NUMA-Aware Non-Blocking Calendar Queue

Maryan Rab*, Romolo Marotta†, Mauro Ianni†, Alessandro Pellegrini†, Francesco Quaglia*†
* *University of Rome "Tor Vergata", Italy*
Email: maryan.rab@gmail.com, francesco.quaglia@uniroma2.it
† *Lockless S.r.l., Rome, Italy*
Email: {marotta, ianni, pellegrini}@lockless.it

***Abstract*—Modern computing platforms are based on multi-processor/multi-core technology. This allows running applications with a high degree of hardware parallelism. However, medium-to-high end machines pose a problem related to the asymmetric delays threads experience when accessing shared data. Specifically, Non-Uniform-Memory-Access (NUMA) is the dominating technology—thanks to its capability for scaled-up memory bandwidth—which however imposes asymmetric distances between CPU-cores and memory banks, making an access by a thread to data placed on a far NUMA node severely impacting performance. In this article, we tackle this problem in the context of shared event-pool management, a relevant aspect in many fields, like parallel discrete event simulation. Specifically, we present a NUMA-aware calendar queue, which also has the advantage of making concurrent threads coordinate via a non-blocking scalable approach. Our proposal is based on work deferring combined with dynamic re-binding of the calendar queue operations (insertions/extractions) to the best suited among the concurrent threads hosted by the underlying computing platform. This changes the locality of the operations by threads in a way positively reflected onto NUMA tasks at the hardware level. We report the results of an experimental study, demonstrating the capability of our solution to achieve the order of 15% better performance compared to state-of-the-art solutions already suited for multi-core environments.**

***Index Terms*—NUMA, calendar queue, non-blocking data structures**

1. Introduction

The current trend in computing architectures is characterized by an ever-increasing core-level parallelism. This is motivated by the need for scaled-up computing capabilities in face of the physical limits imposed by transistors technology [1], [2], [3]. This trend has brought concurrent and parallel programming paradigms to become mandatory for current and next-generation applications. Furthermore, it has brought non-blocking thread coordination [4] to assume a central role in the design and implementation of modern concurrent applications. Incidentally, this type of coordination, which avoids critical sections and simply exploits atomic Read-Modify-Write (RMW) instructions offered by the ISA—to let concurrent threads gather information on whether conflicting accesses to shared data structures have occurred—also allows resilience of performance degradation in CPU-stealing context, like Cloud computing.

However, modern hardware platforms are also characterized by asymmetries, which play as well a role in the actual performance deliverable by parallel/concurrent applications. One of the most important asymmetries is the so-called Non-Uniform-Memory-Access (NUMA). It is based on having memory banks organized in a configuration where each processor has some close bank(s)—this is the local NUMA node—and more far ones—which form the far NUMA nodes. Consequently, the need for accessing data from far NUMA nodes induces higher latency and traffic on the memory-interconnection hardware components. In these architectures, locality in the accesses not only plays a role for cache exploitation, but also for RAM exploitation, since accesses to far RAM banks should be avoided as much as possible.

The challenges posed by multi-processor/multi-core NUMA platforms have been faced since long time in the literature. In fact, most Operating System (OS) implementations offer API to directly control the placement of logical pages to RAM memory—or dynamically migrate them across NUMA nodes if required. Also, OS-level solutions have the capability to migrate threads, and the data they are currently touching, to favor accesses to the nearest (local) NUMA node of a given CPU core.

However, OS-level solutions only provide mid/long term binding between threads/data and NUMA nodes. Furthermore, the concept at the base of these solutions is to pack threads and their hot data on a same NUMA node, which is a solution not adequate for the case of very large thread counts—and CPU-bound threads—which share very large amounts of logical memory, possibly performing frequent fine grain operations on it. This is the case of last generation parallel simulation platforms, especially those based on speculative processing schemes [5]. In these scenarios, the "same NUMA-node" packing approach does not work since threads would simply be brought to compete for the same CPU-cores, leading to performance degradation.

Based on the above considerations, we feel that the

978-1-7281-7343-6/20/$31.00 ©2020 IEEE

NUMA aspect should be directly incorporated into the design of algorithms for managing shared data. Hence, in this article we present a NUMA-aware design of a shared event-pool based on the Calendar-Queue archetype. Our solution explicitly controls the locality of the accesses to hardware level memory resources, namely NUMA nodes. This is done by relying on cross-thread insertions of elements in the calendar, where the thread starting the insertion will not complete it in case the target time-bucket of the calendar is hosted by a far NUMA node. In these scenarios, we adopt a deferred-work strategy, leading other threads participating in the application, which are hosted by those far NUMA nodes, to actually finalize (flush) these insertion operations. On the other hand, we explicitly control the delay in the deferring scheme so to avoid that, when the elements whose insertion was deferred need to be extracted, then the whole burden of managing the flush of the deferred insertions related to the target time-bucket is put to an inconvenient thread—one running on a far NUMA node. In our scheme, we do not only share the data structure among threads, but we also share the work to be done on the data structure so as to make it be carried out by the most convenient threads. Our solution relies anyhow on non-blocking coordination of the threads in all of their operations, including the ones of posting the deferred work, and the ones of flushing it. This enables us to actually achieve non-blocking insertions/extractions from the calendar, which has already been shown to play a core role in concurrent event-pool management applications [5]. Based on all its features, we have called our data structure as Lock-Free Deferred-Work Calendar Queue (LFDWCQ).

We also report data for an experimental comparison of LFDWCQ with state-of-the-art non-blocking versions of the Calendar Queue—which are however not NUMA-aware—showing how our proposal can achieve up to 15% better performance when running a classical event-pool benchmark on top of a commodity machine equipped with 32 CPU-cores and 64GB of memory organized in 8 NUMA nodes.

The remainder of this article is structured as the following. In Section 2 we discuss related work. LFDWCQ is presented in Section 3. In Section 4 we report experimental results.

2. Related Work

As pointed out, an approach to cope with NUMA is based on OS level (or middleware level) facilities that dynamically place threads and their working set of logical pages on a same NUMA node [6], [7], [8], [9], [10]. These approaches have been shown to be effective in scenarios where threads actually express locality in the access to groups of logical pages. They are not suited for scenarios where the access pattern to data is highly variable, like when there is no stable binding of tasks by threads to portions of the shared data [5]. We cope with this limitation for the case of fully shared event-pool management, which is a core aspect in modern simulation systems to be run on top of multi-core machines.

A different approach has been provided in [11]. It is based on a general technique for NUMA awareness, called Fast Fly-weight Delegation (FFWD) introduced in [12], which resorts to a dedicated *server* thread to operate on remote memory banks in a NUMA topology. With this solution, locality at the hardware level is achieved since data hosted by a given NUMA node are only touched by specific server threads, which are bound to CPU-cores on that NUMA node. However, this approach is implicitly blocking, since a thread that asks a server thread to operate on the data structure is blocked until the server reply arrives. Contrarily, our solution is fully non-blocking—hence it is implicitly more scalable—and does not require pre-partitioning of threads into clients and servers (with respect to NUMA oriented data access).

The issue of memory-access latency asymmetries in NUMA architectures has also been tackled by using strategies where the same data are replicated across multiple NUMA nodes [13]. This makes them fast accessible to threads running on CPU-cores hosted by whatever NUMA node, at the cost of using mechanisms for making the replicated data instances coherent—this cost may become prohibitive for intensive and/or fine grain data update operations. In our solution we avoid at all this cost since we do not use replication. Furthermore, we are able to manage NUMA optimized accesses in scenarios with fine-grain tasks operating in update mode on the data structures—in fact, insertions and extractions from LFDWCQ are actual update operations.

As for the specific problem we tackle in this article, namely event-pool management, a wide literature exists on making the event pool efficient—in terms of both asymptotic and actual costs—like Calendar [14], Ladder [15] and LOCT [16] queues. Furthermore, enhancements of these data structures (or of other data structure flavors like lists or trees) have been proposed for the case of concurrent accesses, in particular by making the data structures accessible via non-blocking algorithms that enable scalability (see, e.g., [17], [18], [19], [20], [21]). However, the proposed algorithms have no intent to improve the locality of accesses with respect to architectures characterized by highly asymmetric memory, such as NUMA platforms. Hence, our proposal is an improvement over these literature solutions, as we also demonstrate via experimental results. In fact, beyond NUMA awareness, we also retain the lock-freedom property, since our data structure provides fully non-blocking operations

3. Lock-Free Deferred-Work Calendar Queue

LFDWCQ is built on top of a non-blocking and conflict-resilient Calendar Queue [21] (CRCQ). This data structure splits the domain of event timestamps into partitions, called virtual buckets, and maps them to a circular array of ordered linked lists, denoted as physical buckets. Also, the number of events per bucket is guaranteed to be bounded by a constant which is independent from the queue size, delivering amortized constant-time accesses. Whenever the number of

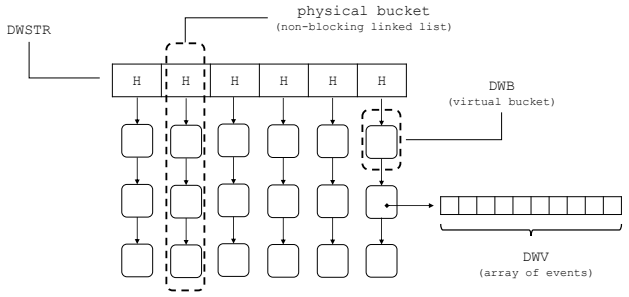


Figure 1. Layout of the front-end DWQ.

items is no longer balanced across the physical buckets, a resize phase doubles/halves the number of physical buckets in order to restore the balance, so as to keep control over the number of steps performed during insertions and extractions (since these steps depend on the number of events in each bucket). CRCQ provides all these features in a non-blocking fashion and jointly delivers conflict resiliency for extractions, which highly contend in the access to the bucket that keeps the minimum-timestamp event and are well-known to be critical for any concurrent priority queue because of their impact on caches in multi-core platforms [17], [22]. We designed our solution in order to maintain the same progress (lock-freedom) and scalability (conflict-resiliency) guarantee of the CRCQ.

In the following sections, we introduce the main idea at the core of LFDWCQ and describe its actual structure and the operations taking place on it.

3.1. The idea in a nutshell

LFDWCQ has three main design principles:

1) postpone (defer) far-future events insertions;
2) group them in a batch to control locality of memory accesses;
3) provide non-blocking progress of threads.

To achieve all these goals, we paired CRCQ with a front-end data structure called Deferred Work Queue (DWQ) aimed at maintaining events whose management has been deferred. In more detail, these events are not directly connected to the underlying calendar queue upon their insertion; rather, they are appended to DWQ in order to be processed later along a more favorable phase of execution of some thread. On the other hand, extractions are performed directly from the CRCQ, which—as mentioned—embeds advanced techniques towards conflict resiliency and scalability. Overall, events inserted into the front-end DWQ will be eventually migrated to the underlying (back-end) CRCQ.

In order to make our approach effective, adding items to DWQ has to be a low latency and low memory-footprint operation, otherwise the costs for posting events overpass the benefits given by their postponed batch-insertion. For this reason, the DWQ layout, shown in Figure 1, is mainly based on arrays. It resembles the classical calendar queue arrangement by having an array, called Deferred Work Struct (DWSTR), of linked lists. Differently from ordinary calendar queues, virtual buckets are explicitly maintained by the means of individual nodes, called Deferred Work Buckets (DWB), within physical buckets. In turn, each node (i.e. a virtual bucket) maintains deferred events in an unsorted array of fixed size, denoted as Deferred Work Values (DWV)[1]. Essentially, our DWQ is a three-dimensional calendar queue where two dimensions are materialized within arrays and one with the usage of ordered non-blocking linked lists [24].

Before items can be extracted, they have to be migrated from DWQ to the underlying CRCQ. Generally, inserting an item into a calendar queue is a cache-unfriendly operation because it involves the traversal of linked lists (the physical buckets) that are well-known to have a poor spatial locality and to be sub-optimal in terms of cache usage (at least for large-sized lists). This is exacerbated in the case of NUMA architectures, where a miss into the Last Level of Cache (LLC) might trigger a request to a remote cache and/or memory component. All these shortcomings are still present when migrating events from DWQ to the back-end CRCQ. However, our solution alleviates all these problems by migrating items falling in the same virtual bucket in batch. This avoids repeated traversals of nodes within the same physical bucket and allows to reuse most of the steps performed during a previous migration of an event, significantly increasing temporal locality of insertions.

Since extractions are performed from the underlying calendar queue, we need to ensure that all the items belonging to the current virtual bucket have been migrated from DWQ, allowing threads to obtain events by resorting to the original extraction logic provided by CRCQ (events with lower timestamps must be extracted before others). To achieve this goal, threads have to trigger a migration whenever a bucket becomes the new target for extractions—it becomes hot. However, this *reactive* approach is not suitable for scalability and NUMA-awareness. In fact, regardless its placement within the NUMA topology, any thread might trigger such a reactive migration, increasing the probability of conflicts.

In more detail, the current bucket (the one keeping the lower timestamp events) has become hot because it is concurrently targeted by all the threads performing extractions. Also, it is frequently updated by memory-write accesses to signal item removals. This has a dramatic impact on performance because of the costs associated with the cache-coherency protocols running on firmware. In order to do not worsen this already challenging scenario, we adopted a *proactive* approach for migrating events from DWQ to the calendar. In particular, instead of migrating items belonging to the already hot buckets, we flush in advance "mid-temperature" virtual buckets, namely those that are in the (near) future of the currently hot bucket. This guarantees that, whenever a flushed bucket becomes hot all its items have been already migrated. To further reduce the probability of conflict during migration phases, we also adopt

1. We could easily support dynamically sized arrays by resorting to lock-free dynamic vectors [23], but this is not the main focus of this work.

Algorithm 1 LFDWCQ algorithms

```
E1: procedure ENQUEUE(event e)
E2:     res ← DWQ.insert(e)
E3:     if res = FAIL then
E4:         res ← CRCQ.insert(e)
E5:     return res
D1: procedure DEQUEUE(void)
D2:     current_vb ← CRCQ.getIndexOfCurrentBucket()
D3:     if current_vb ≠ last_vb then
D4:         future_vb ← getWarmBucket(current_vb)
D5:         if ¬ DWQ.isMigrated(future_vb) then
D6:             DWQ.migrate(future_vb)        ▷ Proactive invocation
D7:         if ¬ DWQ.isMigrated(current_vb) then
D8:             DWQ.migrate(current_vb)       ▷ Reactive invocation
D9:         last_vb ← current_vb
D10:    return CRCQ.dequeue()
R1: procedure RESHUFFLE(void)
R2:     for each vb in DWQ do
R3:         DWQ.migrate(vb)
R4:     CRCQ.reshuffle()
```

assignments policy of virtual buckets to NUMA nodes, namely only those threads running on a specific NUMA node can flush a given bucket. Moreover, if multiple threads detect that they are migrating the same bucket, we allow just one thread to proceed while other threads can migrate other buckets or simply continue in their computation, reducing the probability of conflicts. Finally, such a proactive work can be performed only by threads that have invoked the dequeue API of LFDWCQ. This is a deliberate choice that allows to reduce the actual contention on a very-expensive prone-to-conflicts operation (event extraction) while still performing useful work.

We stress again that all these features have been provided in a non-blocking fashion thanks to the usage of custom state machines and algorithms aimed to maintain the same progress and correctness guarantees provided by the underlying CRCQ, namely lock-freedom [4] and linearizability [25].

3.2. LFDWCQ operations

LFDWCQ exposes the same API of a conventional priority queue. The logic embedded in the API functions is depicted in Algorithm 1. As hinted, insertions are deferred. On the other hand, extractions are never postponed. Consequently, the latter are first required to check that specific prerequisites have been satisfied before proceeding. In fact, we have to be sure that items belonging to the current virtual bucket—the one targeted by the extraction—have been migrated from DWQ to CRCQ. This step can be performed in two different ways: *proactively*, namely migrating items associated with a virtual bucket which is not the one currently used for extractions, and *reactively* in the opposite case. Such migrations introduce additional activities not required in the original extraction operations characterizing CRCQ, which we exploited as a back-off scheme to reduce contention and conflicts upon item removals.

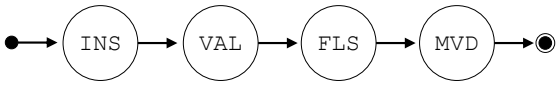


Figure 2. State machine representing the evolution of virtual buckets within the DWQ.

Since we want to provide non-blocking progress, the migration phase requires a careful handling of the virtual bucket states within the DWQ. Their evolution is driven by the state machine shown in Figure 2, whose states are described in the following.

- **INS:** indicates that the virtual bucket still accepts new events;
- **VAL:** signals that virtual bucket no longer accepts new items and a validation of already inserted items has started;
- **FLS:** tells us that the validation has completed and items are being migrated into the CRCQ;
- **MVD:** is used to communicate that all the items have been migrated and this instance of a virtual bucket can be collected to reclaim its memory.

The main role of these states is to ensure that items cannot be lost during DWQ operations due to the non-atomicity of a migration phase. As typical in non-blocking data structures [24], the state of a virtual bucket is materialized within the Least-Significant Bits of the conventional *next* field of a DWB, which is a node of a linked list. This ensures correct manipulation in case of concurrent insertions and removals of virtual buckets within the DWQ.

The reshuffle phase ensuring that the number of events is balanced across the physical buckets is not explicitly discussed since we simply resort to the algorithm provided by the underlying CRCQ. Consequently, whenever such an algorithm has to be triggered, we simply flush all the items stored within DWQ into the CRCQ by exploiting the migration algorithm to be discussed in Section 3.2.2.

3.2.1. DWQ insertion. The insertion logic (shown in Algorithm 2) resembles the insertion within a classical calendar queue. In fact, first we identify the entry of the DWSTR array that maintains the virtual bucket (DWB) that we want to update by inserting a new item, say the event e. Then, we perform a list traversal in order to detect the correct position of the DWB that will contain e upon insertion completion. If such a DWB does not exists, we simply create and connect a new one prefilled with e. Conversely, if it exists, we first check if its current state is compliant with a new insertion, namely its state is INS. In this case, we proceed by inserting the item within the DWV array associated with the target DWB. This is done with just two atomic instructions: one `Fetch&Add` to get a free index within DWV and one `Compare&Swap` to insert the item within the acquired entry. If the latter succeeds the operation is completed and we can return. Conversely, if it fails or the `Fetch&Add` returns an index outside the array boundaries, we simply proceed with the classical insertion

Algorithm 2 DWQ Insertion algorithm

```
I1:  procedure INSERT(event e)
I2:      vb_idx ← computeIndex(e)
I3:      bucket ← getDWB(vb_idx)
I4:      state ← getState(bucket)
I5:      if state ≠ INS then
I6:          return FAIL
I7:      cell_idx ← Fetch&Add(bucket.index)
I8:      cell_ptr ← &bucket.DWV[cell_idx]
I9:      if cell_idx ≥ bucket.size then
I10:         return FAIL
I11:     if ¬Compare&Swap(cell_ptr, NULL, e) then
I12:         return FAIL
I13:     return OK
```

into the underlying calendar queue (see Algorithm 1). As a last note, the same fall back path is executed whenever the target DWB is in a state that does not accept new items (any state different from INS).

3.2.2. DWQ migration. The migration protocol (depicted in Algorithm 3) is the core of our DWQ because it hides all the complexity of deferred management of insertions. In fact, we need to provide a non-blocking approach for this task, allowing multiple threads to collaborate while migrating the same virtual bucket. The migration has three main phases, each one corresponding to one of the three transitions of the state machine depicted in Figure 2.

The first step consists in ensuring that the to-be-migrated bucket becomes stable in terms of enqueued items. In particular, we need to ensure that there exists a moment in time such that no new insertion in the target DWB can succeed. To reach this goal, we set the current state of the bucket as VAL via `Compare&Swap`, indicating that a validation is in progress. This ensures that new upcoming insertions cannot succeed in adding a new item into DWB. However, some concurrent thread might still try to insert an event. Consequently, we simulate that the DWV array is full by acquiring all the cells with an individual `Fetch&Add`. Then, we publish the old value N (returned by the `Fetch&Add` instruction) with an atomic exchange. At this point, we have globally signalled that no items can be inserted in entries of the array whose index is greater than N. Since concurrent threads might have not completed their insertions, also in this case we cannot safely migrate items from DWB to the calendar queue. In particular, we need to perform a scan of the first N entries of the DWV array and block with a `Compare&Swap` instruction each cell that results as still empty. This guarantees that a thread that has acquired an index cannot complete its insertion into the array. After this "validation" phase, we are now sure that the number of items actually inserted in the to-be-migrated virtual bucket cannot change over time. Consequently, we apply another state transition by setting FLS as the current state of the DWB via `Compare&Swap`, starting the second phase of the migration algorithm called *flush*.

This phase consists of migrating items from DWB to the physical buckets of the underlying CRCQ. To reach this goal, we applied the same copy-and-validate strategy

Algorithm 3 DWQ Migration algorithm

```
M1:  procedure MIGRATE(DWB b)
M2:      while (state ← getState(b)) ≠ MVD do
M3:          if state = INS then
M4:              Compare&Swap(&b.state, INS, VAL)
M5:          if state = VAL ∧ b.pending_items = 0 then
M6:              tmp ← Fetch&Add(&b.index, MAX_DWB_SIZE)
M7:              tmp ← tmp% MAX_DWB_SIZE
M8:              b.pending_items ← tmp
M9:          if state = VAL ∧ b.pending_items ≠ 0 then
M10:             for i ← 0 to b.pending_items do
M11:                 if b.DWV[i] = NULL then
M12:                     Compare&Swap(&b.DWV[i],NULL,BLOCK)
M13:             Compare&Swap(&b.state, VAL, FLS)
M14:         if state = FLS then
M15:             for i ← 0 to b.pending_items do
M16:                 evt ← b.DWV[i]
M17:                 if evt ≠ BLOCK ∧ evt.replica = NULL then
M18:                     tmp ← clone(evt)
M19:                     CRCQ.insert_invalid(tmp)
M20:                     Compare&Swap(&evt.replica,NULL,tmp)
M21:                     validate(evt.replica)
M22:             Compare&Swap(&b.state, FLS, MVD)
```

implemented within the original CRCQ during a calendar resize. It consists of creating an invalid replica of the to-be-migrated event and then inserting such a copy within the target physical bucket. Clearly, since multiple copies of the same event can be inserted into the target bucket by multiple threads, we need to choose which one has to be considered as valid and hence be used for extraction—skipping this step might lead to multiple extractions of the same event, violating the priority queue semantic. This is achieved by having each thread trying to promote its copy as a master one by making the original event (the one within DWQ) point to its replica with a `Compare&Swap`. Such a complex algorithm guarantees non-blocking (wait-free) migrations that could not be guaranteed if an item were migrated by a unique thread—this would make the whole DWQ data structure blocking or, even worse, would lead to non-ordered extractions. This step is repeated until each item has been migrated into the calendar queue. Note that this adopted batch-insertion allows us to perform a single traversal of a physical bucket within the underlying CRCQ, providing amortized cost.

We signal that all the items have been migrated by setting the state of the virtual bucket as MVD. Even though the corresponding DWB cannot be longer used for insertions and migrations, it will be kept connected to the DWQ until it does not belong to the past, namely the highest priority event is held by a virtual bucket corresponding to a subsequent interval in the timestamp-based priority domain. This allows us to avoid corner cases, such as the continuous dematerialization and materialization of DWBs, whose management might hamper the responsiveness of our data structure.

3.2.3. LFDWCQ optimizations. There are three main aspects of LFDWCQ that have a relevant impact on performance since they might change the locality of memory

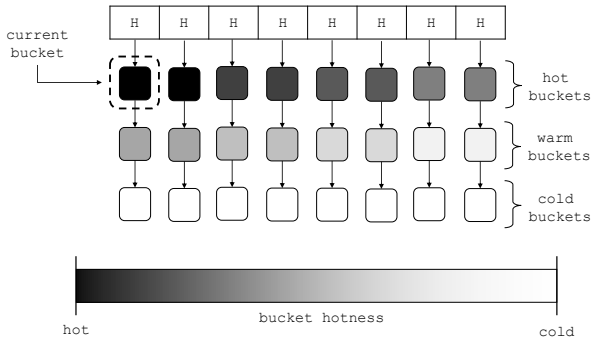


Figure 3. Visual representation of the bucket hotness, namely the likelihood that it becomes active for extractions along wall-clock time.

accesses and the effect of executing read-modify-write instructions. The first one is the average number of items that belongs to a virtual bucket. This clearly has an effect on the utilization of DWV arrays within DWBs of our DWQ. In fact, when a DWV is full, threads insert items directly into the underlying calendar queue. Consequently, the more virtual buckets are dense, the more threads can rely on DWQ to reduce the insertion latency. Moreover, since deferred enqueues are processed in a batch, we can tolerate longer lists (physical buckets) than both blocking and non-blocking calendar queues. In more detail, we used the same approach for bucket sizing adopted in CRCQ, which makes the number of events per bucket proportional to the average number of concurrent threads accessing the data structure.

The second key point for optimizing LFDWCQ consists in controlling the timeline of items' migrations from DWQ to CRCQ. Such an operation is carried out by extractions. On the one hand, since threads have to handle deferred work, the contention upon extractions and hence impact of conflicts is reduced. On the other hand, migrating items from DWQ to CRCQ is an operation characterized by a heavy-weight usage of RMW instructions, which might hamper performance due to their impact on caches. These effects can be alleviated if the updated cache lines are unlikely shared. To this aim, we make extractions proactively migrate items by flushing those in a subsequent bucket of the current one, namely the virtual bucket currently targeted by extractions. The idea is ensuring that when a virtual bucket becomes hot, namely it becomes the new target for extractions, it is already filled with all its (already migrated) items. This avoids that multiple threads try to migrate the same events, share cache lines and increase the pressure on the cache-coherency firmware.

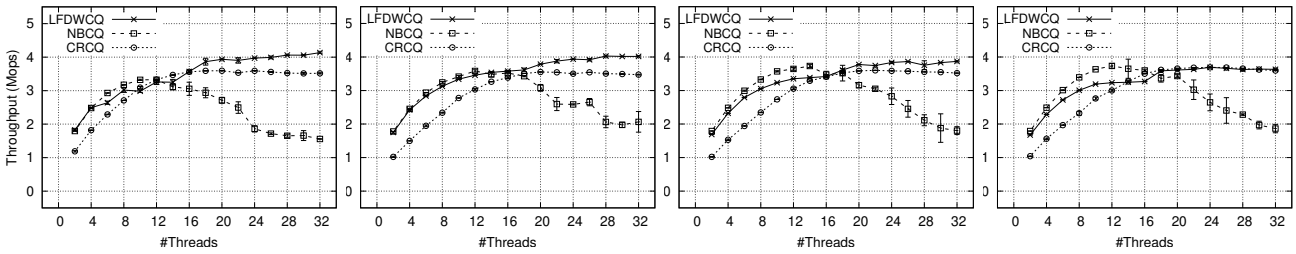
Virtual buckets that immediately follow the hot one in the priority domain will be active for extractions soon. Hence, we can define the *hotness* of a bucket B as the distance in the timestamp-based priority domain between B and the hottest one, namely the bucket currently used for extractions. A visualization of this concept is provided in Figure 3. As suggested before, migrating an already hot bucket increases the likelihood of conflicts. On the other hand, migrating *cold* buckets, those far away from the hottest one, might reduce the utilization of the DWQ and makes most of the insertions be performed directly on the underlying calendar queue, losing control on locality. Consequently, our proactive migration targets *warm* buckets, whose distance from the hottest one is enough large to avoid conflicts and not too far to reduce DWQ utilization. Since the advancement of the highest priority is fast as threads extract events, the beginning and the end of the warm region is strictly related to the actual level of concurrency insisting on the data structure. Basing on this consideration, we consider the first $2N$ buckets after the one currently used for extractions as either hot or warm, where N is the number of active threads. Since the hot region shifts by one bucket at a time, buckets near the hottest one have higher likelihood of being proactively migrated and the utilization of cold buckets is not hampered.

Whenever a thread detects a conflict during a proactive migration phase, we make just one thread proceed while the other can fallback by executing its extraction from the current bucket. Such a back-off scheme can be easily implemented by exploiting the result of an individual `Compare&Swap` performed by a thread. In particular, if a swap performed by a thread A fails, it means that another thread B is working on the same bucket. Thus, we avoid any additional conflict by making thread A stop running the migration protocol and proceed with a classical extraction from the CRCQ. To further reduce the probability of conflicts upon migration phases, buckets of both calendars are assigned to NUMA nodes in a circular fashion and we make threads proactively migrate only buckets assigned to the NUMA nodes on which they are running. The benefits introduced by this simple scheme are two-fold. On the one hand, we reduce the set of threads that might compete for migrating a given bucket, hence the likelihood of conflicts. On the other hand, memory requests issued during migrations do not propagate towards far NUMA nodes, reducing latency for accessing and updating memory.

In order to make such a binding effective, we need to ensure that the memory buffers used for a virtual bucket and its events are effectively allocated on the target NUMA node. Ensuring this require from none to small adjustments on the underlying memory allocator. In fact, if threads are pinned to run on a specific core, no countermeasures have to be taken at all because the OS (e.g. the Linux kernel) typically allocates memory frames on the NUMA node nearest to the core issuing the request and/or first accessing the just allocated page. Conversely, if thread execution might migrate on different cores and NUMA nodes, a NUMA-aware allocator is required to have full control of memory buffer placement.

4. Experimental evaluation

We have compared the behavior of LFDWCQ with the ones of recent implementations of non-blocking calendar queues, namely the Non-blocking Calendar Queue (NBCQ) presented in [20] and the Conflict-Resilient Calendar Queue (CRCQ) [21]. All the data structures use the Epoch-Based



(a) 400K events (b) 1M events (c) 2M events (d) 4M events

Figure 4. Average Throughput for different queue sizes and thread counts.

Garbage Collector described in [26]. The used benchmark is the well-know Classic HOLD [27] where the queue is initially pre-populated to reach a target size and then it is stressed out by having multiple threads performing a hold operation, namely an extraction immediately followed by an insertion. The timestamp increment of the new inserted event (compared to the last extracted one) is obtained via an exponential distribution with mean equal to 1. Our aim was to evaluate the performance of the data structure at steady state, so we ran it for 10 seconds after the pre-population phase has completed. The performance metric we used is the average throughput computed over 10 different executions.

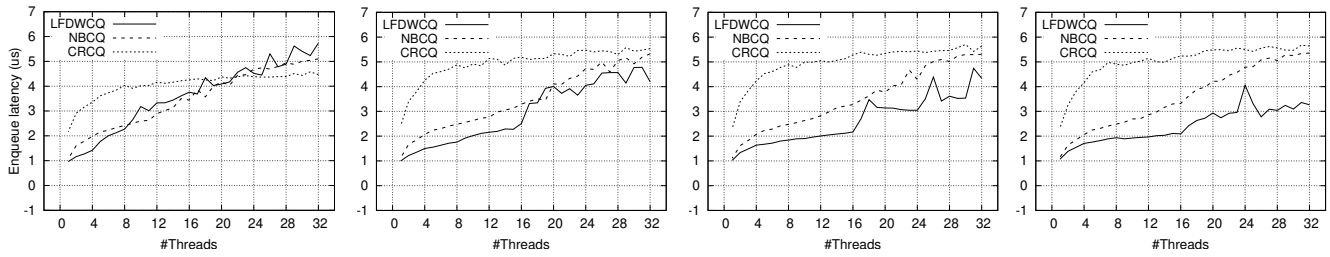
All the experiments have been carried out on an HP Proliant server equipped with 4 AMD Opteron 6128 processors running at 2 GHz. Each processor has 8 cores for a total of 32 hardware threads. The machine has 64GB of RAM arranged in 8 NUMA nodes and runs Debian 9.2. (version 5.4.0 of Linux kernel) as Operating System. All the code of the tested solutions is written in C and compiled with gcc 9.2.1 with the highest optimization flag (O3).

Figure 4 shows the average throughput (and its standard deviation) while running the benchmark with queue size ranging from $4 \cdot 10^5$ to $4 \cdot 10^6$ and different thread counts (from 1 to 32). It clearly shows that our approach pays off and has an improved behavior across all evaluated scenarios. To make this concept clear consider the trend of NBCQ and CRCQ while increasing both queue size and concurrency level. On the one hand, if the thread count is lower than 16, NBCQ has higher throughput than CRCQ. On the other hand, when the concurrency level increases, the roles are reversed. This is because, the conflict resiliency provided by CRCQ is traded off with latency (a trend well-known in the literature [17]), penalizing performance at lower concurrency/contention.

Thanks to its NUMA-awareness, LFDWCQ allows alleviating the CRCQ inefficiencies with lower thread counts by improving the locality of memory accesses. In fact, the throughput of LFDWCQ always stands between the one of CRCQ and NBCQ when the number of threads is lower than 16, maintaining a distance from the optimum bounded to 15% and reducing the performance loss of CRCQ compared to NBCQ by at least 50% and up to 100%. This is extremely relevant since the actual concurrency level of applications can vary. Consequently, improving the behavior of non-blocking calendar queues in a wider range of scenarios allows to relief the developer from the burden of choosing the implementation that best fits her/his use case. When the number of active threads oversteps 16, LFDWCQ provides up to the 15% performance improvement w.r.t. the optimum in 3 out of 4 cases, showing that our approach towards NUMA-awareness is effective also at high levels of concurrency/contention. In particular, when the queue size is smaller than 4 millions, the trend is still up rising, suggesting that, if we could increase the number of cores, the gap between CRCQ and LFDWCQ would likely increase. Consequently, an improved scalability has emerged as a secondary benefit of our approach. However, when the queue size is set to 4 millions, we achieve the same performance of the original approach, showing no gain at full concurrency. This is because the benefits of batch insertions are reduced when the density of the events per virtual bucket increases too much.

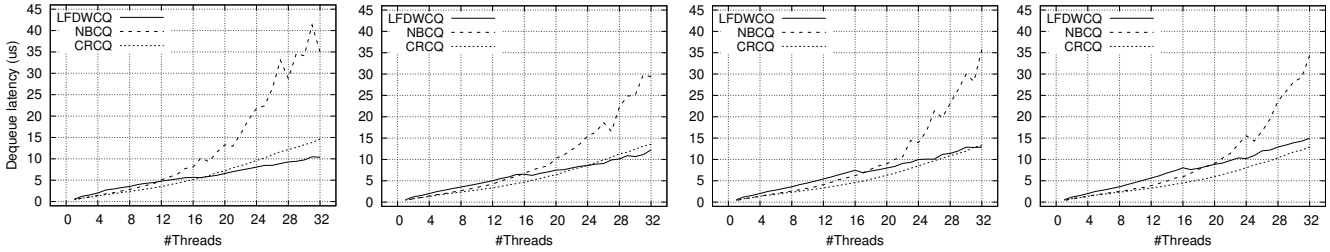
This behavior is more evident when observing the latency of both insertion and extraction routines shown in Figure 5 and 6, respectively. The costs per insertion (enqueue) decreases when the event density increases because we have more opportunities to exploit DWQ for insertion of future events, leading up to a 50% improvement at full concurrency and largest queue size. On the other hand, the batch migration of items performed by extraction (dequeue) invocations alleviates the impact of high contention with smaller queue size (up to the 33% improvement w.r.t. pure CRCQ). However, the latency becomes larger than the one provided by CRCQ when the queue size increases, compensating the gains achieved by the enqueues. This suggests that our approach can be further extended by removing the batch migration from the critical path of dequeue executions, e.g. resorting to helper threads whose role is just the one of migrating items from DWQ to the underlying calendar queue. This is a direction we will explore as future work.

Finally, we analyzed the impact on caches of the different calendar queue implementations by monitoring the miss ratio of LLC accesses. This has been computed by resorting to Hardware Performance Counter statistics gathered with LIKWID [28], a well-known suite to access such low-level monitors—the hardware events to be sampled have been chosen according to the formula for LLC miss-ratio given in [29]. Figure 7 shows that our approach reduces the miss ratio by 50% independently from the queue size. This shows that rely on work deferring to insert events in batch is an



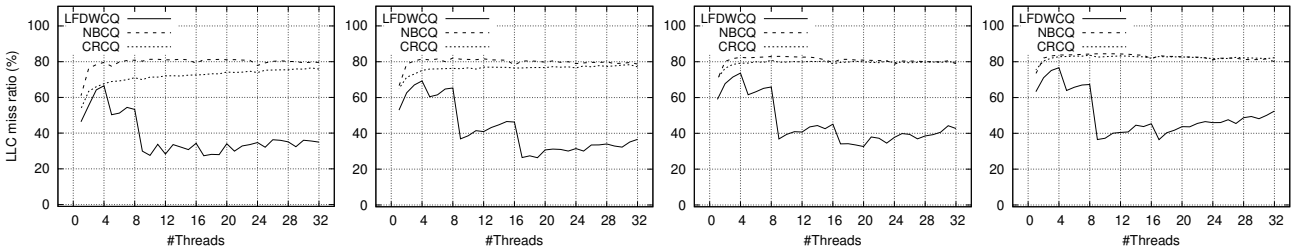
(a) 400K events (b) 1M events (c) 2M events (d) 4M events

Figure 5. Average enqueue latency.



(a) 400K events (b) 1M events (c) 2M events (d) 4M events

Figure 6. Average dequeue latency.



(a) 400K events (b) 1M events (c) 2M events (d) 4M events

Figure 7. Miss ratio LLC.

effective technique to improve locality.

5. Conclusions

The management of event pools plays a central role in many applications, including simulation. The recent hardware trend towards multi/many-core platforms has therefore generated a great interest in having event-pool management algorithms capable of providing scalability in face of concurrent accesses. However, much less explorations have been performed in order to keep into account another factor characterizing modern parallel machines, particularly Non-Uniform-Memory-Access (NUMA). In this article, we have presented the Lock-Free Deferred-Work Calendar Queue, an event pool, based on the Calendar-Queue archetype, which jointly offers scalable thread coordination—via non-blocking solutions–and NUMA-awareness. The latter feature has been achieved via an approach that changes the actual locality of operations by threads in the different memory banks in the NUMA architecture, an objective that has been reached by incorporating work deferring concepts into our solution. Performance tests with a classical benchmark, executed on top of an off-the-shelf medium-end machine equipped with 32 physical cores and 8 NUMA nodes–globally entailing 64GB of RAM—have shown how our solution can provide up to 15% performance boost compared to state-of-the-art event-pool management algorithms already suited for multi-core machines.

References

[1] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 176–188. [Online]. Available: http://doi.acm.org/10.1145/106972.106991

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on*

Computer Architecture, ser. ISCA 11. New York, NY, USA: Association for Computing Machinery, 2011, p. 365376. [Online]. Available: https://doi.org/10.1145/2000064.2000108

[4] M. Herlihy and N. Shavit, "On the nature of progress," in *Proceedings of the 15th International Conference on Principles of Distributed Systems*, ser. OPODIS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 313–328. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25873-2_22

[5] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia, "The ultimate share-everything PDES system," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Rome, Italy, May 23-25, 2018*, F. Quaglia, A. Pellegrini, and G. K. Theodoropoulos, Eds. ACM, 2018, pp. 73–84. [Online]. Available: https://doi.org/10.1145/3200921.3200931

[6] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 13. New York, NY, USA: Association for Computing Machinery, 2013, p. 381394. [Online]. Available: https://doi.org/10.1145/2451116.2451157

[7] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing googles warehouse scale computers: The numa experience," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA 13. USA: IEEE Computer Society, 2013, p. 188197. [Online]. Available: https://doi.org/10.1109/HPCA.2013.6522318

[8] B. Lepers, V. Quema, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 277–289. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

[9] A. Pellegrini and F. Quaglia, "NUMA time warp," in *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015*, S. J. E. Taylor, N. Mustafee, and Y. Son, Eds. ACM, 2015, pp. 59–70. [Online]. Available: https://doi.org/10.1145/2769458.2769479

[10] I. D. Gennaro, A. Pellegrini, and F. Quaglia, "Os-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. IEEE Computer Society, 2016, pp. 291–300. [Online]. Available: https://doi.org/10.1109/CCGrid.2016.91

[11] F. Strati, C. Giannoula, D. Siakavaras, G. Goumas, and N. Koziris, "An adaptive concurrent priority queue for numa architectures," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 135144. [Online]. Available: https://doi.org/10.1145/3310273.3323164

[12] S. Roghanchi, J. Eriksson, and N. Basu, "Ffwd: Delegation is (much) faster than you think," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 342358. [Online]. Available: https://doi.org/10.1145/3132747.3132771

[13] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for numa architectures," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 207221. [Online]. Available: https://doi.org/10.1145/3037697.3037721

[14] R. Brown, "Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, vol. 31, no. 10, p. 12201227, Oct. 1988. [Online]. Available: https://doi.org/10.1145/63039.63045

[15] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation," *ACM Trans. Model. Comput. Simul.*, vol. 15, no. 3, p. 175204, Jul. 2005. [Online]. Available: https://doi.org/10.1145/1103323.1103324

[16] F. Quaglia, "A low-overhead constant-time lowest-timestamp-first cpu scheduler for high-performance optimistic simulation platforms," *Simulation Modelling Practice and Theory*, vol. 53, pp. 103 – 122, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X15000209

[17] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Principles of Distributed Systems*, R. Baldoni, N. Nisse, and M. van Steen, Eds. Cham: Springer International Publishing, 2013, pp. 206–220.

[18] S. Gupta and P. A. Wilsey, "Lock-free pending event set management in time warp," in *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM PADS 14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1526. [Online]. Available: https://doi.org/10.1145/2601381.2601393

[19] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A non-blocking priority queue for the pending event set," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS16. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, p. 4655.

[20] ——, "A lock-free o(1) event pool and its application to share-everything pdes platforms," in *Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications*, ser. DS-RT 16. IEEE Press, 2016, p. 5360. [Online]. Available: https://doi.org/10.1109/DS-RT.2016.33

[21] ——, "A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1526. [Online]. Available: https://doi.org/10.1145/3064911.3064926

[22] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," *SIGPLAN Not.*, vol. 50, no. 8, p. 1120, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2858788.2688523

[23] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Lock-free dynamically resizable arrays," in *Proceedings of the 10th International Conference on Principles of Distributed Systems*, ser. OPODIS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 142–156. [Online]. Available: http://dx.doi.org/10.1007/11945529_11

[24] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314. [Online]. Available: http://dl.acm.org/citation.cfm?id=645958.676105

[25] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[26] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2004.

[27] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 2, p. 157209, Apr. 1997. [Online]. Available: https://doi.org/10.1145/249204.249205

[28] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW 10. USA: IEEE Computer Society, 2010, p. 207216. [Online]. Available: https://doi.org/10.1109/ICPPW.2010.38

[29] P. J. Drongowski and B. D. Center, "Basic performance measurements for amd athlon 64, amd opteron and amd phenom processors," *AMD whitepaper*, vol. 25, 2008.