*. Mwait is derived from LRwait, but without a matching SCwait. Instead, the reservation placed by Mwait is used to identify the core that needs to be notified of a change. For instance, a core may monitor a queue and be woken up when an element is pushed onto the queue. Our experiments show that Mwait provides a simple and efficient mechanism for monitoring memory

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.

locations, allowing cores to be woken up only when necessary. To handle the possibility that the change we wish to observe has already occurred, we provide Mwait with an expected value. If the memory location already differs from the expected value when Mwait is served, the core is immediately notified.

IV. Colibri

Colibri implements a distributed queue, similar to a linked list, shown in Fig. 1. It alleviates the huge hardware overhead of the hardware queues at each memory controller, replacing it with a dedicated head and tail node per queue and a simple controller. On top of that, each core requires its own hardware node, called *queue node (Qnode)*, to enqueue itself. Since each core can only be in one queue, one Qnode per core is enough. Therefore, Colibri only requires $O(n + 2m)$ nodes and scales linearly with the system size.

Since the queue is distributed across Qnodes and the head/tail nodes next to the memory banks, updating the queue becomes more complex. In comparison to the ideal LRwait, an enqueue operation from an LRwait, or a dequeue operation by an SCwait, does not happen in one place and a single cycle.

We present a simple example of the construction and deconstruction of the queue in Fig. 2 with a single memory and two cores contending for the same address. Both cores have their own Qnodes, and the memory has a head and tail node. We call the cores *A* and *B* for simplicity.

a) LRwait: Core A issues an LRwait request to the memory (1). Since the queue is initially empty, the head and tail nodes are set to A, and a reservation to the specified location is set up. The memory then sends the value A (2). During or after the described events, B's LRwait request arrives at the memory (3). When the B's LRwait request arrives at the memory, the controller appends B at the tail of the queue and then adds it as the successor to A. This is done by sending a so-called *SuccessorUpdate* to A (4). This SuccessorUpdate writes to A's Qnode to make it point to B. In this final state shown in the top half of Fig. 2, A and B form a queue with A at the head of the queue. At this point, A can issue an SCwait while B is sleeping, waiting for a response.

b) SCwait: Core A finishes its LRSCwait pair by issuing an SCwait with the modified value (5). Immediately after an SCwait passes the Qnode, it sends a *WakeUpRequest* to the memory containing its successor, i.e., B (6). On arrival of the SCwait request at the memory, the head node and reservation

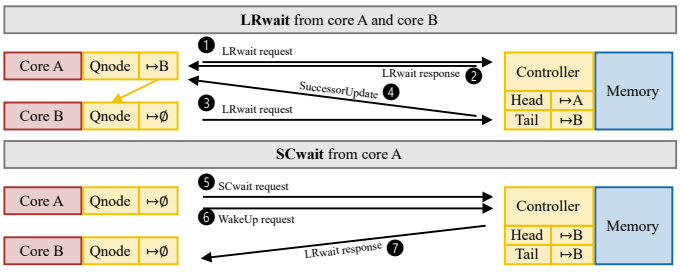

Fig. 2. LRwait and SCwait sequence in Colibri with two cores and one queue.

are checked. If everything is valid, the head node is temporarily invalidated to prevent a future SCwait from the same core from succeeding without reservation, and the SCwait is written to memory. The WakeUpRequest sets the head node to the successor node and triggers an LRwait response with the latest memory value written by A, i.e., for B (7). Core B is now free to issue an SCwait. Finally, the head and tail nodes point to B since B is the only core in the queue.

This sequence can be generalized to more cores. Qnodes accept SuccessorUpdates even when the core is asleep, allowing the queue to be enlarged independent of the cores' state.

A. Correctness of Colibri

1) LRwait: When an LRwait enqueues a node, it must update the tail to point to the newly enqueued node and append it to the previous tail node if it existed. If not, the enqueue operation inherently becomes atomic. Otherwise, to update the predecessor, the memory controller sends a SuccessorUpdate to the previous tail and overwrites the tail node atomically. Since we can only have one LRwait per core and SuccessorUpdates are only sent when overwriting a tail node, only a single SuccessorUpdate will ever be in flight to a Qnode, guaranteeing no lost links in the queue. If the SuccessorUpdate arrives after the core issued an SCwait, it will immediately bounce back as a WakeUpRequest. If the next LRwait arrives while the SuccessorUpdate is still in flight, the tail will be updated again, and the SuccessorUpdate will be sent to the next core. While a glance at the Qnodes might reveal broken links momentarily, the links only have to be made when a core issues its SCwait, which requires an LRwait response from the memory controller since memory transactions are ordered, this will always happen after the SuccessorUpdate.

2) SCwait: If a core issuing an SCwait is the only one in the queue, i.e., the head and tail are equal, dequeuing itself by clearing the head and tail is trivial. Otherwise, the SCwait will invalidate the head node while leaving the value unchanged. A core would need to overwrite the head node to reach an inconsistent queue from this stage. This is only allowed for an LRwait reaching an empty queue or a WakeUpRequest arriving at the memory after invalidating the head node by an SCwait. A WakeUpRequest can only be triggered by an SCwait passing the Qnode, which can only be sent by a core at the head of the queue since the other cores are still waiting for their LRwait response. Thus, the WakeUpRequest arriving at the memory node guarantees that the queue is in a consistent state again.

B. Extending Colibri with Mwait

A core can issue an Mwait request to enqueue into Colibri's queue to monitor a memory location. The memory controller then waits for a write to the monitored location, just like for LRwait's reservation. After a write, the memory controller triggers a response to the Mwait instruction. For Mwait, the head node is sleeping as well in contrast to LRSCwait where the head is free to issue an SCwait. The Mwait response makes the Qnode dispatch the WakeUpReq for its successor, which then bounces to the memory controller, where the next Mwait response is released. In contrast to LRSCwait, the whole reservation queue is woken up without any interference from the cores.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.

TABLE I
AREA OF A MEMPOOL_TILE WITH DIFFERENT LRSCWAIT DESIGNS.

Architecture	Parameters	Area[kGE]	Area[%]
MemPool tile	none	691	100.0
with LRSCwait$_1$	1 queue slot	790	116.4
with LRSCwait$_8$	8 queue slots	865	127.4
with Colibri with MWait	1 address	732	105.9
with Colibri with MWait	2 addresses	750	108.5
with Colibri with MWait	4 addresses	761	110.1
with Colibri with MWait	8 addresses	802	116.3

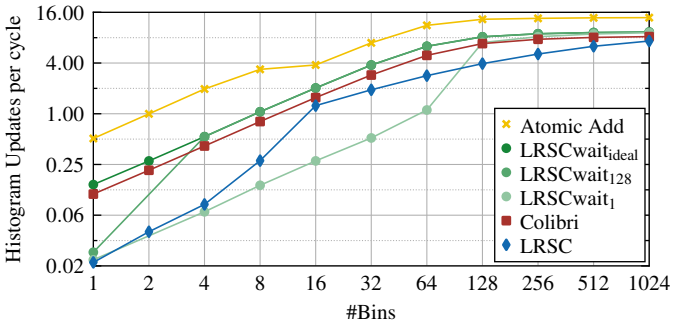

Fig. 3. Throughput of different LRSCwait implementations and standard RISC-V atomics at varying contention.

V. RESULTS

We implement and evaluate various LRSCwait variations and Colibri in MemPool, an open-source, 256-core RISC-V system with 1024 SPM banks [5]. All our results are taken from cycle-accurate register-transfer level (RTL) simulation. Physical implementation results come from implementing MemPool in GlobalFoundries' 22FDX fully depleted silicon-on-insulator (FD-SOI) technology. Power consumption is evaluated in typical conditions (TT/0.80 V/25 °C), with switching activities from a post-layout gate-level simulation running at 600 MHz.

The area overhead of different implementations is shown in Table I. Even optimized implementations of LRSCwait quickly grow in size, while LRSCwait$_{ideal}$ is physically infeasible for a system of MemPool's scale. Colibri, on the other hand, grows linearly and allows up to eight queues per memory controller with a similar area overhead to LRSCwait$_1$ of 16%.

A. Benchmarking

a) Histogram: We implement a concurrent histogram benchmark to evaluate Colibri's performance at different levels of contentions. The application atomically increments a parametrizable number of bins. The fewer bins, the higher the contention. We increment a bin with different atomic operations and compare their performance as updates per clock cycle.

The throughput of different LRSCwait implementations is shown in Fig. 3. LRSCwait$_{ideal}$ outperforms all implementations across the whole spectrum of contention. The optimized implementations show similar performance at low contention but achieve much lower performance when the contention is higher than their number of reservations. Finally, Colibri achieves near-ideal performance across all contentions. The slight performance

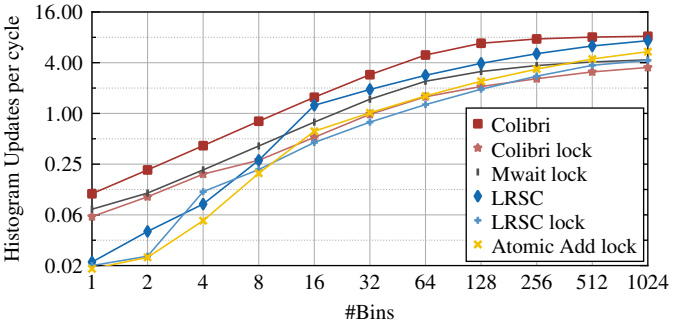

Fig. 4. Throughput of different lock implementations compared to generic RMW atomics at varying contention.

penalty comes from the extra roundtrips of Colibri's node update messages. Colibri outperforms the LRSC-based implementation by a factor of 6.5× at high contention and 13% at low contention. For completeness, we also show the throughput of an *Atomic Add* implementation, which is designed specifically to increment a memory location atomically and represents the plot's roofline. However, most concurrent algorithms need more complex atomic RMW operations than an increment, where programmers have to resort to locks of generic RMW atomics like LRSCwait.

Fig. 4 compares Colibri to various lock-based implementations. Colibri, LRSC, and Atomic Add locks are spin locks with a backoff of 128 cycles, while Mwait lock implements an MCS lock, where Mwait is used to avoid polling. Colibri outperforms all other approaches for any contention. We observe that the LRSC and AMO-lock approaches perform worst at high contention due to their heavy polling and retry traffic, while waiting-based approaches perform average. At low contention, the waiting-based approaches perform worst because of their management overhead, while the other atomics tend to Colibri.

b) Interference: We showed that LRSCwait can significantly improve the throughput of atomic operations across all levels of contention. On top of this increase in performance, eliminating the need to retry failed operations and polling also reduces traffic and frees up resources for cores not executing atomics. Cores working on computation experience less negative interference from the constant polling of atomics. To measure this effect, we partitioned the 256 cores of MemPool to either work on a matrix multiplication or to execute atomic operations. We measure the execution time of the matrix multiplication compared to an execution time without any interference. Figure 5 shows the relative performance for various types of atomic operations and distributions of working cores. Our Colibri implementation has a negligible impact on the worker cores, even at high contention and with a poller-to-worker ratio of 252:4. The retries of the LRSC operations, on the other hand, significantly impact the workers' performance, despite a backoff of 128 cycles. At the same ratio of poller-to-workers, the LRSC implementation slows the workers down to 26%.

c) Queue: To evaluate Colibri on a commonly used concurrent algorithm, we implement an MCS queue with LRSC and LRSCwait, as well as a lock-based queue using atomic adds. Concurrent queues are widely used for task scheduling

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.

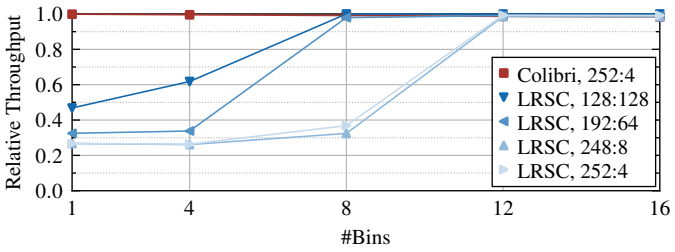

Fig. 5. Matrix multiplication performance with interference from atomics. The poller-to-worker ratio is annotated in the figure with poller:worker.

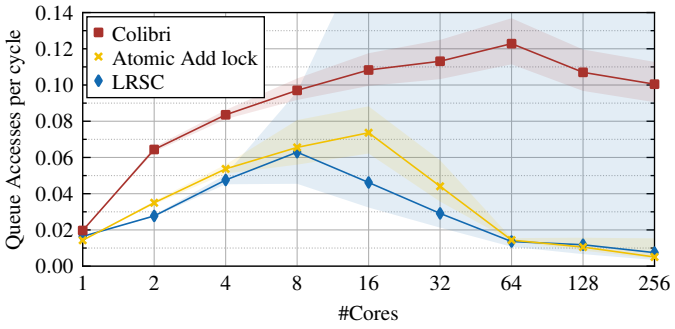

Fig. 6. Queue operations throughput with different atomics.

or producer/consumer pipelines. Figure 6 shows the number of queue operations for a range of cores accessing a single queue. Colibri performs best and can sustain a high performance even at 256 cores. It outperforms the LRSC and lock-based approaches by 1.54× and 1.48× times with eight cores before both implementations drop in performance due to excessive retries and polling. At 64 cores, Colibri is 9× faster. The shaded areas show each implementation's slowest and fastest core performance range. It illustrates how Colibri results in a very balanced and fair workload distribution, while LRSC can have very big variations.

d) Energy efficiency: Table II shows the energy per operation for atomic accesses to the histogram at the highest contention. Comparing Colibri to the Atomic Add, which represents an ideal atomic update, we can see how energy-efficient Colibri is for a generic RMW operation that consists of an LRwait, add, and SCwait operation. Compared to the LRSC or lock-based implementation, we observe the large benefit of the reduction in polling and retry traffic for improving energy efficiency by a factor of 7.1× and 8.8×.

VI. Conclusion

In this work, we propose the LRwait and Mwait synchronization primitives and their implementation, Colibri, which demonstrate a novel and effective solution for the LRSC synchronization problem in cache-less manycore systems. Colibri offers superior performance and scalability compared to existing hardware and software approaches, reduces polling, and improves throughput in a fair manner. Our experiments show that Colibri outperforms other implementations in both high and low contention scenarios by up to 6.5× and improved energy efficiency by up to 8.8×. The polling and retries of LRSC-based solutions can lead to performance degradation of unrelated workers by up to 4×, while Colibri can operate even at high contention without impacting other cores. Additionally, Colibri can be easily integrated into existing RISC-V systems with a small hardware overhead and can be used as a drop-in replacement for LRSC or as an extension.

TABLE II
Area results for a `mempool_tile` for ideal LRWait.

Atomic access	Backoff	Power (mW)	Energy (pJ/OP)	Δ
Atomic Add	0	175	29	−77%
Colibri	0	169	124	±0%
LRSC	128	186	884	+613%
Atomic Add lock	128	188	1092	+780%

Acknowledgment

This work is funded in part by the COREnext project supported by the EU Horizon Europe research and innovation programme under grant agreement No. 101092598.

References

[1] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, 1990.
[2] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*, 2nd ed., S. R. Merken, Ed. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., 2020.
[3] GreenWaves Technologies SAS, "GAP9 next generation processor for hearables and smart sensors," GreenWaves Technologies SAS, Tech. Rep., 2021. [Online]. Available: https://greenwaves-technologies.com/wp-content/uploads/2022/06/Product-Brief-GAP9-Sensors-General-V1_14.pdf
[4] R. Ginosar, P. Aviely, T. Israeli, and H. Meirov, "RC64: High performance rad-hard manycore," in *IEEE Aerosp. Conf. Proc.* IEEE, Jun. 2016, pp. 2074–2082.
[5] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "MemPool: A scalable manycore architecture with a low-latency shared L1 memory," *IEEE Trans. Comput.*, vol. 72, no. 12, pp. 3561–3575, 2023.
[6] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.
[7] T. B. Strøm, J. Sparsø, and M. Schoeberl, "Hardlock: Real-time multicore locking," *J. Syst. Archit.*, vol. 97, pp. 467–476, 2019.
[8] F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang, and L. Benini, "Energy-efficient hardware-accelerated synchronization for shared-L1-memory multiprocessor clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 633–648, Mar. 2021.
[9] J. L. Abellán, J. Fernández, and M. E. Acacio, "Design of an efficient communication infrastructure for highly contended locks in many-core cmps," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 972–985, 2013.
[10] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "An efficient synchronization technique for multiprocessor systems on-chip," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 1, pp. 33–40, Mar. 2006.
[11] A. Kurth, S. Riedel, F. Zaruba, T. Hoefler, and L. Benini, "ATUNs: Modular and scalable support for atomic operations in a shared memory multiprocessor," in *ACM/IEEE Des. Autom. Conf.*, vol. 57. San Francisco, CA, USA: IEEE, Jul. 2020, pp. 902–907.
[12] K. Asanović *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[13] J. Gray, "Implementation of LR/SC on the GRVI multiprocessor," 2016. [Online]. Available: https://groups.google.com/a/groups.riscv.org/g/hw-dev/c/Mt9Q94f_l2w?pli=1
[14] S. Liu and J. L. Gaudiot, "Synchronization mechanisms on modern multi-core architectures," in *Proc. 12th Asia-Pacific Conf. Adv. Comput. Syst. Archit.* Seoul, Korea: Springer Verlag, 2007, pp. 290–303.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 24,2025 at 10:41:49 UTC from IEEE Xplore. Restrictions apply.