Brief Announcement: Black-Box Concurrent Data Structures for NUMA Architectures

Irina Calciu[1], Siddhartha Sen[2], Mahesh Balakrishnan[3], and Marcos K. Aguilera[4]

1 **VMware Research, Palo Alto, CA, USA**
2 **Microsoft Research, New York, NY, USA**
3 **Yale University, New Haven, CT, USA**
4 **VMware Research, Palo Alto, CA, USA**

Abstract

Recent work introduced a method to automatically produce concurrent data structures for NUMA architectures. We present a summary of that work.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent data structures, log, NUMA architecture, replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.45

1 Introduction

Modern data centers increasingly employ non-uniform memory access (NUMA) machines with multiple NUMA *nodes*. Each node consists of some (hardware) threads and local memory. A thread can access memory in any node, but accessing local memory is faster than remote memory at another node. To obtain the best performance, concurrent data structures must take this fact into consideration: they must be NUMA-*aware*. Unfortunately, designing concurrent data structures is difficult, and NUMA-awareness makes it harder because the algorithm must try to reduce the number of remote memory accesses.

In this work, we show how to obtain NUMA-aware concurrent data structures *automatically*. We propose an algorithm called Node Replication or NR, which can transform any sequential data structure into a NUMA-aware concurrent data structure. In a nutshell, NR relies on three techniques: replication, an efficient log data structure, and flat combining.

The data structures produced by NR are linearizable [3], thus providing a strong consistency guarantee: an operation appears to take effect instantaneously at some point in time between the operation's invocation and response.

This brief announcement summarizes work published recently [1].

2 The Node Replication algorithm

The NR algorithm takes a sequential data structure and replicates it across NUMA nodes to promote locality of accesses. We use flat combining [2] within each node to ensure safe access to each replica (§2.2). Across nodes, we use a shared log data structure to provide consistency across replicas (§2.1). The log is implemented as a circular buffer allocated from the memory of one of the nodes, providing efficient memory management. Only the combiner within each node accesses the log, ensuring the amount of sharing and contention between nodes is kept to a minimum.

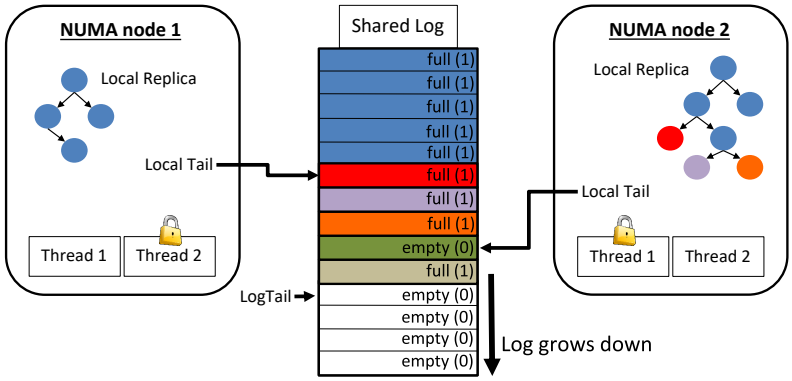

Figure 1 NR algorithm, shared log and per-node replicas. *logTail* indicates the first unreserved entry in the log. Each *localTail* indicates the next operation in the log to be executed on each local replica. Threads on the same node share a replica and coordinate access to the replica using a combiner lock. Threads 2 and 1 are the combiners for nodes 1 and 2, respectively. Node 1's replica executed 5 operations from the log. Node 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, so Thread 1 cannot proceed until the entry is filled. Readers can return when they find an empty entry without waiting (§2.3).

2.1 Sharing across NUMA nodes: the log data structure

A shared log data structure is used to encode the state of the concurrent data structure. This log contains a representation of all the update operations performed on the data structure, giving a total order of these operations. Read-only operations, which do not modify the data structure, are not included in the log.

There are a few important variables that act as indices on the log. First, *logTail* is a global index in the log that points to the next empty entry in the log. Second, each NUMA node has an index into the log, *localTail*, which indicates how far each replica has been updated from the log. When the *localTail*s differ, replicas will be in different states.

A single elected thread on each node, the *combiner*, can access the log on behalf of all the concurrent threads executing at the same time on that node. This thread collects all the other operations and writes them to the log in a batch using a Compare-And-Swap (CAS) instruction to first reserve space in the log, and then using normal stores to write the operations. This batching strategy decreases the amount of contention on the log. Next, the combiner reads the old operations from the log, starting with the entry at *localTail*, and updates the local replica with the operations logged before its own batch. The combiner may find empty entries in the log, identified by a bit in the entry. If so, it must wait until the operation becomes available in the entry. Figure 1 shows two combiners on different NUMA nodes reading from the log to update their local replicas.

The log is implemented as a circular buffer, for efficient memory management. The bit indicating empty entries alternates as the log wraps around. Another index in the log, *logMin*, indicates which entries are safe to write to – the ones that have been applied to all replicas. This index is updated lazily and in a lock-free manner, by the thread that writes to the last available safe entry in the log. This thread checks the values of all *localTail* indices on all NUMA nodes and updates *logMin* to the smallest one. The algorithm could block if a node is slow to update its replica, but in practice this is not a problem if at least one thread on each node accesses the data structure regularly.

2.2 Sharing within a NUMA node: combining

On each NUMA node, safe access to the replica on that node is provided by flat combining [2]. Using this method, threads announce their operations in per-thread slots[1] and then try to acquire a combiner lock. The thread who succeeds becomes the *combiner*: it collects operations from all other threads, writes them to the shared log, and executes them sequentially on the local replica as described above. Only the combiner on each NUMA node accesses the shared log (Figure 1).

2.3 Read-only operations

Flat combining treats update and read-only operations in the same way: the combiner executes all operations sequentially. In contrast, NR optimizes read-only operations, by extending the local replicas with a readers-writer lock that enables the threads performing read-only operations to proceed in parallel. Moreover, read-only operations do not need to be inserted in the log, because they do not need to be executed at all replicas. However, before returning a value read from the local replica, a read-only operation needs to ensure that the replica is fresh so that it does not return a stale value. We use a new index in the shared log, *completedTail*, to indicate all completed operations in the log. The read-only operation needs to read this index as it starts executing and ensure that the replica is updated at least until this index. A thread performing a read-only operation could either wait for a co-located combiner to update the replica, or acquire a writer lock and update the replica itself if no co-located combiner exists.

3 Conclusion

This brief announcement summarized NR, a method to automatically transform sequential data structures into concurrent data structures optimized for NUMA architectures. NR replicates the sequential data structure on each NUMA node, using a shared log to synchronize the replicas across nodes and flat combining to synchronize access to each replica. We implemented and evaluated NR (not shown in this paper) [1]. We found that NR outperforms prior black-box techniques for concurrent data structures and, under high contention, can even perform better than specialized data structures.

References

1 Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221. ACM, 2017. doi:10.1145/3037697.3037721.

2 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, June 2010. doi:10.1145/1810479.1810540.

3 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[1] We refer to the locations used by the combiner as *slots*, and to the locations in the shared log as *entries*.