DOI:10.1145/3282506

How to Implement Any Concurrent Data Structure

By Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera

Abstract

We propose a method called Node Replication (NR) to implement any concurrent data structure. The method takes a single-threaded implementation of a data structure and automatically transforms it into a concurrent (thread-safe) implementation. The result is designed to work well with and harness the power of modern servers, which are complex Non-Uniform Memory Access (NUMA) machines with many processor sockets and subtle performance characteristics. Using NR requires no expertise in concurrent data structure design, and the result is free of concurrency bugs. NR represents a paradigm shift of how concurrent algorithms are developed: rather than designing for a data structure, we design for the architecture.

1. INTRODUCTION

Concurrent data structures are 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 the performance of the system.

Due to recent architectural changes, high-performance servers today are Non-Uniform Memory Access (NUMA) machines. Such machines have multiple processor sockets, herein called *nodes*, each with some local cache and memory. Although cores in a node can access the memory in other nodes, it is faster to access local memory and to share cache lines within a node than across nodes. To fully harness the power of NUMA, data structures must take this asymmetry into consideration: they must be NUMA-aware to reduce cross-node communication and minimize accesses to remote caches and memory.

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 (Section 2).

We propose a new technique, called Node Replication (NR), to obtain NUMA-aware data structures, by automatically transforming any single-threaded data structure into a corresponding concurrent (thread-safe) 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.[8]

Node Replication 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[1]). The shared log is realized by a hierarchical, NUMA-aware design that uses flat combining[5] 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.

Node Replication represents a paradigm shift of how concurrent algorithms are designed. Currently, each new concurrent data structure requires its own design, and our community of experts has spent decades writing papers and developing algorithms for all kinds of structures (skip lists, queues, priority queues, and hash tables, etc). However, computer architectures are now fluid with the introduction of new memory features (non-volatility, in-memory processing), new memory models (NUMA, non-coherent caches), new processing elements (GPU, FPGA, TPU), new processor features (transactional memory, SGX), and more. Unfortunately, the old algorithms do not work well in the new architectures, so the community has to redesign the algorithms for each new architecture.

Node Replication shows there is a better way to design algorithms, by using a black-box approach that is independent of the data structure. Thus, rather than designing for a data structure, we design for the architecture. This approach significantly reduces the design effort to a few architectures, instead of the product of the number of architectures and the number of data structures. While we demonstrate the black-box approach for NUMA here, we envision its general applicability to other new architectures as they emerge.

Node Replication cannot always outperform algorithms that specialize for a single data structure and architecture. However, perhaps surprisingly, NR performs well in many cases, particularly when there is contention, where 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. Many systems have shown how servers can scale the handling of network requests and minimize Remote Procedure Calls (RPC) bottlenecks.[10] There is less research on how to scale the servicing of the requests. These

The original version of this paper, titled "*Black-box Concurrent Data Structures for NUMA Architectures*," was published in ASPLOS 2017. For more information, please check https://research.vmware.com/projects/nodereplication.

systems either implement a simple service (e.g., get/put) that can partition requests across cores;[10] or they develop sophisticated concurrent data structures from scratch to support more complex operations,[11] and doing this requires expertise in concurrent algorithms. This is where our black-box approach comes handy: NR provides these concurrent data structures automatically from single-threaded implementations. For Redis, we were able to convert a single-threaded sorted set into a concurrent one with just 20 new lines of wrapper code. The result outperforms data structures obtained from other methods by up to 14x.

Although NR is powerful, easy to use, and efficient, it has three limitations. First, it 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.[2] 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.

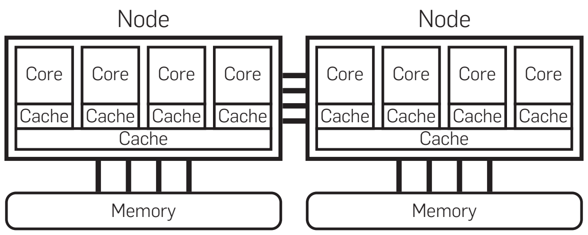
We plan to make the source code for NR available in our project page at https://research.vmware.com/projects/nodereplication.

2. BACKGROUND

2.1. NUMA architectures

Our work is motivated by recent trends in computer architecture. To support a large number of cores, data center servers have adopted a NUMA architecture with many processor sockets or *nodes* (see Figure 1). Each node has many processor cores and a shared cache, while individual cores have private caches. 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 eight processors (nodes), each with 28 cores, while a typical server might have two processors, each with 8–16 cores. AMD and Oracle have similar machines. To best use these cores, we need appropriate concurrent data structures.

Figure 1. NUMA architecture of a modern server in a data center. The server has many processor sockets, herein called nodes. Each node has many processor cores and some 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 cost. Typically, cores have local caches, and cores on a node share a last level cache.



2.2. Concurrent data structures

Concurrent data structures permit many threads to operate on common data using a high-level interface. When a data structure is accessed concurrently by many threads, its semantics are typically defined by a property called linearizability,[8] 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 do not return a result. Operation contention is challenging because operations must observe each other across cores.

Much work has been devoted to designing and implementing efficient concurrent data structures; we provide a broad overview in Calciu, Sen et al.[3] Unfortunately, each data structure requires its own algorithm with novel techniques, which involve considerable work from experts in the field. To get a sense, a new concurrent data structure often leads to a scientific publication just for its algorithm.

Unfortunately, most existing concurrent data structures and techniques are for Uniform Memory Access (UMA), including some prior black-box methods.[5, 6, 16] These algorithms are not sensitive to the asymmetry and limitations of NUMA, which hinders their performance.[9] There are some recent NUMA-aware algorithms,[2, 12, 14] but they cover few data structures. Moreover, these solutions are not applicable when applications compose data structures and wish to modify several of them with a single composed operation (e.g., remove an item from a hash table and a skip list simultaneously). This is the case in the Redis application, which we describe later in the paper.

3. NODE REPLICATION (NR)

Node Replication is a NUMA-aware algorithm for concurrent data structures. Unlike traditional algorithms, which target a specific data structure, NR implements *all* data structures at once. Furthermore, NR is designed to work well under operation contention. Specifically, under update-heavy contended workloads, some algorithms drop performance as we add more cores; in contrast, NR can avoid the drops, so that the parallelizable parts of the application can benefit from more cores without being hindered by the data structures. NR cannot always outperform specialized data structures with tailored optimizations, but it can be competitive in a broad class of workloads.

While NR can provide any concurrent data structures, it does not automatically convert entire single-threaded applications to multiple threads. Applications have a broad interface, unlike data structures, so they are less amenable to black-box methods.

3.1. API

To work with an arbitrary data structure, NR expects a single-threaded implementation of the data structure provided as four generic methods:

$Create() \rightarrow ptr$
$Execute(ptr, op, args) \rightarrow result$
$IsReadOnly(ptr, op) \rightarrow Boolean$
$Destroy()$

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, but we allow nondeterminism inside the operation execution and the data structure (e.g., levels of nodes in a skip list). Similarly, operations can use randomization internally, but results should not be random (results *can* be pseudorandom with a fixed initial seed). The *IsReadOnly* method indicates if an operation is read-only; we use this information for read-only optimizations in NR. The *Destroy* method deallocates the replicas and the log. NR provides a new method *ExecuteConcurrent* that can be called concurrently from different threads.

For example, to implement a hash table, a developer provides a Create method that creates an empty hash table; an Execute method that recognizes three *op* parameters (insert, lookup, remove) with the *args* parameter being a key-value pair or a key; and a IsReadOnly method that returns true for op=lookup and false otherwise. The Execute method implements the three operations of a hash table in a single-threaded setting (not thread-safe). NR then provides a concurrent (thread-safe) implementation of the hash table via a new method *ExecuteConcurrent*. For convenience, the developer may subsequently write three simple wrappers (insert, lookup, remove) that invoke *ExecuteConcurrent* with the appropriate *op* parameter.

3.2. Basic idea

Node Replication replicates the data structure on each NUMA node, so that threads can execute operations on a replica that is local to their node. Replication brings two benefits. First, an operation can access the data structure on memory that is local to the node. Second, operations can execute concurrently across nodes on different replicas. Replication, however, raises the question of how threads coordinate access to the replicas and maintain them in sync.

For efficiency, NR uses different mechanisms to coordinate threads within nodes 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[5]). Flat combining batches operations from multiple threads and then executes the batch using a single thread, called the *combiner*. The combiner is analogous to a leader in distributed systems. In NR, we batch operations from threads in the same node, using one combiner per node. The combiner of a node is responsible for checking if threads within the node have any outstanding update operations, and then it executes all such operations on behalf of the other threads. Which thread is the combiner? The choice is made dynamically among threads within a node that have outstanding operations. The combiner changes over time: it abdicates when it finishes executing the outstanding updates, up to a maximum number. Batching can gather many operations, because there are many threads per node (e.g., 28 in our machine). Batching in NR is advantageous because it localizes synchronization within a node.

Across nodes, threads coordinate through a shared log (a technique from distributed systems[1]). 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.

Node Replication applies an optimization to *read-only* operations (operations that do not change the state of the data structure). Such operations execute without going through the log, by reading directly the local replica. To ensure consistency (linearizability[8]), the operation must ensure that the local replica is fresh: the log must be replayed at least until the last operation that completed before the read started.

We have considered an additional optimization, which dedicates a thread to run the combiner for each node; this thread replays the log proactively. This optimization is sensible for systems that have many threads per node, which is an ongoing trend in processor architecture. However, we have not employed this optimization in the results we present here.

The techniques above provide a number of benefits:

- *Reduce Cross-Node Synchronization and Contention:* NR appends to the log without acquiring locks; instead, it uses the atomic Compare-And-Swap (CAS) instruction on the log tail to reserve new entries in the log. The CAS instruction incurs little cross-node synchronization because only the combiners execute the CAS, and there is at most one combiner per node—hence synchronization required for the CAS involves only a few threads (typically 2–8). In addition, the cost of a CAS is amortized over many operations due to batching.

- *Read and Write to the Log in Parallel:* Combiners can concurrently read the log to update their local replicas. Moreover, combiners can also concurrently write to the log: after combiners have reserved new entries using CAS, combiners can fill their entries concurrently.
- *Read Locally in Parallel:* Read-only operations in the data structure execute against the local replica, and so they can proceed in parallel 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.
- *Use 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. For example, clearing a dictionary might modify many parts of the data structure, but we only communicate the operation description across nodes.

A complication that must be addressed is how to recycle the log. This must be done without much coordination, for performance, but must also ensure that a log entry is recycled only after it has been applied at all the replicas. Roughly speaking, NR uses a lightweight lazy mechanism that reduces synchronization by delegating responsibility of recycling to one of the threads.

In what follows, we describe these ideas in more detail.

3.3. Intra-node coordination: combining

To execute an operation, a thread posts its operation in a reserved slot[a] and tries to become the combiner by acquiring the *combiner lock*. The combiner reads the slots of the threads in the node and forms a batch *B* of operations to execute. The combiner then proceeds to write the operations in *B* to the log, and to update the local replica with the entries from the log.

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 it again after finally adding the batch to the log. Figure 2 depicts the general ideas.

3.4. Inter-node coordination: circular buffer

Node Replication replicates the data structure across nodes using a log realized as a shared circular buffer that stores update operations on the data structure. This buffer can be allocated from the memory of one of the NUMA nodes, or it could be spread across nodes. The log is accessed by at most one thread per node, and it provides coordination and consistency across nodes.

A variable *logTail* contains the index of the next available entry. 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 (Section 3.3).

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

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

Figure 2. NR replicates the data structure across nodes. A shared log stores updates that are later applied to each replica. Here, there are two nodes and hence two replicas of a tree. The replicas are not in sync, because the right replica has incorporated more updates from the shared log. Threads in the same node share the replica in that node; they coordinate access to the replica using a lock and a technique called flat combining (Section 3.3). Flat combining is particularly efficient in UMA systems. Effectively, NR treats each node as a separate UMA system.

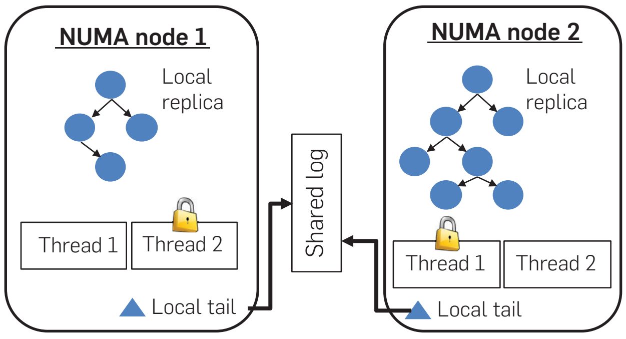
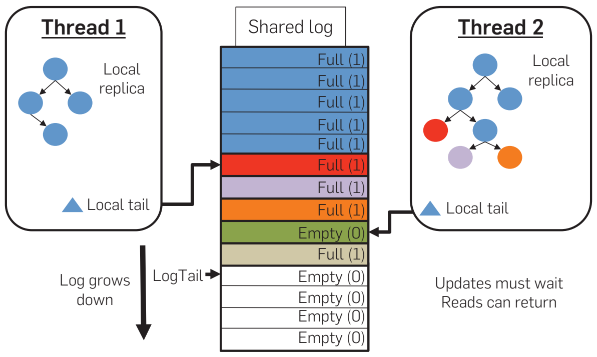


Figure 3. The shared log in NR is realized as a circular buffer, shown here as an array for simplicity. There is a global *log-Tail* variable that indicates the first unreserved entry in the log. Each node has a *localTail* variable that indicates the next operation in the log to be executed on each local replica. The figure shows only one thread for each node—the thread that is currently chosen as the combiner for that node—but there are other threads. Thread 1's replica executed 5 operations from the log. Thread 2's replica executed 3 more operations and found an "empty" 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 (Section 3.6).



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 3 shows two combiners accessing the log to update their local replicas, which they do in parallel.

3.5. 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 node becomes a bottleneck if no thread on that node updates the *localTail*. This problem is avoided using a larger log size.

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

3.7. Readers-combiner parallelism

Node Replication'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 (Section 3.6), 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.

3.8. Better readers-writer lock

Vyukov's distributed readers-writer lock 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.

3.9. Practical considerations

We now discuss some important practical considerations that arised when we implemented NR.

Software and hardware threads. So far, we have assumed that software threads correspond one-to-one with hardware threads, and we have used the term *thread* indistinguishably to refer to either of them. However, in practice applications may have many more software threads than available hardware threads. To handle this situation, we can have more combiner slots than hardware threads, and then assign each software thread to a combiner slot. Beyond a certain number of software threads, they can share combiner slots using CAS to insert requests. When a software thread waits for the local combiner, it yields instead of spinning, so that the underlying hardware thread can run other software threads to generate larger combiner batches and 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? The solution is to dynamically resize the log if it gets full. This is 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 scheme gradually adjusts the log size until it is large enough.

Memory allocation. Memory allocation can become a performance bottleneck. We need an allocator that (1) avoids too much coordination across threads, and (2) allocates memory local to each node. We use a simple allocator in which threads get buffers from local pools. 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 threads in a node execute no operation on the data structure, the replica of that node stops replaying entries from the log, causing the log to fill up. This problem is solved by periodically running a thread per node that refreshes the local replica if the node has no operations to execute.

Coupled data structures. In some 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.

Fake update operations. Some update operations become readonly during execution (e.g., remove of a nonexistent 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.

4. EVALUATION

We have evaluated NR 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 applications? How does NR behave on different NUMA architectures? What are the benefits of NR's techniques? What are the costs of NR? Here, we highlight the most representative results and focus on the first three questions; the complete set of results are available in Calciu, Sen et al.[3] We report on two classes of experiments:

- *Real Data Structures (Section 4.1):* We run microbenchmarks on real data structures: a skip list priority queue and a skip list dictionary.
- *Real Application (Section 4.2):* We run macro-benchmarks on the data structures of a real application: the Redis storage server modified to use many threads.

We compare NR against other methods (baselines) shown in Figure 4. Single Lock (SL) and Readers-Writer Lock (RWL) are methods often used in practice; they work by protecting the data structure with a SL or a single RWL. For RWL we use the same readers-writer lock as NR (Section 3.8). FC consists of flat combining used to implement the entire data structure. FC can be used as a black-box method, but it can also use data-structure-specific optimizations to combine operations for faster execution (hence its name); we use these optimizations whenever possible. FC+ is an improvement of FC by using a readers-writer lock to execute read-only operations more efficiently. Lock-Free Baseline (LF) is a lock-free algorithm specialized for a specific data structure; this baseline is available only for some data structures. In the real application (Redis), threads must atomically update multiple data structures but existing lock-free algorithms do not support that. LF requires a mechanism to garbage collect memory, such as hazard pointers[13] or epoch reclamation;[4] these mechanisms can reduce performance by 5x. We do not use these mechanisms, so the reported numbers for LF are better than in reality.

Figure 4. Other methods for comparison (baselines).

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

Summary of results. On the real data structures (Section 4.1), 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. We also find that NR consumes more memory than other methods. On a real application's data structures (Section 4.2), NR outperforms alternatives by 2.6x–14x on workloads with 10% updates, or by 1.1x–4.4x on 100% updates.

Testbed. We use a Dell server with 512GB RAM and 56 cores on four Intel Xeon E7-4850v3 processors at 2.2GHz. Each processor is a NUMA node with 14 cores, a 35MB shared L3 cache, and a private L2/L1 cache of size 256KB/64KB per core. Each core has 2 hyper-threads for a total of 112 hyper-threads. Cache lines have 64B.

4.1. Real data structures

These experiments use two real data structures: a skip list priority queue and a skip list dictionary. (Additional results using two other data structures are given in Calciu, Sen et al.,[3] but these results are qualitatively similar to the ones we present here.) 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. We implement these operations using a skip list to order the elements and keep the minimum at the beginning of the list. A dictionary provides operations to insert, lookup, and remove elements, and we use a skip list to provide the dictionary. NR, FC, and FC+ internally use the same single-threaded implementation of a skip list;[15] FC uses the flat combining implementation from Hendler et al.[5] For the LF, we use the skip-list-based priority queue and skip list dictionary from Herlihy and Shavit.[7]

We use the benchmark from the flat combining paper,[5] which runs a mix of generic *add*, *remove*, and *read* operations. We map these operations to each data structure as shown here.

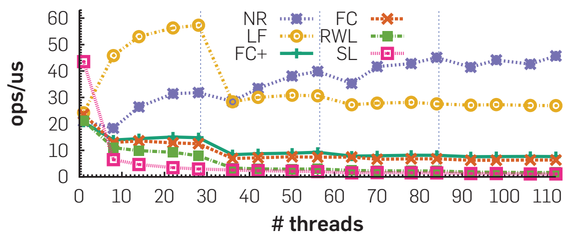
generic	priority queue	dictionary
add	*insert*(*rnd*, *v*)	*insert*(*rnd*, *v*)
remove	*deleteMin*()	*delete*(*rnd*)
read	*findMin*()	*lookup*(*rnd*)

Here, *rnd* indicates a key chosen at random and *v* is an arbitrary value. We use the same ratio of *add* and *remove* to keep the data structure size approximately constant over time, and the results aggregate these two operations as "update operations." We consider two ratios of update-to-read operations: 10%, 100% updates (90%, 0% reads). For the priority queue, we choose random keys from a uniform distribution. For the dictionary, we vary the operation contention by drawing the keys from two distributions: uniform (low contention) and zipf with parameter 1.5 (high contention).

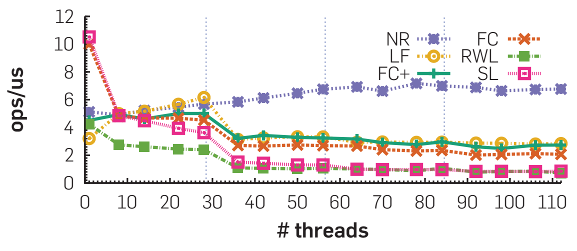
Between operations the benchmark optionally does work by writing *e* random locations external to the data structure. This work causes cache pollution and reduces the arrival rate of operations. We first populate the data structure with 200,000 items, and then measure the performance of the methods for 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.

Results for priority queue. For the priority queue, we see the following results (see Figure 5).

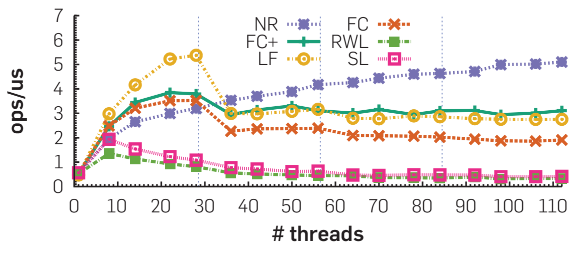
Figure 5. Performance of priority queue made concurrent using different methods. Vertical lines show the boundaries between NUMA nodes.



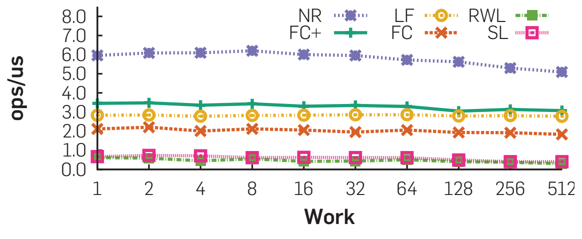
(a) 10% update rate, e=0



(b) 100% update rate, e=0



(c) 100% update rate, e=512



(d) 100% update rate, max threads

(e)

	NR	others
memory at max threads (MB)	148	34

(a) 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 one 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, indicating lower cross-node traffic.

(b) 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.

(c) In some methods, one thread outperforms many threads, but not when there is work outside the data structure, as in many real applications. In such applications, we need more threads to scale the application and we want the shared data structure to not become a bottleneck.

(d) Node Replication 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.

(e) The cost of NR is that it consumes more memory, namely, 148MB of memory at 112 threads (4.4x the other methods): 12MB for the log and 34MB 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.

Results for dictionary. For the dictionary, we see the following results (see Figure 6). 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).

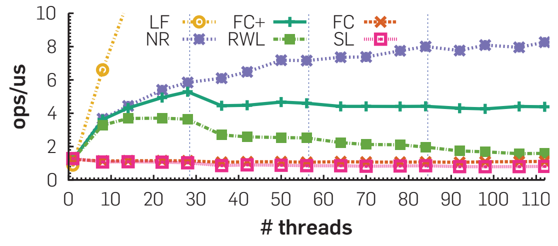
However, with high contention (zipf keys), LF loses its benefit, becoming the worst method for 100% updates. There is a high probability of collisions in the vicinity of the hot keys and the skip list starts to suffer from many failed CASs: with uniform keys, the skip list has ≈300*K* failed CASs, but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

4.2. Redis

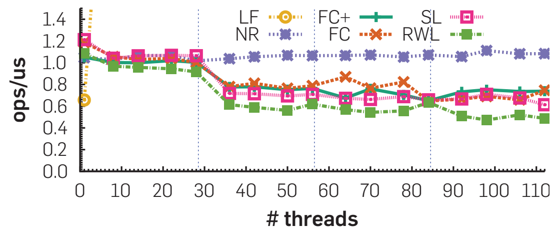
We now consider the data structures of the Redis server, made concurrent using various black-box methods, including NR.

We evaluate the sorted set data structure in Redis, which sorts items based on a score. In Redis, sorted sets are

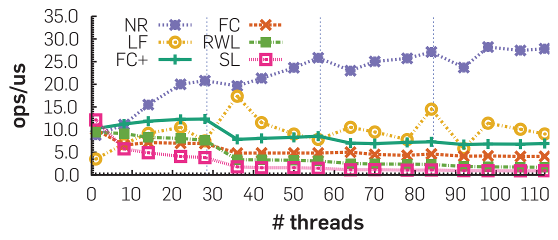
Figure 6. Performance of skip list dictionary.



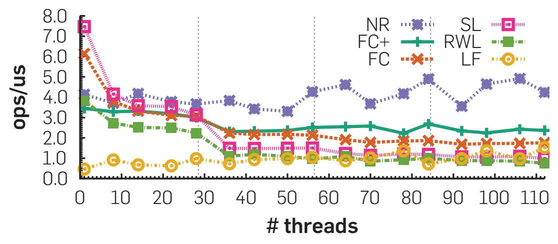
(a) uniform keys, 10% update rate



(b) uniform keys, 100% update rate



(c) zipf keys, 10% update rate

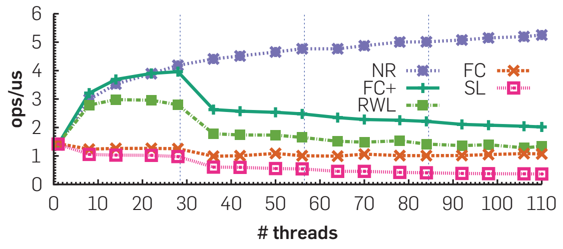


(d) zipf keys, 100% update rate

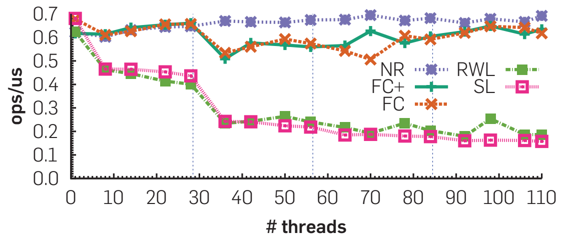
(e)

	NR	others
memory at max threads (MB)	148	34

Figure 7. Performance of Redis application.



(a) 10% update rate



(b) 100% update rate

implemented by a composed data structure that combines a hash table (for fast lookup) and a skip list (for fast rank/range queries). Every element in the sorted set is kept in both hash table and skip list. These underlying data structures must be updated atomically without the possibility that a user observes an update reflected in the hash table without it being reflected in the skip list, and vice versa.

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.

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 (0%, 10% updates) in addition to 100% updates.

To overcome the significant overheads of the Redis RPC and approximate a high-performance RPC,[10] 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.

Results. We see the following results (see Figure 7). For 10% updates, we see that all methods except NR drop after threads grow beyond a single node, making NR the best method for maximum threads. 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. 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).

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

5. CONCLUSION

Node Replication is a general black-box method to transform single-threaded 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 also required comparable effort, but once realized, it can be used to provide all data structures with no extra work. With such a black-box method, we design for the architecture (in this case, NUMA) rather than for a data structure. We believe the community should investigate this black-box approach for future new architectures. **C**

References

1. Balakrishnan, M., Malkhi, D., Davis, J. P., Prabhakaran, V., Wei, M., Wobber, T. CORFU: a distributed shared log. *ACM Trans. Comp. Syst. 31*, 4 (Dec. 2013).
2. Calciu, I., Gottschlich, J.E., Herlihy, M. Using delegation and elimination to implement a scalable NUMA-friendly stack. In *USENIX Workshop on Hot Topics in Parallelism* (June 2013).
3. Calciu, I., Sen, S., Balakrishnan, M., Aguilera, M.K. Black-box concurrent data structures for NUMA architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 2017), 207–221.
4. Fraser, K. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory (Feb. 2004).
5. Hendler, D., Incze, I., Shavit, N., Tzafrir, M. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures* (June 2010), 355–364.
6. Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 11*, 1 (Jan. 1991), 124–149.
7. Herlihy, M., Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
8. Herlihy, M.P., Wing, J.M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.
9. Lameter, C. NUMA (non-uniform memory access): an overview. *ACM Queue 11*, 7 (July 2013).
10. Lim, H., Han, D., Andersen, D.G., Kaminsky, M. MICA: a holistic approach to fast in-memory key-value storage. In *Symposium on Networked Systems Design and Implementation* (Apr. 2014), 429–444.
11. Mao, Y., Kohler, E., Morris, R.T. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems* (Apr. 2012), 183–196.
12. Metreveli, Z., Zeldovich, N., Kaashoek, M.F. CPHash: a cache-partitioned hash table. In *ACM Symposium on Principles and Practice of Parallel Programming* (Feb. 2012), 319–320.
13. Michael, M.M. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst. 15*, 6 (June 2004), 491–504.
14. Porobic, D., Liarou, E., Tözün, P., Ailamaki, A. ATraPos: adaptive transaction processing on hardware islands. In *International Conference on Data Engineering* (Mar. 2014), 688–699.
15. Pugh, W. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM 33*, 6 (June 1990), 668–676.
16. Shalev, O., Shavit, N. Predictive log-synchronization. In *European Conference on Computer Systems* (Apr. 2006), 305–316.

Irina Calciu and Marcos K. Aguilera, VMware Research Group, Palo Alto, CA, USA.

Siddhartha Sen, Microsoft Research, New York, NY, USA.

Mahesh Balakrishnan, Yale University , New Haven, CT, USA.