Black-box Concurrent Data Structures for NUMA Architectures

Irina Calciu
VMware Research

Siddhartha Sen
Microsoft Research

Mahesh Balakrishnan
Yale University

Marcos K. Aguilera
VMware Research

Abstract

High-performance servers are non-uniform memory access (NUMA) machines. To fully leverage these machines, programmers need efficient concurrent data structures that are aware of the NUMA performance artifacts. We propose Node Replication (NR), a *black-box* approach to obtaining such data structures. NR takes an arbitrary sequential data structure and automatically transforms it into a NUMA-aware concurrent data structure satisfying linearizability. Using NR requires no expertise in concurrent data structure design, and the result is free of concurrency bugs. NR draws ideas from two disciplines: shared-memory algorithms and distributed systems. Briefly, NR implements a NUMA-aware shared log, and then uses the log to replicate data structures consistently across NUMA nodes. NR is best suited for contended data structures, where it can outperform lock-free algorithms by 3.1x, and lock-based solutions by 30x. To show the benefits of NR to a real application, we apply NR to the data structures of Redis, an in-memory storage system. The result outperforms other methods by up to 14x. The cost of NR is additional memory for its log and replicas.

1. Introduction

Concurrent data structures are used everywhere in the software stack, from the kernel (e.g., priority queues for scheduling), to application libraries (e.g., tries for memory allocation), to applications (e.g., balanced trees for indexing). These data structures, when inefficient, can cripple performance.

Due to recent architectural changes, high-performance servers today are non-uniform memory access (NUMA) machines. In such machines, the cores are grouped into *nodes*, where each node has some local cache and memory. Although a node can access the memory of other nodes, it is faster to access local memory and to share cache lines within a node. To fully harness the power of NUMA, software designers need NUMA-*aware data structures*, which reduce cross-node communication and minimize accesses to remote caches and memory [17, 21, 50]. Without these characteristics, performance falters as we add more nodes (§2).

Unfortunately, there are few NUMA-aware concurrent data structures, and designing new ones is hard. The key challenge is how to deal with contention on the data structure, where simple techniques limit concurrency and scale poorly, while efficient techniques are complex, error-prone, and rigid (§3).

We propose a new technique called Node Replication (NR) to obtain NUMA-aware data structures, by automatically transforming any sequential data structure into a corresponding concurrent NUMA-aware structure. NR is general and *black-box:* it requires no inner knowledge of the structure and no expertise in NUMA software design. The resulting concurrent structure provides strong consistency in the form of linearizability [38].

NR combines ideas from two disciplines: distributed systems and shared-memory algorithms. NR maintains per-node replicas of an arbitrary data structure and synchronizes them via a shared log (an idea from distributed systems [6, 43, 55]). The shared log is realized by a hierarchical, NUMA-aware design that uses flat combining [30] within nodes and lock-free appending across nodes (ideas from shared-memory algorithms). With this interdisciplinary approach, only a handful of threads need to synchronize across nodes, so most synchronization occurs efficiently within each node.

We evaluate NR to determine when it performs well compared to the alternatives. We find that NR excels under contention, in which an operation often affects the output of other operations. On a contended priority queue and a dictionary, NR can outperform lock-free algorithms by up to 2.4x and 3.1x with 112 threads; and NR can outperform a lock-based solution by 8x and 30x on the same data structures. To demonstrate the benefits to a real application, we apply NR to the data structures of the Redis storage server [1]. Many systems have shown how servers can scale the handling of network requests and minimize RPC bottlenecks [39, 46, 47, 52, 59]. There is less research on how to scale the servicing of the requests. These systems either implement a simple service (e.g., get/put) that can partition requests across cores [39, 46]; or they develop sophisticated concurrent data structures from scratch to support more complex operations [47], and doing this requires expertise in concurrent algorithms. This is where our black-box approach

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037721

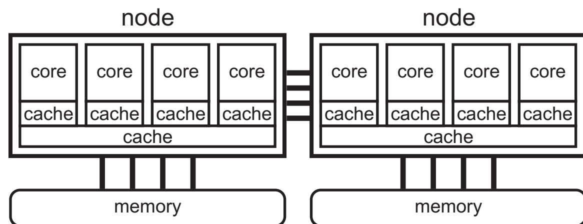


Figure 1. NUMA architecture: cores are grouped into nodes, each with its local memory. Nodes are connected by an interconnect, so that cores in one node can access the remote memory of another node, but these accesses come at a higher cost. Typically, cores have local caches, and cores on a node share a last level cache.

comes handy: NR provides these concurrent data structures automatically from sequential implementations. For Redis, we were able to convert a sequential sorted set into a concurrent one with just 20 new lines of wrapper code. Moreover, experiments show that NR outperforms data structures obtained from other methods by up to 14x.

Our approach has three limitations. First, NR incurs space overhead due to replication: it consumes *n* times more memory, where *n* is the number of nodes. Thus, NR is best suited for smaller structures that occupy just a fraction of the available memory (e.g., up to hundreds of MB). Second, NR is *blocking*: a thread that stops executing operations can block the progress of other threads; in practice, we did not find that to be a problem as long as threads keep executing operations on the data structure. Finding a non-blocking variant of NR is an interesting research direction. Finally, NR may be outperformed by non-black-box algorithms crafted for a given data structure—for example, a lock-free skip list running on low-contention workloads, or a NUMA-aware stack. Thus, the generality of black-box methods has some cost. However, in some cases NR outperforms even the crafted algorithms; we observe this for the same lock-free skip list running instead on high-contention workloads.

2. Background, motivation, goals

NUMA architectures. Our work is motivated by recent trends in computer architecture; to accommodate many cores, machines have adopted a NUMA architecture, where cores are clustered into groups called NUMA nodes or simply *nodes* (Figure 1). Each core has one or more private caches and each node has a shared cache. Sharing a cache line within a node is more efficient than across nodes because the cache coherence protocol operates more efficiently within a node. Each node has some local memory, and a core can access local memory faster than memory in a remote node. A similar architecture—non-uniform cache access (NUCA)—has a single shared memory but nodes have local caches as in NUMA. Our ideas are applicable to NUCA too.

NUMA is everywhere now. A high-performance Intel server might have 8 processors (nodes), each with 18 cores. AMD and Oracle have similar machines. To best use these cores, we need appropriate concurrent data structures.

Concurrent data structures. Concurrent data structures permit many threads to operate on common data using a high-level interface. When accessed concurrently, the semantics are typically defined by a property called linearizability [38], which provides strong consistency. Linearizability requires that each operation appear to take effect instantly at some point between the operation's invocation and response.

The key challenge in designing concurrent data structures is dealing with *operation contention*, which occurs when an operation often affects the output of another operation. More precisely, given an execution, we say that an operation affects another if the removal of the first causes the second to return a different result. For example, a write of a new value affects a subsequent read. A workload has operation contention if a large fraction of operations affect a large fraction of operations occurring soon after them. Examples include a storage system where users read and write a popular object, a priority queue where threads often remove the minimum element, a stack where threads push and pop data, and a bounded queue where threads enqueue and dequeue data. Non-examples include read-only workloads and write-only workloads where writes return void. Operation contention is challenging because operations must observe each other across cores.

Goals and non-goals. Our goal is to provide NUMA-aware concurrent data structures. We are particularly interested in developing structures that work well under operation contention, where existing methods (§3) perform poorly: their performance *drops* with more cores, especially as we cross node boundaries (§8). We want to do better. The ideal is to provide perfect scalability with the number of cores. With update-heavy contended workloads, we want to at least avoid the performance drops, so that the parallelizable parts of the application can benefit from more cores without being hindered by the data structures. Further, we want to obtain data structures using black-box techniques, so that any data structure can be made concurrent and NUMA-aware without effort or knowledge about NUMA programming. Moreover, we would like to provide strong consistency semantics (linearizability). We do not expect our black-box data structures to always outperform specialized data structures with tailored optimizations, but we hope to be competitive in a broad class of workloads. We do not wish to automatically convert single-threaded applications to multiple threads; many applications are multi-threaded already; if they are not, converting them remains a manual engineering procedure: applications have a broad interface, unlike sequential data structures, so are less amenable to black-box methods.

3. Related work

Data structures. Currently, developers have five options when they need a NUMA-aware data structure. The first is to use existing NUMA solutions, but there are very few NUMA data structures available [17, 50, 53].

The second option is to use existing concurrent data structures oblivious of NUMA—called uniform memory access (UMA) structures—including lock-based, lock-free, and wait-free algorithms. These algorithms are not sensitive to the asymmetry and limitations of NUMA, which potentially hinders their performance [9, 42, 44, 45]. Moreover, these solutions are not applicable when applications need custom data structures or compose many data structures, which is the case of Redis.

The third option is to use existing black-box methods. But such methods are currently targeted at UMA; like the second option, they are not designed for NUMA. Herlihy's universal construction [33] is a theoretical breakthrough in wait-free algorithms; however, despite improvements (e.g., [5, 24]), it remains impractical due to high copying overheads. OpLog [12] uses local logs to scale updates that return void, but underperforms on reads or updates that return values. Predictive log synchronization [57] scales, but provides weak consistency and is limited to data structures with prediction mechanisms. Delegation [16] reduces synchronization overheads, but it does not scale to many cores. Flat combining [30]—which we leverage within each node—is designed to mitigate the cost of coordination, but does not scale well because it restricts concurrency. Parallel flat combining [31] improves scalability with many combiners but applies only to certain data structures (synchronous queues).

The fourth option is to design new data structures that work well under NUMA, using one of many techniques. The most common is to use monitors or coarse-grained locks: a thread locks the entire data structure before executing an operation. This technique does not scale because it inhibits concurrent access to the data structure. An alternative to coarse-grained locking is to lock finer sections of code, but this is complex and can cause subtle bugs and unexpected races [37]. Software and hardware transactional memory [36, 58] have emerged as alternatives to locks. However, it is unlikely that software transactional memory can outperform carefully crafted concurrent data structures written by experts [63], and prior work has shown that hardware transactional memory also suffers from NUMA performance pathologies [14]. Another alternative to locks is Read-Copy-Update (RCU [49]) or Read-Log-Update (RLU [48]); with these methods, NUMA-awareness must be provided manually; moreover, RCU is notably hard to use in general; RLU is easier but still requires an expert to decide what to rlu_lock and when, similarly to fine-grained locking. Another technique is to develop lock- and wait-free data structures, which rely on algorithmic design to dispense with locks and transactions. Unfortunately, designing new concurrent data structures is hard and prone to subtle concurrency bugs [60]. Moreover, lock- and wait-free algorithms are rigid: it is hard to extend them to support operations for which they were not originally designed. Another approach is to use asynchronized concurrency patterns [22] for search data structures. But designing the algorithm remains a complex data-structure-dependent task. Another approach is to modify the cache coherence protocol to deal with contention [28], but this requires hardware changes.

The fifth option is to synthesize code mechanically. Xiang and Scott [63] propose a compiler tool that takes lock-based code with user annotations to generate concurrent data structures. Hawkins et al. [29] synthesize the data structures and locking code from concurrent relations. These approaches require programmer assistance or programming paradigms not based on concurrent data structures.

NR is most similar to the third approach above, but targets NUMA systems.

Similar goal, different mechanism. Much prior work has proposed replication of data as a way to optimize performance in shared-memory. These works differ from ours in the *context* and in the *generality* of replication. None of these works propose methods of replication that can transform sequential data structures into concurrent ones in a black-box manner. That is, the key innovation of our paper is not the idea of using replication, but rather *how* to do it in a general way for data structures. At the OS level, there is much work on NUMA memory management that replicates pages for performance (e.g., [10, 15, 19, 20, 62]). Because of the high copying costs, replication is intended for data that changes rarely, not concurrent data structures. Memprof [41] mentions replication as an optimization for NUMA systems, but only covers replication of read-only objects (e.g., a read-only matrix is replicated on creation). Munin [8] is a DSM system that uses replication in the cache coherence mechanism; there, replication is done by message-passing algorithms. Similarly, Barrelfish [7] uses message-passing algorithms to maintain the consistency of replicas of kernel data structures; an enhanced message-passing algorithm was subsequently proposed in [23]. Shahmirzadi et al. [56] analyze the tradeoffs between replication and partitioning in designing a high-performance map in a message passing system. Tornado [27] and K42 [40] can replicate clustered objects, but the replication is left to the specific implementation of each object. Corey [11] allows applications to control which data structures are shared across cores.

Different goal, similar mechanism. Our method is based on in-memory logs. Prior work has used in-memory logs, but for goals different from ours. Herlihy and Calciu [34] implement a replicated transactional memory by using a shared transactional log to synchronize the replicas. In predictive log synchronization [57], threads log updates to a common log, and the system keeps exactly two replicas, so that threads can read a consistent replica while the other is

being updated with the updates in the log. OpLog [12] keeps multiple independent logs, one per core, to optimize update-heavy workloads. Updates are deferred by recording them in the local logs. When a read occurs, the system applies updates in all logs.

4. Overview

Assumptions and API. To work with an arbitrary data structure, our approach expects a sequential implementation of the data structure as a class with three generic methods:

$Create() \rightarrow Ptr$
$Execute(ptr, op, args) \rightarrow Result$
$IsReadOnly(ptr, op) \rightarrow Boolean$

The *Create* method creates an instance of the data structure, returning its pointer. The *Execute* method takes a data structure pointer, an operation, and its arguments; it executes the operation on the data structure, returning the result. The method must produce side effects only on the data structure and it must not block. Operation results must be deterministic; we allow nondeterminism if it does not affect the results (e.g., levels of nodes in a skip list); for randomization that affects the results, one can use a pseudo-random number generator with a known seed. The *IsReadOnly* method indicates if an operation is read-only; we use this information for read-only optimizations in NR. Our technique provides a new method *ExecuteConcurrent* that can be called concurrently from different threads.

Basic idea. NR replicates the data structure on each node, and uses different techniques to coordinate threads within and across nodes. At the highest level, NR leverages the fact that coordination within a node is cheaper than across nodes.

Within each node, NR uses flat combining (a technique from concurrent computing [30]) to batch updates from threads in the same node. Flat combining works by electing a leader for the node, called *a combiner*, which handles outstanding update operations from threads within the node. Leadership is short-lived: the leader abdicates when it finishes executing the outstanding updates, up to a maximum number. Batching can gather many operations even though threads have at most one outstanding operation each, because there are many threads per node (e.g., 28 in our machine). Batching helps because it localizes synchronization within a node.

Across nodes, threads coordinate through a shared log (a technique from distributed systems [6, 43, 55]). The combiner of each node reserves entries in the log, writes the outstanding update operations to the log, brings the local replica up-to-date by replaying the log if necessary, and executes the local outstanding update operations.

For efficiency, NR handles read operations differently, by reading directly from the local replica. To satisfy linearizability, a thread executing a read must replay the log on the local replica at least until the last operation that completed before the read started. It can skip later operations in the log, because they are concurrent.

An optional optimization is to use a dedicated combiner for each node, which can replay the log even before local threads execute their operation.

NR brings multiple efficiency benefits:

- *Limited cross-node synchronization and contention:* NR executes a lock-free append to the log, using an uncontended compare-and-swap (CAS) on the log tail to reserve entries. Only the combiners execute the CAS, so the cross-node synchronization is limited to one thread per node, which is a small number (typically 2–8). In addition, the cost of a CAS is amortized over many operations due to batching.
- *Parallel reads and writes to the log:* Combiners update local replicas and write many operations to the log in parallel (after reserving entries for them).
- *Local parallel reads:* Read operations run in parallel and locally, if the replica is fresh. Checking for freshness might fetch a cache line across nodes, but this fetch populates the local cache and benefits many local readers. Readers execute in parallel with combiners on different nodes, and with the local combiner when it is filling entries in the log.
- *Compact representation of shared data:* Operations often have a shorter description than the effects they produce, and thus communicating the operation via the log incurs less communication across cores than sharing the modifications to the data structure.

A complication is that the log entries need to be recycled, but only after all the replicas have been updated using those entries. NR uses a lightweight lazy mechanism for recycling the entries that avoids synchronization by delegating responsibility to one of the threads.

5. The NR algorithm

NR replicates the data structure across nodes using a log realized as a shared circular buffer. This buffer can be allocated from the memory of one of the NUMA nodes, or it could be spread across nodes; our current implementation does the former. The log is accessed by at most one thread per node (§5.1), and it provides coordination and consistency across nodes. Within a node, threads coordinate to access the log and update the local replicas (§5.2). A node may have dozens of threads, but they can leverage a shared last level cache for fast coordination.

5.1 Inter-node coordination: circular buffer

The log is a circular buffer that stores update operations on the data structure. It has a variable *logTail* containing the index of the next available entry. For now, assume the buffer is unbounded; we discuss the wrap-around in Section 5.6.

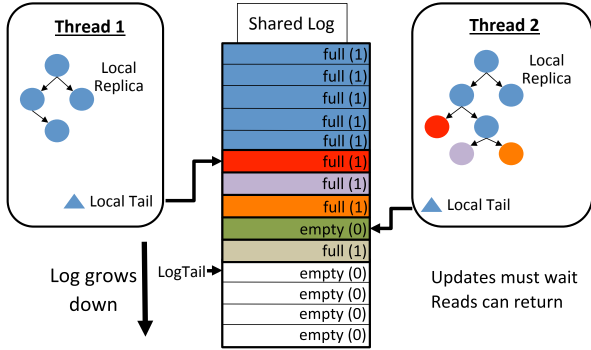


Figure 2. NR algorithm, shared log. *logTail* indicates the first unreserved entry in the log. Each *localTail* indicates the next operation in the log to be executed on each local replica. Threads 1 and 2 are the combiners for nodes 1 and 2. Thread 1's replica executed 5 operations from the log. Thread 2's replica executed 3 more operations and found a reserved entry that is not yet filled. A combiner must wait for all empty entries preceding its batch in the log. Readers can return when they find an empty entry (§5.3).

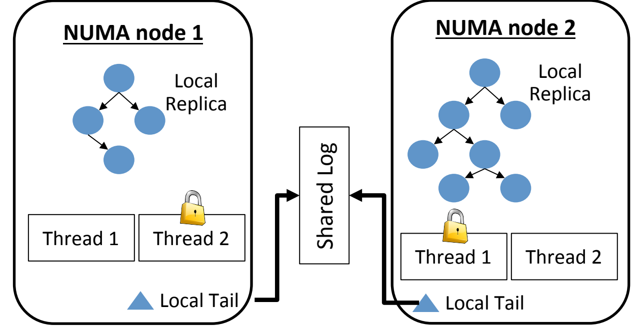


Figure 3. NR algorithm, per-node replicas. Threads located on the same node share a replica and coordinate access to the replica using a combiner lock.

Each node has a replica of the data structure and a variable *localTail* indicating how far in the log the replica has been updated. A node elects a temporary leader thread called a *combiner* to write to the buffer (§5.2).

The combiner writes many operations (a batch) to the log at a time. To do so, it first allocates space by using a CAS to advance *logTail* by the batch size. Then, it writes the buffer entries with the operations and arguments. Next, it updates the local replica by replaying the entries from *localTail* to right before the entries it allocated. In doing so the combiner may find empty entries allocated by other threads; in that case, it waits until the entry is filled (identified by a bit in the entry). Figure 2 shows two combiners accessing the log to update their local replicas, which they do in parallel.

5.2 Intra-node coordination: combining

Within a node, threads use flat combining [30] to coordinate. Roughly, flat combining elects a temporary leader, called *combiner*, to execute outstanding operations of other threads. NR applies flat combining once at each node, where each combiner executes the operations of its node. The combiners coordinate to write the log (§5.1).

More precisely, to execute an operation, a thread posts its operation in a reserved slot[1] and tries to become the combiner by acquiring the *combiner lock*. The combiner reads the slots of the threads in the node, marks filled slots by setting a reserved bit in the slot, and remembers the number of slots filled (the batch size B). The combiner then proceeds as explained in §5.1 to write the B operations to the log, from *startEntry* to *endEntry* in Algorithm 1, and update the local replica up to *startEntry*. Next, the combiner executes the operations in the slots it marked before, conveying the results to each thread that submitted an operation. Note that the combiner does not use the newly written entries in the log to update its replica with its own batch. Instead, it uses the combining slots, which are local to the node.

While original flat combining could opportunistically execute new operations outside the current batch, NR cannot do that because it has multiple combiners. To avoid small inefficient batches, the combiner in NR waits if the batch size is smaller than a parameter *MIN_BATCH*. Rather than idle waiting, the combiner refreshes the local replica from the log, though it might need to refresh again after finally adding the batch to the log. Per-node replicas are shown in Figure 3.

5.3 Read-only operations

Threads performing read-only operations (*readers*) do not reserve space in the log, because their operations do not affect the other replicas. Moreover, a reader that is updating from the log can return and proceed with the read if it encounters an empty entry. Unlike flat combining, NR optimizes read-only operations by executing them directly on the local replica using a readers-writer lock for each node. The combiner acquires the lock in write mode when it wishes to modify the local replica, while *reader* threads acquire the lock in read mode. To avoid stale reads that violate linearizability, a reader must ensure the local replica is fresh. However, the replica need not reflect all operations up to *logTail*, only to the last operation that had completed before the reader started. To do this, we keep a *completedTail* variable, which is an index $\leq$ *logTail* that points to a log entry after which there are no completed operations. After a combiner refreshes its local replica, it updates *completedTail* using a CAS to its last batch entry if it is smaller. A reader reads *completedTail* when it starts, storing it in a local variable *readTail*. If the reader sees that a combiner exists, it just waits until *localTail* $\geq$ *readTail*; otherwise, the reader acquires the

[1] We call *slots* the locations where threads post operations for the combiners; we call *entries* the locations in the shared log.

Algorithm 1 NR Algorithm

```
 1: SHARED: sharedLog, logTail, completedTail
 2: LOCAL (PER NODE): replica, localTail, combinerLock, rwLock
 3: LOCAL (PER CORE): slot, response
 4: function UPDATEFROMLOG(to)
 5:     for index: localTail → to do
 6:         while sharedLog[index] = ⊥ do
 7:             Wait()
 8:         replica.Execute(sharedLog[index])
 9: function RESERVELOGENTRIES(N)
10:     val ← logTail
11:     Repeat CAS(logTail → val + N) until SUCCESS
12:     return val
13: function APPENDTOLOG(N, batch)
14:     k ← 0; start ← ReserveLogEntries(N)
15:     for index: start → start + N do
16:         sharedLog[index] ← (batch[k].slot.args, batch[k++].slot.op)
17:     return start
18: function COMBINE(op, args)
19:     self().response ← ⊥; self().slot ← (args, op)
20:     while TRUE do
21:         if combinerLock.TryLock() then
22:             // combiner
23:             for all threads t on node with t.slot.op ≠ ⊥ do
24:                 batch.Add(t)
25:             startEntry ← AppendToLog(size(batch), batch)
26:             rwLock.Acquire-Writer()
27:             UpdateFromLog(startEntry)
28:             endEntry ← startEntry + size(batch)
29:             localTail ← endEntry
30:             Repeat CAS(completedTail → endEntry) until
31:               (SUCCESS or (endEntry < completedTail))
32:             for t in batch do
33:                 t.response ← replica.Execute(t.slot.op, t.slot.args)
34:             rwLock.Release-Writer(); combinerLock.Release()
35:             return self().response
36:         else
37:             // not combiner
38:             while self().response=⊥ and combinerLock do
39:                 Wait()
40:             if self().response ≠ ⊥ then
41:                 return self().response
42: function READONLY(args)
43:     readTail ← sharedLog.completedTail
44:     while localTail < readTail do
45:         // reader might acquire writer lock and update
46:         WaitOrUpdate(readTail)
47:     rwLock.Acquire-Reader()
48:     response ← replica.ReadOnly(args)
49:     rwLock.Release-Reader()
50:     return response
51: function EXECUTECONCURRENT(op, args)
52:     if replica.IsReadOnly(op) then return ReadOnly(op, args)
53:     return Combine(op, args)
```

readers-writer lock in writer mode and refreshes the replica itself.

5.4 Readers-combiner parallelism

NR's algorithm is designed for readers to execute in parallel with combiners in the same node. In early versions of the algorithm, the combiner lock also protected the local replica against readers, but this prevented the desired parallelism. By separating the combiner lock and the readers-writer lock (§5.3), readers can access the replica while a combiner is reading the slots or writing the log, before it refreshes the replica. Furthermore, to enable parallelism, readers must wait for *completedTail* as described, not *logTail* because otherwise readers block on the hole created by the local combiner, despite the readers lock being available. The pseudo-code for NR is shown in Algorithm 1.

5.5 Better readers-writer lock

The distributed readers-writer lock of [2] uses a per-reader lock to reduce reader overhead; the writer must acquire the locks from all readers. We modify this algorithm to reduce writer overhead as well, by adding an additional writer lock. To enter the critical section, the writer must acquire the writer lock and wait for all the readers locks to be released, without acquiring them; to exit, it releases its lock. A reader waits if the writer lock is taken, then acquires its local lock, and checks the writer lock again; if this lock is taken, the reader releases its local lock and restarts; otherwise, it enters the critical section; to exit, it releases the local lock. With this scheme, the writer and readers incur just one atomic write each on distinct cache lines to enter the critical section. Readers may starve if writers keep coming, but this is unlikely with NR, as often only one thread wishes to be a writer at a time (the combiner) and that thread has significant work outside the critical section. We omit a proof of correctness for brevity.

5.6 Recycling log entries

Each log entry has a bit that alternates when the log wraps around to indicate empty entries. An index *logMin* stores the last known safe location to write; for efficiency, this index is updated only when a thread reaches a *low mark* in the log, which is *max_batch* entries before *logMin*. The thread that reserves the low mark entry updates *logMin* to the smallest *localTail* of all nodes; meanwhile, other threads wait for *logMin* to change. This scheme is efficient: it incurs no synchronization and reads *localTail* rarely if the log is large. A drawback is that a slow thread becomes a bottleneck if it does not update its *localTail*. This problem is avoided using a larger log size.

5.7 Architecture-specific optimizations

NR has many low-level optimizations specific to each architecture and related to the details of the cache coherency protocol. These optimizations are restricted to NR's implementation, and are not required for the sequential data structure implementation, relieving the programmer from the burden of platform-aware optimizations.

Replicas and node-local data are allocated from the node's local memory; data is padded and cache aligned to avoid false sharing. Threads often communicate without atomic instructions but with carefully placed barriers. To atomically write an operation and its arguments in a combiner slot, the combiner uses no locks; rather, it writes in a certain order—argument before operation—so that an operation is never without its arguments. The combiner needs to write a bit to the slot (§5.2), but again no lock is needed, as the combiner writes the bit as part of the operation. The slot is padded to fill the cache line and the operation comes after the arguments to match the order in which a thread writes the slot. The thread waits for the response on a different cache line. A similar interaction occurs when the combiner writes the response.

Across nodes, a combiner uses stores to write to the log; again, the store order and data placement are important. The cadence of the algorithm ensures that log cache lines do not often ping pong across nodes as a combiner typically writes a full cache line before others attempt to read it.

6. Practical considerations

We now discuss some important considerations of how to apply our algorithms to practice. Where appropriate, we describe our experience with Redis.

Threads and cores. The basic NR algorithm assumes that software threads correspond to hardware cores. However, we can adapt NR to allow more threads than cores, as follows (not implemented). Each thread has a combiner slot, or threads share slots using CAS to insert requests. When cores wait for the local combiner, rather than spinning they can run threads to generate larger combiner batches to increase efficiency.

Log length. NR uses a circular array for its log; if the array gets full, threads pause until older entries are consumed. This is undesirable, so one should use a large log, but how large? A solution is to dynamically resize the log if it gets full. This can be done by writing a special log entry that indicates that the log has grown so that all replicas agree on the new size after consuming the special entry. This does not prevent the initial blocking that occurs when the log gets full, but it gradually adjusts the log size until it is sufficiently large.

Memory allocation. As memory allocation can become a bottleneck, we need an efficient allocator that (1) avoids too much coordination across threads [3], and (2) allocates memory local to each node. We use a simple allocator in which threads get buffers from local pools [4]. The allocator incurs coordination only if a buffer is allocated in one thread and freed in another; this requires returning the buffer to the allocating thread's pool. This is done in batches to reduce coordination.

Inactive replica. If a replica is inactive (threads in the node execute no operation), it will stop replaying entries from the log, causing the log to fill up. This problem can be solved by periodically refreshing each replica using a dedicated combiner per node (§4). But many applications do not face this issue, as work is spread over threads (e.g., using a thread pool as we did in our multi-threaded version of Redis).

Coupled data structures. In many applications, data structures are read or updated together. For example, Redis implements sorted sets using a hash table and a skip list, which are updated atomically by each request. NR can provide these atomic updates, by treating the data structures as a single larger data structure with combined operations. (Lock- and wait-free algorithms cannot fundamentally support that.)

Fake update operations. Some update operations become read-only during execution (e.g., remove of a non-existent key). Black-box methods must know about read-only operations at invocation time. If updates become read-only often, one can first attempt to execute them as read-only and, if not possible, then execute them as updates (e.g., remove(key) first tries to look up the key). This requires a simple wrapper around remove(). We did not implement this.

7. Implementation

We implemented NR in C++ for Linux and Windows, with fewer than 1500 lines of code. We found that in practice fixing the size of the log to 1M entries works well, so we did not implement dynamic resizing of the log. We also did not use the dedicated combiner optimization (§4).

Redis. We implemented a multi-threaded version of the Windows port of Redis, using different methods for the concurrent data structures. We added to Redis a thread pool and work queuing, support for per-key locking, and synchronization mechanisms to ensure correct concurrent operation between worker threads and Redis's event-processing thread. Redis can resize its in-memory hash table, which can cause a read-only operation to update data; we disabled this for read-only operations but kept it for update operations. Redis keeps several global counters for statistics (memory consumption, number of operations per type, etc). They impaired multi-threaded performance, so we replaced them with thread-local counters. Altogether, these changes amounted to 1400 lines of code, most of which for making Redis multi-threaded[2]; it took only 20 lines of code to convert each Redis data structure to a concurrent one using NR.

8. Evaluation

We wish to answer five broad questions: How does NR scale with the number of cores for different data structures and workloads? How does NR compare with other concurrent data structures? What is the benefit of NR to real applica-

[2] Note that our black-box approach produces concurrent data structures from sequential ones, not entire multi-threaded applications from single-threaded ones (§2).

Baseline	Description
SL	One big lock (spinlock)
RWL	One big readers-writer lock
FC	Flat combining
FC+	Flat combining with readers-writer lock
LF	Lock-free algorithm
NA	NUMA-aware algorithm

Figure 4. Other methods for comparison (baselines).

tions? How does NR behave on different NUMA architectures? What are the benefits of NR's techniques? What are the costs of NR? To answer these questions, we perform five classes of experiments:

- *Real data structures (§8.1).* We run micro-benchmarks on real data structures: a pairing-heap priority queue [26], a skip list priority queue, a skip list dictionary, and a stack.
- *Synthetic data structure (§8.2).* We run micro-benchmarks on a synthetic data structure (a simple buffer) with a contended workload, to study the effects of data structure characteristics (data size, operation size, etc).
- *Real application (§8.3).* We run macro-benchmarks on the data structures of a real application: the Redis storage server modified to use many threads.
- *Another* NUMA *architecture (§8.4).* We run the same Redis macro-benchmarks on another NUMA architecture to see if there are qualitative differences.
- *Benefits of techniques (§8.5).* We disable individual ideas in NR (§5) and measure the impact.

We compare NR against the other methods (baselines) shown in Figure 4. SL and RWL are methods often used in practice. For RWL we use the same readers-writer lock as NR §5.5. FC can be used as a black-box method[3], while FC+ improves on this method using a readers-writer lock to execute read-only operations efficiently. LF and NA are available only for some of the real data structures; for the synthetic structure, no lock-free or NUMA-aware algorithms exist, and in the real application (Redis), threads must atomically update multiple data structures but lock-free and existing NUMA-aware algorithms do not support that. LF requires a mechanism to garbage collect memory, such as hazard pointers [35, 51] or epoch reclamation [25]; these mechanisms can reduce performance by 5x [13]. We do not use these mechanisms, so the reported numbers for LF are optimistic.

In all experiments, we pin threads to cores. We first use all threads within a node[4], including hyperthreads; as we add more threads, we use threads of more nodes.

Summary of results. On the real data structures (§8.1), we find that NR outperforms other methods at many threads under high operation contention, with the exception of NUMA-aware algorithms tailored to the data structure. The other methods, including lock-free algorithms, tend to lose significant performance beyond a NUMA node. On the synthetic data structure (§8.2), we see that these results hold even as we vary the amount of data accessed per operation and data structure size. Under low operation contention, LF outperforms NR significantly, revealing a limitation of NR. Also, NR consumes more memory than other methods. On a real application's data structures (§8.3), NR outperforms alternatives by 2.6x–14x on workloads with 10% updates, or by 1.1x–4.4x on 100% updates. Finally, the techniques in NR contribute significantly to its success (§8.5).

Testbed. We use a Dell server with 512 GB RAM and 56 cores on four Intel Xeon E7-4850v3 processors at 2.2 GHz. Each processor is a NUMA node with 14 cores, a 35 MB shared L3 cache, and a private L2/L1 cache of size 256 KB/64 KB per core. Each core has 2 hyperthreads for a total of 112 hyperthreads. Cache lines have 64 bytes. We also run an experiment with another NUMA architecture, from AMD, which we detail in the relevant section.

8.1 Real data structures

These experiments use four real data structures: a skip list priority queue, a pairing heap priority queue, a skip list dictionary, and a stack. A priority queue provides two update operations and one read-only operation: *insert(i)* inserts element *i*, *deleteMin()* removes and returns the smallest element, and *findMin()* returns the smallest element without removing it. Priority queues can be based on different mechanisms; we consider two of them: a skip list and a pairing heap. A dictionary provides operations to insert, lookup, and remove elements, and we use a skip list to provide the dictionary. A stack provides operations to push and pop elements. NR, FC and FC+ use the same black-box sequential pairing heap [26], and skip list [54]. For the skip list priority queue and for the stack, FC is not black-box: we use the FC implementation from [30], which batches multiple operations.

For the lock-free baseline (LF), we use the skip-list-based priority queue and skip list dictionary from [37], and the stack from [61]; there are no lock-free implementations for pairing heaps, so we omit LF in that case. For the NUMA-aware baseline (NA), only the stack has a NUMA-aware implementation [17], so we omit NA for the others. The table below summarizes the LF and NA baselines we use:

Data structure	LF	NA
Skip list priority queue	[37]	—
Pairing heap priority queue	—	—
Skip list dictionary	[37]	—
Stack	[61]	[17]

We use the benchmark from the flat combining paper [30], which runs a mix of generic *add*, *remove*, and *read* operations. We map these operations to each data structure as shown below (pq stands for priority queue and dict for dictionary):

[3] FC can also use data-structure-specific optimizations to combine operations—a non-black-box optimization that we enable for FC.

[4] We tried different pinning policies, but they were inferior for all methods.

generic	skip list pq	pairing heap pq	skip list dict	stack
add	*insert*(rnd, v)	*insert*(rnd, v)	*insert*(rnd, v)	*push*(v)
remove	*deleteMin*()	*deleteMin*()	*delete*(rnd)	*pop*()
read	*findMin*()	*findMin*()	*lookup*(rnd)	—

where *rnd* indicates a key chosen at random and v is an arbitrary value. The stack does not have a read operation. The *add* and *remove* operations are the update operations, and we use the same ratio of *add* and *remove* to keep the data structure size nearly constant over time. For data structures other than the stack, we consider three ratios of update-to-read operations: 0%,10%,100% updates (100%,90%,0% reads). For the stack, all operations are updates. Between operations the benchmark optionally does work by writing e random locations external to the data structure. This work causes cache pollution and reduces the arrival rate of operations. We first populate the data structure with 200 000 items, and then measure the performance of the methods for various workload mixes. In each experiment, we fix a method, a ratio of update-to-read operations, an external work amount e, and a number of threads.

8.1.1 Skip list priority queue

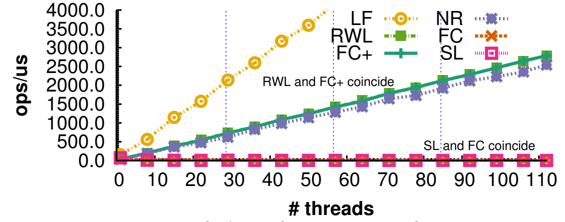
Figure 5 shows the results. (a) For 0% updates, LF, FC+, RWL, and NR scale well, where LF dominates others by ≈2.9x at max threads. LF incurs no locking overheads; FC+, RWL, and NR use readers locks, which are uncontended but require a barrier—a high overhead since the *findMin* read operation is short. NR is ≈9% lower than FC+ and RWL because of NR's overheads of checking the log—which are small but significant relative to *findMin*'s cost. FC and SL do not scale because their reads execute serially.

(b) For 10% updates, all methods drop in performance at the NUMA node boundaries due to the cross-node overheads; but NR drops little, making it the best after 1 NUMA node. At max threads, NR is better than LF, FC+, FC, RWL, SL by 1.7x, 6x, 7x, 27x, 41x. Checking the CPU performance counters, NR had the fewest L3 cache misses and L3 cache misses served from remote caches, indicative of lower cross-node traffic.

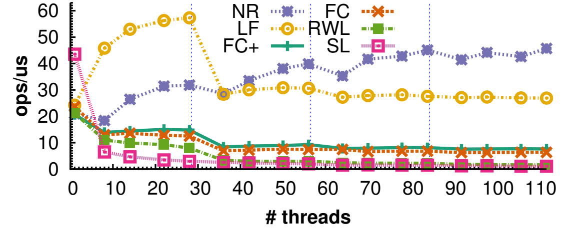
(c) For 100% updates, LF loses its advantage due to higher operation contention: even within a NUMA node, NR is close to LF. After one node, NR is best as before. At max threads, NR is better than LF, FC+, FC, SL, RWL by 2.4x, 2.5x, 3.3x, 8x, 9.4x. In some methods, one thread outperforms many threads, but not when there is work outside the data structure, as is common in a real application, shown in (d). Moreover, we need more threads to scale the application and we want the shared data structure to not become a bottleneck.

(e) We see that NR remains the best method even as we vary the amount of external work e and cache pollution. With e=512, NR is better than FC+, LF, FC, SL, RWL by 1.7x, 1.8x, 2.8x, 12.6x, 16.9x.

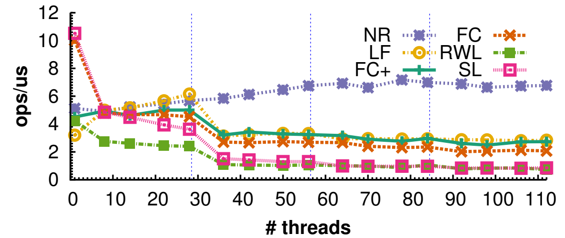
(f) The cost of NR is that it consumes more memory, namely, 148 MB of memory at 112 threads (4.4x the other methods): 12 MB for the log and 34 MB for each of the four replicas. Technically, NR has another cost: it executes an operation many times, one per replica. However, this cost is relatively small as NR makes up for it with better overall performance.



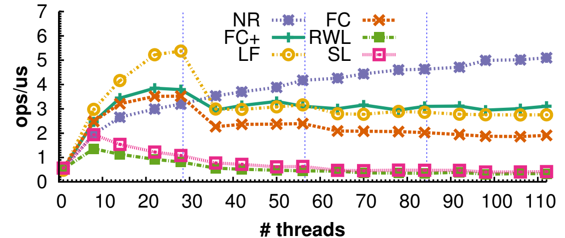
(a) 0% update rate, e=0



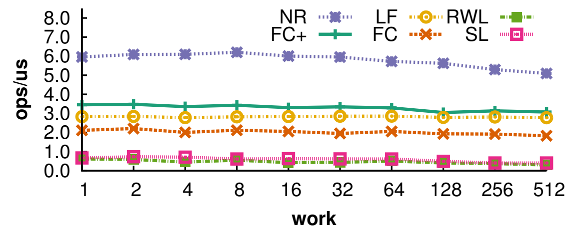
(b) 10% update rate, e=0



(c) 100% update rate, e=0



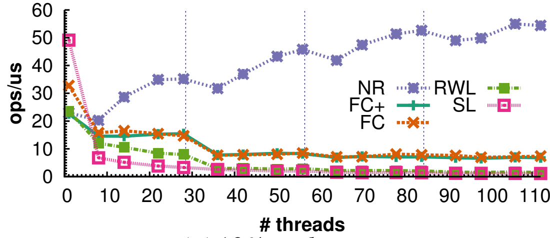
(d) 100% update rate, e=512



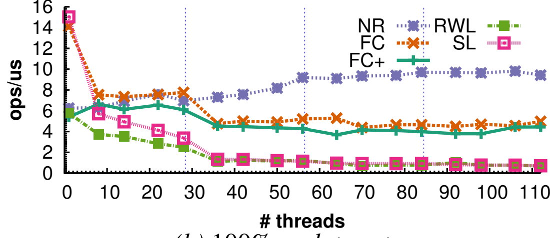
(e) 100% update rate, max threads

(f)	NR	others
memory at max threads (MB)	148	34

Figure 5. Skip list priority queue made concurrent using different methods. Vertical lines show the boundaries between NUMA nodes.



(a) 10% *update rate*



(b) 100% *update rate*

(c)

	NR	others
memory at max threads (MB)	44	8

Figure 6. Performance of pairing heap priority queue. (0% update rate is omitted since it is nearly identical to Figure 5).

8.1.2 Pairing heap priority queue

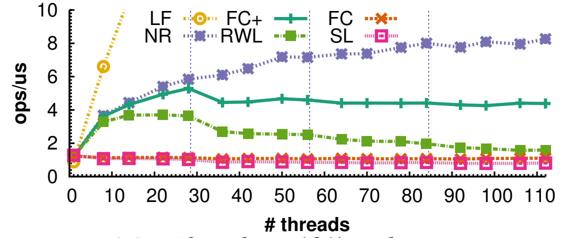
Figure 6 shows the results. We show only e=0 due to space limitations, but the results are similar to the ones for the skip list priority queue (§8.1.1). NR, FC, and FC+ perform slightly better than in the skip list priority queue because the sequential data structure is more efficient, while RWL and SL remain bottlenecked by the lock. As before, NR outperforms significantly once threads grow beyond the first node, and the cost is additional memory consumption.

8.1.3 Skip list dictionary

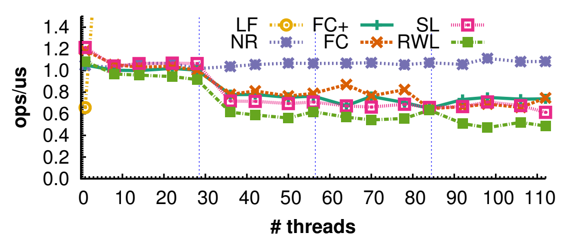
To vary the operation contention, we pick keys using two distributions: uniform (low contention) and zipf with parameter 1.5 (high contention).

The results are shown in Figure 7. As before, for 0% updates, NR, LF, FC+, and RWL scale well, while other methods do not (not shown). When there are updates, performance depends on the level of contention. With low contention (uniform keys), LF outperforms other methods (it is off the charts): at maximum threads, it is 7x and 14x better than NR for 10% and 100% updates, respectively. This is due to the parallelism of the skip list unhindered by contention. Excluding LF, NR outperforms the other methods (with 100% updates, it does so after threads grow beyond a node).

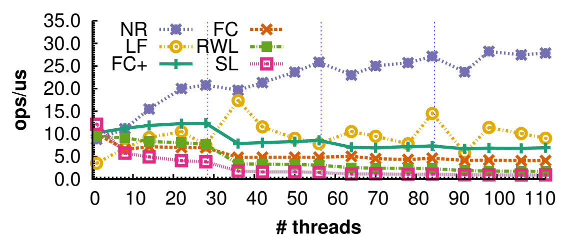
However, with high contention (zipf keys), LF loses its benefit, becoming the worst method for 100% updates. There is a high probability of collisions in the vicinity of the hot keys and the skip list starts to suffer from many failed CASs: with uniform keys, the skip list has $\approx$300K failed CASs, but with the zipf keys this number increases to $>$7M.



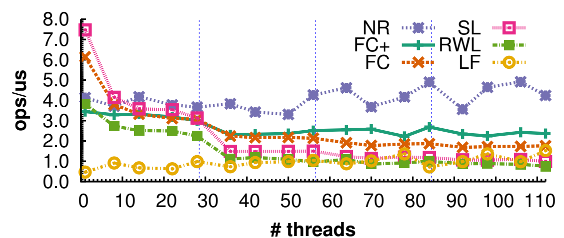
(a) uniform keys, 10% *update rate*



(b) uniform keys, 100% *update rate*



(c) zipf keys, 10% *update rate*



(d) zipf keys, 100% *update rate*

(e)

	NR	others
memory at max threads (MB)	148	34

Figure 7. Performance of skip list dictionary.

NR is the best method after 8 threads. Contention in the data structure does not disrupt the NR log. On the contrary, data structure contention improves cache locality with NR. With maximum threads and 10% updates, NR is better than LF, FC+, FC, RWL, SL by 3.1x, 4.0x, 6.8x, 16x, 30x. With 100% updates, NR is better by 2.8x, 1.8x, 2.4x, 5.7x, 4.3x.

8.1.4 Stack

Figure 8 shows the results. NA and NR scale well, while the other methods do not. At max threads, NR is better than FC, FC+, LF, SL, RWL by 2.3x, 3.0x, 6.2x, 21x, 24x. Note that LF performs poorly; this is consistent with the results in Sections 8.1.1 and 8.1.3, since a stack has significant operation contention.

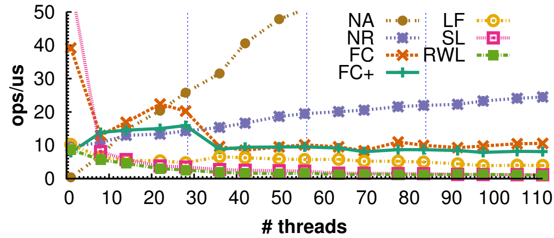


Figure 8. Performance of stack.

NA outperforms NR at 14 threads or more, getting up to 3.6x better at max threads (off the chart). NA achieves that using a stack-specific technique: within a NUMA-node, it uses elimination [32] to match up concurrent pushes and pops, so that they execute without global synchronization. This shows that NUMA-optimizations tailored to the data structure can be effective, and that black-box methods can be limited by their generality.

8.2 Synthetic data structures

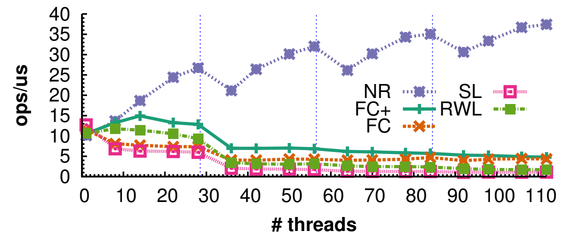
For these experiments, we use a synthetic data structure with parameters that we vary to represent different types of data structure and workload: how large is the data structure (n cache lines), how much data is manipulated per operation (c cache lines) and what fraction of operations are updates (fraction u of operations). More precisely, the synthetic data structure is a buffer with n entries, where each entry takes one cache line. We place a spare cache line between entries to avoid performance artifacts from data prefetching. Each operation accesses c entries by either reading them (read operation) or by reading and writing them (update operation). To produce contention, one of the n entries is accessed by every operation; this is intended to model the tail pointers of stacks, roots of trees, head nodes of skip lists, etc. The other entries are accessed uniformly at random. Each thread runs in a closed loop where it continually issues an update or read operations randomly according to the fraction u.

8.2.1 Scalability

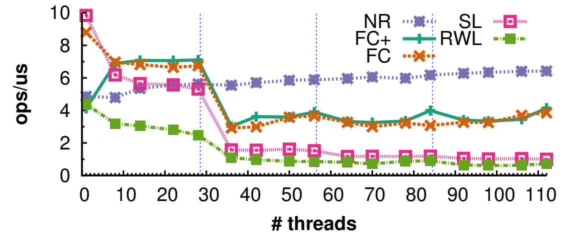
We study the performance of each method as we increase the number of threads. In each experiment, we fix a method, an update rate, and a number of threads, and we measure the aggregate throughput across all threads. We consider three update rates: $u = 0, 10\%, 100\%$. We pick $n = 200K$ and $c = 8$; we later study the effect of these parameters.

Figure 9 shows the results. With 0% updates, FC+, RWL, and NR scale well, with NR lower than FC+ and RWL by 10% and 6%, while FC and SL do not scale due to lock contention (the graph is omitted as it is similar to the 0% update graph in Figure 5 without the LF line).

When the system has operation contention (10% and 100% updates), NR stands out. The other methods lose performance at node boundaries, because they are not designed for NUMA. With u=10%, NR has a bigger advantage because



(a) 10% update rate



(b) 100% update rate

Figure 9. Scalability results for synthetic data structure.

the read-only operations benefit from local replicas significantly. With u=100%, all methods are subject to high operation contention but NR more effectively deals with NUMA issues. We also see that FC+ and FC are better than SL and RWL because the batching in flat combining reduces the synchronization overheads. Moreover, our implementation of flat combining performs operations in the order of NUMA nodes, further decreasing NUMA cache traffic.

8.2.2 Effect of data accessed per operation

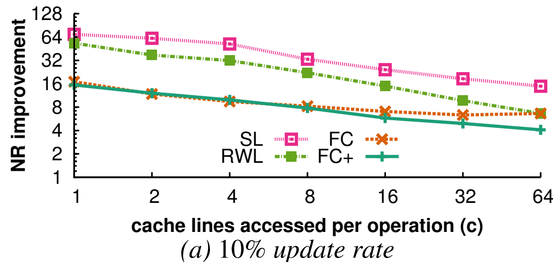
We study the effect of the amount of data accessed by an operation (parameter c). In each experiment, we fix a method, an update rate, and parameter c; we measure the aggregate throughput when the system runs the maximum number of threads. We set $n = 200K$ and consider two update rates: $u = 10\%, 100\%$.

Figure 10 shows the results. We see NR is always better than the other methods, but this advantage decreases as we increase c. This is because of two effects: (1) as we increase c, the operation overhead grows while the synchronization overhead remains the same, thereby decreasing the advantage of any faster method, and (2) NR replicates the operation 4 times, and so a larger c increases the work of NR more than the work of other methods.

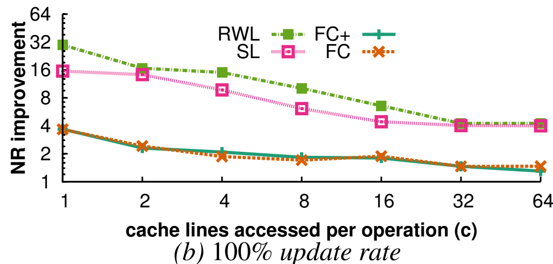
8.2.3 Effect of data structure size

We measure performance for various data structure sizes as we vary n. We experiment with extreme settings for c (1 and 64), vary the mix of operations (10% and 100% update), and set the number of threads to the maximum. In each experiment, we fix these parameters and measure the aggregate throughput.

We find that data structure size does not affect performance much when the data fits in the L3 cache (we omit

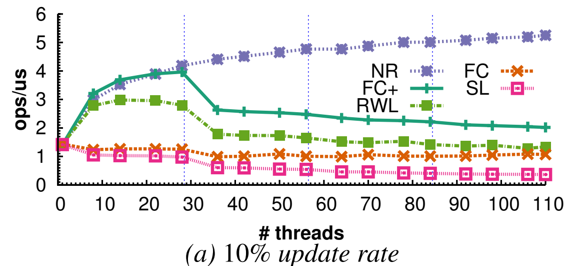


(a) 10% update rate

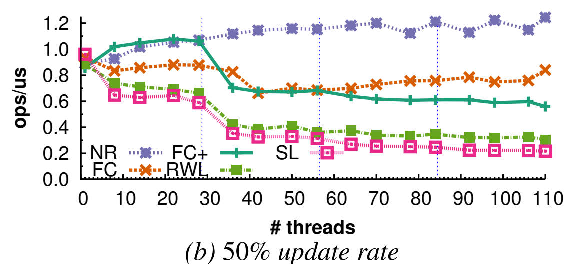


(b) 100% update rate

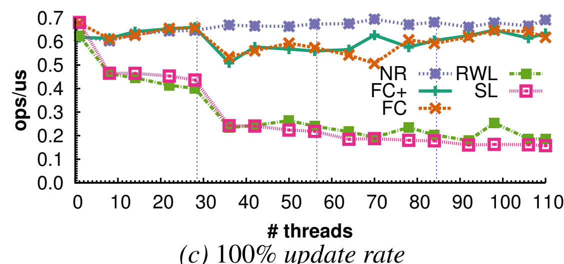
Figure 10. Effect of number of cache lines per operation on the benefit provided by NR using 112 threads. The y-axis indicate the speed-up of NR over each of the other methods.



(a) 10% update rate



(b) 50% update rate



(c) 100% update rate

Figure 11. Redis performance.

the graph due to space limitations). For instance, with 100% updates, for any method, the throughput difference between $n{=}2K$ and $n{=}20K$ is at most 8%. Once outside L3, NR, FC, and FC+ drop by ≈50%, and NR remains the best method.

8.3 Real application

We now consider the data structures of the Redis server, made concurrent using various black-box methods, including NR.

We evaluate Redis's sorted sets, which sort items based on a score. In Redis, sorted sets use a hash table (for fast lookup) and a skip list (for fast rank/range queries). For read operations, we use the ZRANK command, which returns the rank of an item in the sorted order. ZRANK finds the item in the hash table; if present, it finds its rank in the skip list. For update operations we use ZINCRBY, which increases the score of an item by a chosen value. ZINCRBY finds the item in the hash table; if present, it updates its score, and deletes and reinserts it into the skip list. The hash table and skip list are updated atomically with each request.

We used the redis-benchmark utility provided in the distribution to generate client load. We modified the benchmark to support hybrid read/write workloads using the update-read mix of the YCSB benchmark [18] (0%, 10%, 50% updates) in addition to 100% updates.

To overcome the significant overheads of the Redis RPC and approximate a high-performance RPC [46, 47], we invoke Redis's operations directly at the server after the RPC layer, instead of generating requests from remote clients.

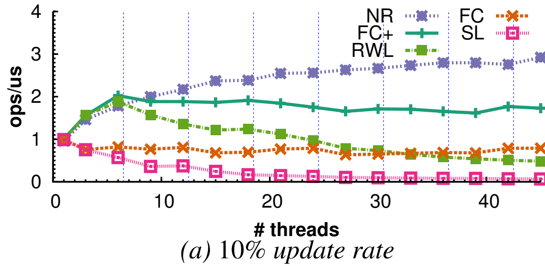
In each experiment, we create a single sorted set with 10,000 items. We launch multiple threads that repeatedly read or update a uniformly distributed random item using ZRANK or ZINCRBY, respectively. In each experiment, we fix an update ratio, a method, and a number of cores, and we measure the aggregate throughput.

Figure 11 shows the results. For 0% updates, RWL, NR and FC+ scale well and have almost identical performance, while FC and SL do not scale (the graph is omitted as it is similar to the previous 0% update graphs). For 10%, 50%, 100% updates, we see that all methods except NR drop after threads grow beyond a single node, making NR the best method for maximum threads. For 10% updates, NR is better than FC+, RWL, FC, SL by 2.6x, 3.9x, 4.9x, 14x, respectively. For 100% updates, NR is better by 1.1x, 3.7x, 1.1x, 4.4x, respectively.

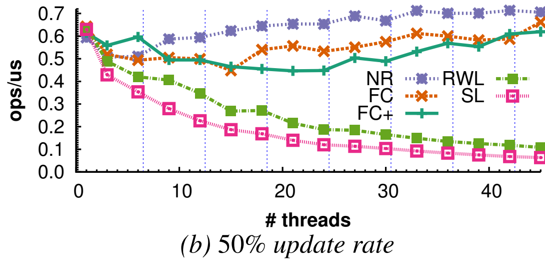
While its scalability is not perfect, NR is the best method here. As discussed, the goal is to reduce data structure bottlenecks so that adding cores benefits the rest of the application.

8.4 Another NUMA architecture

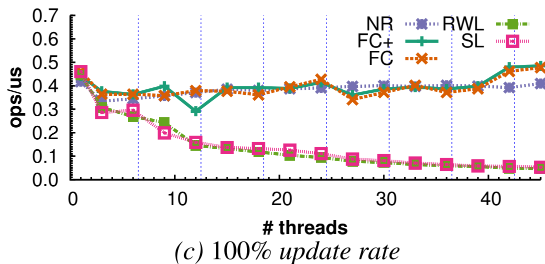
We now study the behavior of NR on another NUMA architecture based on AMD processors, to see how it differs from the results on Intel. We use a server with 120 GB RAM and 48 cores on eight AMD Magny-Cours processors at 1.9 GHz. Each processor has 6 cores, a 10 MB shared L3 cache, and a private L2/L1 cache size of 512 KB/64 KB per core.



(a) 10% update rate



(b) 50% update rate



(c) 100% update rate

Figure 12. Redis experiment on another processor (AMD).

We run the same Redis macro-benchmark as in Section 8.3, but use the AMD machine instead.

Figure 12 shows the results. They are qualitatively similar to those of Section 8.3, with lower absolute numbers, as the AMD machine is older. We also see less of an impact on FC and FC+ as we cross node boundaries; as a result, these methods are competitive with NR with 50% and 100% updates. This happens because the AMD machine uses an incomplete cache directory, which can cause communication across nodes even when data is shared only within a node [21], decreasing the benefit of the NUMA design of NR.

8.5 Benefits of techniques

We study the benefit of the techniques in NR's algorithm, by individually disabling them and observing the impact on different workloads. We use the real data structures benchmark (§8.1) with maximum threads and measure the drop in NR throughput when a technique is disabled. We consider five techniques as shown in Figure 13.

Figure 14 shows the throughput loss of NR as we disable each technique. The results are for the skip list priority queue, but the other structures are qualitatively similar.
We can see that updating the replicas in parallel significantly helps performance across nodes (#4). Within a node, flat combining (#1) and the optimized readers-writer lock (#5) have noticeable impact. In some cases, the techniques come at a cost to update operations (#3 and #5), but the cost is small and bring considerable benefit for read operations.

Technique	How it was disabled
#1. flat combining §5.2	use the readers-writer lock for all synch within node; all threads write to the log
#2. read optimization §5.3,§5.4	readers wait until logTail instead of completedTail
#3. separate replica lock §5.4	combiner lock protects replica
#4. parallel replicas update §5.1	combiners wait on completedTail before getting writer lock[5]
#5. better readers-writer lock §5.5	use standard readers-writer lock

Figure 13. Techniques disabled to understand their benefit.

Workload	#1	#2	#3	#4	#5
10% update	69.8%	34.8%	41.6%	63.7	85.9%
100% update	65.8%	0%	-1.9%	68.9%	-0.9%

Figure 14. Performance loss after disabling each technique.

9. Conclusion

We proposed, implemented, and evaluated a general method to transform sequential data structures into NUMA-aware concurrent data structures. Lock-free data structures are considered state-of-the-art, but they were designed for UMA. Creating new lock-free algorithms for NUMA is a herculean effort, as each data structure requires highly specialized new techniques. NR required such techniques, but once implemented it can be used to implement all concurrent data structures without additional effort. We found that NR performs well for data structures of small and medium sizes, on workloads with operation contention, which are traditionally hard to tackle. Intuitively, NR handles this contention using a hierarchical approach, where most threads synchronize locally within a node, and a few threads synchronize globally across nodes. NR's general approach is to first optimize a shared log for the architecture at hand (NUMA), and then use the log to replicate. We believe this approach could be applicable to future new architectures as well.

Acknowledgements. We are grateful to the anonymous reviewers for the feedback that helped improve the paper.

References

[1] http://redis.io.

[2] Dmitry Vyukov. Distributed Reader-Writer Mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex.

[3] http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[4] M. K. Aguilera. Thread-local malloc. https://github.com/mkaguilera/tmalloc.

[5] In more detail, combiners spin until *completedTail* reaches their batch in the log. This causes the replicas to be updated in series, because *completedTail* is advanced only after the combiner finishes updating its replica.

[5] J. H. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, Dec. 1999.

[6] M. Balakrishnan, D. Malkhi, J. P. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), Dec. 2013.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles*, pages 29–44, Oct. 2009.

[8] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[9] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *USENIX Annual Technical Conference*, Oct. 2011.

[10] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple but effective techniques for NUMA memory management. In *ACM Symposium on Operating Systems Principles*, pages 19–31, Dec. 1989.

[11] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Symposium on Operating Systems Design and Implementation*, pages 43–57, Dec. 2008.

[12] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report TR-2014-019, MIT CSAIL, Sept. 2014.

[13] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 33–42, July 2013.

[14] T. Brown, A. Kogan, Y. Lev, and V. Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 121–132, July 2016.

[15] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, Nov. 1997.

[16] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97, Dec. 2013.

[17] I. Calciu, J. E. Gottschlich, and M. Herlihy. Using delegation and elimination to implement a scalable NUMA-friendly stack. In *USENIX Workshop on Hot Topics in Parallelism*, June 2013.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154, June 2010.

[19] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *ACM Symposium on Operating Systems Principles*, pages 32–44, Dec. 1989.

[20] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–394, Mar. 2013.

[21] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM Symposium on Operating Systems Principles*, pages 33–48, Nov. 2013.

[22] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, Mar. 2015.

[23] T. David, R. Guerraoui, and M. Yabandeh. Consensus inside. In *International Middleware Conference*, pages 145–156, Dec. 2014.

[24] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–334, June 2011.

[25] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[26] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, Jan. 1986.

[27] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, Feb. 1999.

[28] S. K. Haider, W. Hasenplaugh, and D. Alistarh. Lease/release: Architectural support for scaling contended data structures. In *ACM Symposium on Principles and Practice of Parallel Programming*, Mar. 2016. Article 17.

[29] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 417–428, June 2012.

[30] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, June 2010.

[31] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Scalable flat-combining based synchronous queues. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, Sept. 2010.

[32] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, Jan. 2010.

[33] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[34] M. Herlihy and I. Calciu. Work in progress: Shared nothing transactional memory. In *Workshop on Systems for Future Multicore Architectures*, Apr. 2011.

[35] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

[36] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, May 1993.

[37] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[38] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[39] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 295–306, Aug. 2014.

[40] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. In *European Conference on Computer Systems*, pages 133–145, Apr. 2006.

[41] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A memory profiler for NUMA multicore systems. In *USENIX Annual Technical Conference*, pages 53–64, June 2012.

[42] C. Lameter. NUMA (non-uniform memory access): An overview. *ACM Queue*, 11(7), July 2013.

[43] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[44] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data*, pages 743–754, June 2014.

[45] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. NUMA-aware algorithms: the case of data shuffling. In *Conference on Innovative Data Systems Research*, Jan. 2013.

[46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Symposium on Networked Systems Design and Implementation*, pages 429–444, Apr. 2014.

[47] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems*, pages 183–196, Apr. 2012.

[48] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *ACM Symposium on Operating Systems Principles*, pages 168–183, Oct. 2015.

[49] P. E. McKenney and J. D. Slingwine. Read-copy-update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Oct. 1998.

[50] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: a cache-partitioned hash table. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 319–320, Feb. 2012.

[51] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles*, pages 29–41, Oct. 2011.

[53] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware islands. In *International Conference on Data Engineering*, pages 688–699, Mar. 2014.

[54] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[55] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[56] O. Shahmirzadi, T. Ropars, and A. Schiper. High-throughput maps on message-passing manycore architectures: Partitioning versus replication. In *International Conference on Parallel Processing*, pages 536–547, Aug. 2014.

[57] O. Shalev and N. Shavit. Predictive log-synchronization. In *European Conference on Computer Systems*, pages 305–316, Apr. 2006.

[58] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[59] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. DaRPC: Data center RPC. In *ACM Symposium on Cloud Computing*, pages 1–13, Nov. 2014.

[60] H. Sutter. Lock-free code: A false sense of security. *Dr. Dobb's*, Sept. 2008.

[61] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, Apr. 1986.

[62] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Oct. 1996.

[63] L. Xiang and M. L. Scott. Compiler aided manual speculation for high performance concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 47–56, Feb. 2013.