NUMASK: High Performance Scalable Skip List for NUMA

Henry Daly
Lehigh University, Bethlehem, PA, USA
hwd219@lehigh.edu

Ahmed Hassan
Alexandria University, Alexandria, Egypt
ahmed.hassan@alexu.edu.eg

Michael F. Spear
Lehigh University, Bethlehem, PA, USA
spear@lehigh.edu

Roberto Palmieri
Lehigh University, Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

This paper presents NUMASK, a skip list data structure specifically designed to exploit the characteristics of Non-Uniform Memory Access (NUMA) architectures to improve performance. NUMASK deploys an architecture around a concurrent skip list so that all metadata accesses (e.g., traversals of the skip list index levels) read and write memory blocks allocated in the NUMA zone where the thread is executing. To the best of our knowledge, NUMASK is the first NUMA-aware skip list design that goes beyond merely limiting the performance penalties introduced by NUMA, and leverages the NUMA architecture to outperform state-of-the-art concurrent high-performance implementations. We tested NUMASK on a four-socket server. Its performance scales for both read-intensive and write-intensive workloads (tested up to 160 threads). In write-intensive workload, NUMASK shows speedups over competitors in the range of 2x to 16x.

2012 ACM Subject Classification Information systems Data structures

Keywords and phrases Skip list, NUMA, Concurrent Data Structure

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.18

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

Acknowledgements Authors would like to thank anonymous reviewers for the insightful comments, Maged Michael and Dave Dice for the early feedback on the paper, and Vincent Gramoli for agreeing to integrate NUMASK into Synchrobench.

1 Introduction

Data structures are one of the most fundamental building blocks in modern software. The creation of performance-optimized data structures is a high-value task, both because of intellectual contributions related to algorithms' design and correctness proofs, and because

32nd International Symposium on Distributed Computing (DISC 2018).
Editors: Ulrich Schmid and Josef Widder; Article No. 18; pp. 18:1–18:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of the impact that even a single data structure can have on the performance of enterprise-level applications. For example, the use of a high-performance non-blocking skip list is *the* fundamental innovation in the MemSQL database [29].

Current and (likely) future generations of enterprise-level computing infrastructures deploy on a hardware design known as Non-Uniform Memory Access (or NUMA) [22, 24], which specifies that memory access latency varies depending on the distance between the processor performing the memory access and the memory chip currently holding the memory location. With NUMA, the memory hierarchy is more complex than before; if a system possesses multiple discrete CPU chips (i.e., physical processors installed on different CPU sockets), each will have faster access to a locally-attached coherent memory and slower (but still cache-coherent) access to the memories attached to other chips. This is mainly because the bandwidth of the hardware channel that connects these multiple chips is limited and its performance is generally poor. As a consequence of these considerations, we can claim that NUMA prefers locality; therefore, applications or systems should be (re)designed with this guideline in mind. Such a claim has been confirmed by a number of recent works [27, 4, 6, 10].

The performance penalty of NUMA architectures has been quantified by many recent efforts [4, 26, 3, 16]. A recurring, although conservative, guideline in those studies is to avoid (if possible) scheduling cooperative threads on different processors. Although this guideline is valid in some applications where there is a clear separation in data access pattern among application threads, it might not be easy to apply in other applications where data is maintained as a set of connected items in a linked data structure. For example, searching for an item usually forces a thread to traverse multiple elements of the data structure in order to reach the target item. Because of this, each operation might produce large traffic on the NUMA interconnection; this traffic is the main reason for degraded performance [9].

Caching will not completely solve the problem either, because concurrent updates mandate refreshing cached locations. From our experience, as we show later in the experimental results in Section 7, the presence of even a few percentage of update operations results in a significant performance drop on NUMA. We conclude that data structures not designed for NUMA do not perform well on modern enterprise-level architectures when concurrent updates mandate refreshing cached locations.

In this paper we present NUMASK, a novel concurrent skip list data structure [20] tailored to a NUMA organization. Unlike existing NUMA-aware solutions for data structures (e.g., [6] see Section 2 for details), our design does not limit parallelism to cope with NUMA; rather, it leverages NUMA characteristics to improve performance. What makes our proposal unique is that its advantages hold even for high update rates and contention. We adhered to the following considerations throughout the development of NUMASK:

(a) *local* memory accesses (i.e. memory close to the executing thread's processor) are favored;
(b) traffic across NUMA zones, often produced by synchronization primitives, is avoided.

In a nutshell, our design produces redundant metadata to be placed on different NUMA zones (which meets requirement *(a)*) and avoids the need of synchronizing this metadata across NUMA zones (which satisfies requirement *(b)*). The final design is a data structure that never limits concurrency and at the same time primarily accesses NUMA local memory (in our evaluation study, >80% of memory accesses are local).

The simple observation that motivated our work is that in a skip list, the actual data resides in the lowest level of the skip list, and the other levels form an index layer whose task is only to accelerate execution of operations. In NUMASK, we exploit this fact in two ways:

- We define independent index layers (one per NUMA zone) for the skip list. Each operation traverses the index layer that is local to the thread that executes it. This way, operations do not need to traverse the interconnection between NUMA zones during the index layer traversal. Importantly, we do not keep these index layers consistent with each other; we allow them to be different. In fact, having different index layers in different NUMA zones does not affect correctness because the actual data (which resides in the lowest level of the skip list) is still synchronized.

- We isolate updates on the index layers in separate (per-NUMA) helper threads instead of performing those updates in the critical path of the insert/remove operations. Although this isolation may delay the synchronization of the index layers, the (probabilistic) logarithmic complexity of the skip list operations can be eventually maintained even with lazy index layer updates [18].

Former designs [8, 12] proposed the isolation of index layer updates in helper threads, but none of them defined per-NUMA index layers. That is why in those proposals, the NUMA overhead is still significant due to traversing a single index layer. NUMASK inherits the idea of applying replication to data structure in order to improve its performance in NUMA architectures, as done by [6], but NUMASK targets only metadata and updates such metadata lazely.

We implement NUMASK in C++ and integrate into Sychrobench [17], a comprehensive suite of data structures implemented in the same optimized software infrastructure. The implementation of NUMASK has been enriched with specific optimizations, such as an efficient NUMA memory allocator, developed on top of `libnuma` [1], to avoid bottleneck. We compare the performance of NUMASK with three state-of-the-art concurrent skip lists: Fraser [15], No Hotspot [8], and Rotating Skip List [12]. Performance shows up to 16x speed up for write workloads and improvements up to 40% in read-intensive workloads. In summary, NUMASK hits an important performance goal: in low-contention workloads, NUMASK adds no overhead to the high-performance concurrent data structures; and in high-contention workloads, NUMASK outperforms all other competitors and keeps scaling (we tested up to 160 threads) while other competitors stop earlier (at 64 threads in our experiments).

NUMASK is part of the core release of Synchrobench [17] available at `https://github.com/gramoli/synchrobench`.

2 Related Work

Many concurrent variants of the original sequential skip list [28] data structure have been proposed in the last decade. Some of them are blocking [6, 21, 19, 20], and others are non-blocking [14, 15, 8, 12]. Among the non-blocking designs, which often demonstrate improved performance over blocking designs [17], Fraser [15] proposed the use of a CAS primitive to create a non-blocking skip list. Crain et al. [8] proposed a contention friendly skip list, called No Hotspot, which serves as the foundation of our NUMASK design. The main innovation in No Hotspot is that it isolates bookkeeping operations (e.g., updating index levels) in a helper thread. The rotating skip list was proposed by Dick et al . [12] to further improve No Hotspot's poor locality of references in order to reduce cache misses. However, none of the above designs is optimized for NUMA architectures and thus they all generate significant NUMA interconnect traffic.

Recent uses of skip lists include ordered maps, priority queues, heaps, and database indexes (e.g., [29]). The NUMASK design can be applied to these data structures, improving their performance through data and index layer separation when deployed in NUMA architectures.

The impact of NUMA organization on the performance of software components (e.g. data structures and thread synchronization) is an important topic. Interestingly, the last decade saw the proposal of many NUMA-aware building blocks to improve application performance. Examples include NUMA-aware lock implementations [11, 5], thread placement policy [23], and smart data arrays [27]. Although helpful, the applicability of these components in linked data structures is limited due to the memory organization required by data structures in order to implement their operations while preserving the asymptotic complexity.

Few specialized NUMA-aware techniques for data structures have been proposed [6, 4]. The most relevant to NUMASK is the method proposed by Calciu et al. [6], wherein data structures can be made NUMA aware. Using a technique called `NR` (Node Replication), replicas are created across NUMA zones. However, replica synchronization across zones forces significant NUMA interconnect traffic. In fact, since synchronous updates of the whole data structure (including the searching layer) are assumed, the authors needed a shared synchronization log to save and replay update operations on each replica of the data structure. Moreover, a read operation would wait for the replay of pending updates in order to guarantee its linearization. On the bright side, this approach is a general technique that applies to different data structure designs, whereas NUMASK can exploit specific optimizations because its goal is to provide a high-performance NUMA-aware skip list. In fact, NUMASK relaxes the need of synchronizing different index layer instances; thus, it does not suffer from the above overheads which impede scalability.

Brown et al. [4] proposed a simple design, effective in small-scale deployments, that maintains the entire index layer in a single NUMA zone. This solution's pitfall is its limited parallelism. For operations to access NUMA-local memory addresses, either the application thread's execution must be migrated to the processor attached to the desired NUMA zone, or the operation must be delegated to one or more serving threads in the target NUMA zone. This inherently limits parallelism to a single processor's maximum computing capability. Our new design overcomes all the above limitations: all application and background threads operate primarily on NUMA-local memory and perform a negligible number of NUMA-remote accesses, eliminating the need for migration or delegation.

Orthogonal to our NUMASK approach, in [27, 25] partitioning techniques have been used for targeting the hardware organization of NUMA architectures to improve the performance of array representations [27] and in-memory transaction processing [25].

3 Terminology, NUMA & Linked Data Structures

In NUMA, each (multicore) CPU is physically connected to a partition of the whole memory available in the system, called a NUMA zone. A hardware interconnection exists between NUMA zones (the NUMA interconnection). The hardware provides applications (including the OS) with the abstraction of a single consistent global memory address space; therefore, threads can access the entire memory range in a manner that is oblivious to the NUMA zone in which each virtual address resides. However, this transparency comes with performance costs associated with having an interconnection between NUMA zones.

This interconnection has limited bandwidth, is slow to traverse, and saturates when many threads attempt to use it. Thus, if a thread executing on one CPU accesses a memory location stored in a NUMA zone physically connected with another CPU (called a *remote* NUMA zone hereafter), it incurs a latency that is significantly higher than the latency needed to access a memory location in the NUMA zone connected with the CPU where the thread executes (called *local* NUMA zone hereafter). In short, we use the term NUMA-local memory when the memory is in the local NUMA zone and the term NUMA-remote memory otherwise.

Linked data structures are particularly affected by the memory latency variation introduced by NUMA. This is because traversing the data structure through pointers can easily lead threads to access memory locations physically maintained in remote NUMA zones. NUMA-aware memory allocation (e.g., `libnuma` [1], which is supported by most Operating System distributions) cannot eliminate this problem because even if threads allocate memory in their local NUMA zone, they might still need to traverse many other nodes to accomplish their operation, and these nodes might be added by threads running on remote NUMA zones.

4 NUMASK: A Concurrent Skip List Designed for NUMA

In this section we illustrate the design of NUMASK. In order to retain decades of high performance skip list results, NUMASK deploys a modular design that re-uses the fundamental operations of an existing concurrent skip-list and wraps these operations around a NUMA-aware architecture. The result is a data structure whose performance improves upon the selected concurrent skip list implementation when deployed on NUMA architectures. Another benefit of our modular design is that the correctness of the resulting NUMA-aware skip list is easy to prove since the wrapping architecture does not modify the core operations of the selected concurrent skip list implementation, which is assumed to be correct.

In the rest of the paper we will use the term *base skip list* to indicate an implementation of a skip list that is wrapped (and improved) by the NUMASK architecture. The *base skip list* is a concurrent skip list whose API are *insert*, *remove*, and *contains* operations, with their default signatures [20]. The only requirement we add to this concurrent skip list is that bookkeeping operations (e.g., updating the searching layers and physical removal of logically deleted nodes) are decoupled from the critical path of the data structure operations (i.e., insert/remove/contains) and executed lazily by a *helper* thread. It is worth noting that the features we require in the base skip list have been successfully deployed in many existing data structure implementations [18, 7, 12] and do not diminish the applicability of our proposal.

In this paper we use Crain et al.'s No Hotspot skip list [8] as the *base skip list* because it defines a helper thread responsible for updating the skip list, and it is one of the state-of-the-art concurrent skip list implementations (as studied in [17]). For completeness, it is worth mentioning that No Hotspot, and thus our NUMASK skip list implementation, is lock-free.

All skip list implementations share one key observation that motivates our design: elements in the data structure, representing the abstract state of the skip list, are reached through an index layer. This index layer is composed of metadata that does not belong to the abstract state of the data structure, and which is used to improve performance by minimizing the number of traversed nodes. Leveraging the above observation, we can split the memory space used by a skip list into a *data layer*, which stores the abstract state of the data structure, and an *index layer*, which includes the metadata exploited to reach the data layer. Figure 1a illustrates this separation.

Managing the data layer and index layer independently is the crucial intuition behind the NUMASK design, for it exploits the different consistency requirements they have to improve performance in NUMA architectures. None of the existing designs of NUMA-aware data structures, when applied to skip lists (e.g., [6]), accounts for such separation.

In a nutshell, in order to improve performance in NUMA architectures, the primary design choice of NUMASK is to create as many index layers as the number of NUMA zones in the system. These index layers are not updated immediately after successful insert/remove operations. Instead, they will be updated independently to avoid (unnecessary) synchronization and traffic on the NUMA interconnection. The ultimate goal of having NUMA-local index layers is to let operations on the data structure only access NUMA-local

(a) Skip list. **(b)** NUMASK.

Figure 1 Separation of layers in base skip list Vs. NUMASK. In 1b, the Intermediate layer has not been updated with key 7 yet.

memory before reaching the data layer. Once there, the (probabilistic) logarithmic complexity of the skip list allows for the traversal of only few nodes in the data layer before finalizing the operation. We empirically demonstrate that traversing these few nodes (possibly NUMA-remote) does not have a significant impact on performance. NUMASK accomplishes the above goal by deploying the following design around a *base skip list*.

4.1 Per-NUMA zone index layers

In skip lists, most of the traversed nodes exist in the index layer; therefore, creating as many index layers as the number of NUMA zones allows application threads to perform mostly NUMA-local accesses. Given that the *base skip list* defers updates to the index layer to a helper thread, having multiple independent indexing layers entails the need of deploying the same amount of helper threads (one per NUMA zone) responsible for their management. Consequently, helper threads will also access NUMA-local memory.

4.2 Per-NUMA zone intermediate layers

Decisions on how to update the index layer usually depend upon the current composition of the data layer. That is why the aforementioned per-NUMA zone helper threads, responsible for updating each instance of the index layers, would have to traverse the data layer nodes in order to decide whether to apply certain modifications (e.g., increasing or lowering a level of a certain node in the data layer) or to leave the index layer instance unaltered. Since the traversed data layer nodes are not necessarily NUMA-local, this can produce excessive NUMA-remote accesses and generate significant traffic on the NUMA interconnection, which is the main source of performance degradation in NUMA.

Because in NUMASK we aim at eliminating any NUMA-remote accesses while updating the index layer instances, we create a NUMA-local view of the data layer, which we name the *intermediate* layer. Creating multiple intermediate layers, one per index layer instance, allows helper threads to fully operate on NUMA-local memory. Logically, the intermediate layer is placed in between the index layer and the data layer. With respect to the index layer, the intermediate layer has the same goal as the *base skip list* data layer, meaning it serves as a knowledge base for the helper thread(s) to update the index layer instance(s).

The peculiarity of the intermediate layer is that it need not be an exact replica of the data layer (e.g., it is enough to be eventually synchronized with the data layer). In fact, any inaccuracy in an index layer instance, which could happen due to a temporarily out-dated intermediate layer, affects only the skip list performance and not its correctness. This is the same rationale that led previous skip list designs [8, 12, 17] to lazily update the index layer. Relaxed constraints on the intermediate layer composition enable its NUMA distribution.

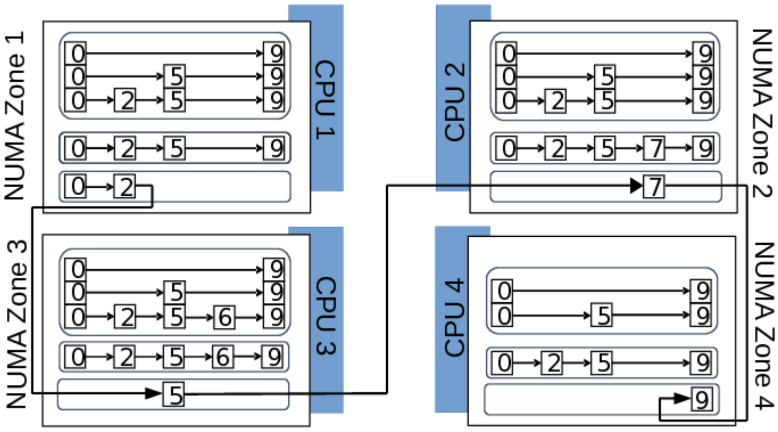


Figure 2 NUMASK deployed on a server with four sockets and four NUMA zones. The four instances of the index and intermediate layer are independent, and the data layer is scattered across available memory. The abstract state of the data structure contains the following keys: {0;2;5;7;9}.

In Figure 1b we show a simple example of NUMASK. Here the abstract state of the skip list is the same as Figure 1a; however, the intermediate layer has not been updated with the element with key 7. This is a plausible case in our design, meaning that the *insert*(7) operation result has not yet been propagated to the intermediate layer. We can easily see that the index layer remains the same as the skip list in Figure 1a. The modifications made by *insert*(7) will eventually be propagated to the intermediate layer using a technique (shown below) that does not increase the duration of the actual data structure operation.

4.3 Propagation of Data Layer Modifications.

The intermediate layer instances need to be periodically updated to reflect the content of the data layer. A naïve way to do this follows: at the end of each update operation (i.e., insert/delete), necessary information is stored in an intermediary data structure (e.g., a queue), and each per-NUMA helper thread later loads this information and updates its local intermediate layer. However, this naïve approach leads to one major drawback: it requires synchronization and memory allocation overhead on the data structure's critical path.

To remove this overhead from the application threads, NUMASK assigns a new helper thread the task of updating the intermediate layer instances. This thread operates at predefined intervals and iterates over the data layer. Every time it finds a node that has been modified (i.e., inserted or logically removed), it propagates this modification to all instances.

It is worth noting that this new helper thread does generate traffic on the NUMA-interconnection. However, the impact of this traffic on the data structure performance is minimal given that it does not operate frequently. Also, thanks to our optimizations in the index layers, the number of NUMA-remote accesses is already low (<15% in our experiments). Thus, the NUMA-interconnection is expected not to be saturated; therefore, this helper thread will not cause significant delay.

4.4 Example of NUMASK deployment

In Figure 2 we deployed NUMASK on a server with 4 processor sockets and 4 NUMA zones. In the example, the abstract state of the skip list is {0;2;5;7;9}. By looking at the data layer we assume that the elements 0 and 2 have been inserted by an application thread executing on CPU1, element 5 by a thread on CPU3, and so on. Each NUMA zone has its own intermediate and index layer instance. The composition of the different intermediate layer instances is different because the data layer modifications are not propagated at the same time to all intermediate later instances. For example, in the figure the element 6 has been removed, but the intermediate layer of NUMA zone 3 still has not applied this modification. Also, in the figure the four index layer instances differ from each other since helper threads work independently and do not proceed synchronously.

4.5 Design Trade-offs

The design of NUMASK presents different trade-offs with respect to the space and time needed to handle its index and intermediate layers, including tuning the configuration associated with the deployed helper threads. These trade-offs are briefly discussed below.

NUMASK introduces space overhead due to the presence of multiple instances of both index layer and intermediate layer. This overhead is proportional to the number of NUMA zones in the system; however it does not increase with the number of application threads. Moreover, as we will detail later, the synchronization overhead to maintain (i.e., traverse and update) this extra space is limited. Finally, it is important to note that, in cases where space utilization is crucial, some optimization can be added to NUMASK to control such utilization. For example, a probabilistic policy can be added to the data layer propagation process. This policy might aim at selecting only some operation made by application threads, rather than all, to be propagated to the different intermediate layer instances.

Another trade off involves the helper threads' frequency of operation. Tuning the backoff time after each iteration of the helper threads might affect the overall performance of NUMASK. One viable solution towards a configuration that is effective in multiple scenarios is to use an adaptive technique, similar to the one adopted in [18], in which the application workload is monitored and backoff time is adjusted accordingly.

5 NUMASK: Protocol Details

In this section we show the algorithmic details of NUMASK. The pseudo-code describing NUMASK is reported in Algorithms 2 and 3. To clarify the presentation, we abstract a base skip list in Algorithm 1. By leveraging this abstraction, we can avoid listing the details of core operations on the skip list (i.e., traversal, modification to data and index layer, logical and physical removal of elements) and focus on our NUMA-aware modifications. Algorithms 2 and 3 include calls to procedures defined in Algorithm 1. All the low-level details of our implementation are public and available in Synchrobench.

Algorithm 1 abstracts the base skip list as two procedures: `Base-Operation` and `Base-Helper`. `Base-Operation` is the handler for the three different types of data structure operations, namely `insert`, `remove`, and `contains`. Each of these operations is split into `Base-Traversal` and `Base-DoOperation` sub-procedures. The former traverses the index layer and returns a pointer to some data layer node where the operation should act. The latter works entirely on the data layer and applies the invoked operation (e.g., if the operation is an insert, the node is physically inserted in the data layer). `Base-Helper` periodically calls `Base-UpdateIndex` for updating the skip list index layer and performing physical removals.

As mentioned before, in our experiments we selected No Hotspot as the underlying base skip list implementation. The details of how No Hotspot implements `Base-Traversal`, `Base-DoOperation`, and `Base-UpdateIndex` can be found in [8].

Algorithm 1 Abstract Base Skip List

```
Global Variable: indexSen                                  ▷ indexSen = sentinel node of index layer
procedure BASE-OPERATION(Type t, Element el)                          ▷ t = Insert/Remove/Contains
    Node n = Base-Traversal(indexSen,el.key); ▷ n is the node with the closest key value less than or equal to the desired node
    boolean res = Base-DoOperation(t,el,n);
    return res;
end procedure

procedure BASE-HELPER(Node s)
    while true do
        Base-UpdateIndex(s);        ▷ This procedure updates the index layer starting from the sentinel node s
                                    ▷ In the base skip list, s is the sentinel node of the lowest level of the skip list
    end while
end procedure
```

5.1 NUMASK: Data Structure Operations

NUMASK's `Insert`, `Remove`, and `Contains` operations (Algorithm 2) can be summarized in the following steps: *i)* each operation traverses the local index layer instance until it retrieves a pointer to a node in the local intermediate layer; *ii)* this intermediate layer node is used as an indirection to reach a pointer to a data layer node; *iii)* this pointer is then used to perform the actual operation on the data layer. Importantly, the operations terminate right after updating the data layer, since all further updates in both intermediate and index layers are delegated to the helper threads (as detailed in the next two subsections).

Algorithm 2 NUMASK: Skip List Operations

```
1:  Global Variable:
2:  Node indexSents[MaxNumaZones]              ▷ Array of index layer sentinel nodes, one per NUMA zone
3:  Node interSents[MaxNumaZones]         ▷ Array of intermediate layer sentinel nodes, one per NUMA zone
4:  Node dataSent                                                           ▷ data layer sentinel node
5:  Queue update-queues[MaxNumaZones] ▷ Queue utilized for updating the MaxNumaZones intermediate layers

6:  Node: a struct with fields
7:      next                                                           ▷ Pointer to next node in the list
8:      down                                                  ▷ Pointer to the node in the level below
9:      status                           ▷ Up to date = 0, recently added = 1, recently removed = 2
10:     level                                       ▷ The height of the tallest tower in the index layer
11:     deleted                                                  ▷ Indicates if node is logically deleted

12: procedure NUMASK_OPERATION(Type t, Element el)
13:     Node intermediate_node = NUMASK_Traversal(getCurrentNUMAZone(), el.key);
14:     Node data_node = intermediate_node.down;
15:     boolean result = NUMASK_DoOperation(t, el, data_node);
16:     return result;
17: end procedure

18: procedure NUMASK_TRAVERSAL(int zone, Key k)      ▷ This procedure traverses the index layer associated
    with the local NUMA zone and returns a node in the intermediate layer
19:     Node n = Base-Traversal(indexSents[zone], k);
20:     return n
21: end procedure

22: procedure NUMASK_DOOPERATION(Type t, Element el, Node n)
23:     boolean result = Base-DoOperation(t, el, n);   ▷ If successful, DoOperation sets the altered node's status
24:     return result;
25: end procedure
```

The details of Algorithm 2 are as follows. In typical skip lists, index layer traversal starts from a known sentinel node. In NUMASK, each NUMA zone has its own index layer instance and therefore its own sentinel node as well (Algorithm 2:2). When a NUMASK traversal is invoked (Algorithm 2:18), the local thread starts from the sentinel node of the local NUMA zone. From this point, all memory accesses of `NUMASK_Traversal` will be NUMA-local. The traversal operates similar to that of the base skip list: it moves to a node on its right in the same level (using the *next* field) as long as its key is less than or equal to the target key (say k), and it moves to the next lower index level (using the *down* field) otherwise. If there is no lower index level to traverse, the traversal exits by returning the pointer to the node in the intermediate layer. Each node in the intermediate layer has a (*down*) pointer to its respective data layer node, from which `Base-DoOperation` can begin.

`Base-DoOperation` operates similar to the base skip list: The data layer is traversed from the pointer reached by the intermediate layer node until either a node with a greater key is found or the list ends. After that, the operation completes based on its type. If it is a `contains` operation, it checks whether the node's key matches k or not. The `insert` and `remove` operations use `Compare-And-Swap` for non-blocking updates (details of how No Hotspot, and thus NUMASK, accomplishes that can be found in [8]).

An important task assigned to `NUMASK_DoOperation` is to update the node's *status* field upon a successful write operation. Setting this field to 1 (respectively 2) indicates to helper threads that the node is newly inserted (respectively removed), and this insertion (respectively removal) is not yet propagated to the intermediate and index layers. To simplify the pseudo-code, we exclude this assignment of the status field, replacing it with a comment in Algorithm 2:23.

5.2 Data-Layer-Helper

In NUMASK, we create a single Data-Layer-Helper thread that periodically traverses the data layer in order to accomplish two objectives: *i)* it is responsible for feeding the different intermediate layer instances with the results of successful update operations on the data layer, and *ii)* it attempts to physically remove any logically-deleted nodes of the data layer.

In order to accomplish *i)*, the NUMASK design provides each intermediate layer instance with a single-producer/single-consumer queue (Algorithm 2: 5). As a consequence of this decision, there are as many queues as NUMA zones in NUMASK. The producer for all the queues is the same: the Data-Layer-Helper thread; while each queue has a different consumer: the Per-NUMA-Helper thread running in the queue's NUMA zone (detailed in the next subsection). We implemented these queues similar to the Vyukov SPSC queue [30].

The above queues are used to synchronize the data layer with intermediate layers as follows: when the Data-Layer-Helper thread traverses the data layer, each node's status field is checked to see if it is nonzero (which means it was recently inserted/removed); if so, it is added to the queue of each NUMA zone (Algorithm 3: 6) and its status field is reset to zero (to indicate that it is now up to date).

In order to accomplish *ii)*, the algorithm checks each node to see if it is logically deleted. If so, then it becomes a candidate to be physically removed. As in No Hotspot (as well as other concurrent skip lists), unlinking a node from the data layer can be done only if no tower above it is present in the index layer. However, since NUMASK deploys multiple index layer instances, the condition for physically removing one node is that no tower above it is present in any index layer instance. Verifying this condition is simple: each node in the data layer has a field named *level*. If the traversed node's *level* equals zero and it is logically deleted (Algorithm 3: 9), then the Data-Layer-Helper will proceed with its physical removal. In the next subsection we discuss how to update this *level* field.

By offloading the above two operations to a dedicated thread, the critical path of the application (`NUMASK_Operation`) is minimized. Note that populating the queues, which is required to update the intermediate layers (and therefore the index layers), entails an additional memory allocation overhead. This memory allocation could have been a dominant cost in the operation's critical path if we did not offload it to a separate helper thread.

A positive side effect of our dedicated Data-Layer-Helper thread is that while the thread

Algorithm 3 NUMASK: Updating Metadata

```
 1: procedure DATA-LAYER-HELPER    ▷ This procedure propagates recently altered nodes to intermediate layers
 2:     while true do
 3:         Node curr = dataSent.next;
 4:         while curr != NULL do
 5:             if curr.status != 0 then
 6:                 Add-Job-To-Queues(curr);
 7:                 curr.status = 0;
 8:             else
 9:                 if curr.level == 0 && curr.deleted then    ▷ If curr is logically deleted and there is no tower above it in any index layer
10:                     remove(curr);
11:                 end if
12:             end if
13:             curr = curr.next;
14:         end while
15:     end while
16: end procedure

17: procedure PER-NUMA-HELPER(int local_zone)
18:     while true do
19:         Update-Intermediate-Layer(local_zone)
20:         Base-UpdateIndex(interSents[local_zone]);    ▷ UpdateIndex is assumed to update the level field of nodes in the data and intermediate layer, when needed
21:     end while
22: end procedure

23: procedure ADD-JOB-TO-QUEUES(Node node)
24:     for  i = 0 to MaxNumaZones do
25:         update-queues[i].push(node);
26:     end for
27: end procedure

28: procedure UPDATE-INTERMEDIATE-LAYER(int z)    ▷ This function updates the intermediate layer of zone z
29:     Node sentinel = indexSents[z];
30:     while update-queues[z] is not empty do
31:         Node updatedNode = update-queue[z].pop();
32:         Node intermediate_node = NUMASK_Traversal(sentinel, updatedNode.key);
33:         if updatedNode.status == 1 then
34:             Node local-node = NUMA_alloc(updatedNode);    ▷ NUMA-aware memory allocator
35:             NUMASK_Operation(INSERT, local-node, intermediate_node);
36:         else
37:             NUMASK_Operation(REMOVE, updatedNode, intermediate_node);
38:         end if
39:     end while
40: end procedure
```

traverses the data layer, it reloads the cache of the processor on which it is executing, which increases cache hits for application threads that access the data layer. We exploit this idea further by rotating iterations of the Data-Layer-Helper thread between different NUMA zones. This way, caches in different NUMA zones (especially the L3 caches) are evenly refreshed. This process of refreshing caches is particularly effective when the data structure is not large; otherwise the number of elements evicted from cache might be large.

5.3 Per-NUMA-Helper

The role of Per-NUMA-Helper is to keep the index and intermediate layer of one NUMA zone updated. Consequently, NUMASK deploys one Per-NUMA-Helper thread per NUMA zone. Each iteration of the Per-NUMA-Helper thread performs two steps. First, it updates the local intermediate layer using the information contained in the queue of its NUMA zone (Algorithm 3:28). Second, it applies any needed modification to the local index layer.

The `Update-Intermediate-Layer` procedure (Algorithm 3:28) is responsible for achieving the first step. In this procedure, the Per-NUMA-Helper thread fetches jobs from the queue in the local NUMA zone and applies them to the local intermediate layer. To do that, Per-NUMA-Helper calls `NUMASK-Traversal` to reach the interested location of the local intermediate layer in logarithmic time. After that, the intermediate layer instance is updated by simply calling `NUMASK-Operation` using the intermediate node pointer returned by `NUMASK-Traversal`.

A critical low-level operation that happens during the `Update-Intermediate-Layer` procedure is the memory allocation of new nodes to be added to the local intermediate layer (Algorithm 3:34). It is required for all memory allocations by each Per-NUMA-Helper thread to be NUMA-local. Otherwise subsequent invocations of `NUMASK-Traversal` are not guaranteed to access entirely NUMA-local memory. In this regard, we tested multiple thread-local [2, 13] and NUMA-aware [1] allocators, but their overhead slowed performance. To deal with this problem, we developed a simple NUMA-aware memory allocator to serve memory allocation requests from Per-NUMA-Helper (see Section 6 for more details).

Once the local intermediate layer is updated, the procedure `Base-UpdateIndex` is called to update the index layer. In our implementation, inspired by No Hotspot, this procedure handles the raising and lowering of towers based on the composition of the intermediate layer, and it also handles removing any logically deleted nodes. First, the helper thread iterates over the intermediate layer, physically removing any nodes marked for deletion without any towers above (similar to what is done to the data layer nodes in `Data-Layer-Helper`). After that and if necessary, towers are raised or lowered to maintain the logarithmic complexity of the index layer traversals. When a tower is entirely removed in an index layer instance, the Per-NUMA-Helper thread accesses the linked node to the data layer and decrements its *level* field. Although changing the status field in such cases entails a NUMA-remote access, it is not a frequent operation, and thus it has a negligible impact on performance.

5.4 Correctness Arguments

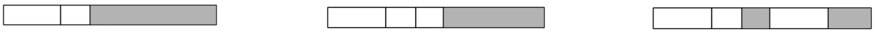
One of the advantages of NUMASK's design is its ability to reuse already-implemented basic operations to manipulate the data (and not metadata) of the data structure. None of our modifications needs to address how to insert or remove a node in the skip list data layer. Even the basic skip list traversal need not be modified.

Such a design makes it possible to integrate the NUMASK approach into other skip list implementations without affecting the overall correctness. This is noticeable by looking at how in Algorithms 2 and 3 we invoke procedures from Algorithm 1. In summary, if the base skip list is correct, then NUMASK will preserve such correctness.

6 NUMASK Optimization

Custom NUMA-aware Memory Allocator. NUMASK requires a mechanism to allocate memory in a thread's local NUMA zone. Without this, the proposed architecture would not be beneficial, as application and helper threads would frequently access NUMA-remote memory. Existing NUMA-aware memory allocators (e.g., *libnuma*) repeatedly interact with the operating system in order to retrieve NUMA-local memory. These interactions introduce a noticeable latency. After trying other memory allocators (e.g., [2, 13]), we decided to address our problem by developing a custom linear allocator to support the NUMASK design. To the best of our knowledge, this is the fastest design for memory allocation that fits our software architecture; it is simple yet effective.

Our NUMA allocator is used to serve allocation requests produced by Per-NUMA-Helper, therefore we deploy as many instances of our allocator as the number of Per-NUMA-Helper threads. Importantly, each of these allocator instances serves only one Per-NUMA-Helper thread; therefore, each allocator instance can be sequential (not concurrent).



(a) Initial allocator layout. **(b)** Layout after half cache line requested. **(c)** Layout after whole cache line requested.

Figure 3 Cache alignment scheme of our allocator. Grey blocks are free space; small white blocks are half cache line; large white blocks are whole cache line.

A linear (or monotonic) allocator consists of a fixed-size memory buffer allocated upon initialization and an internal offset to the beginning of the buffer's free space. Allocation requests increment the buffer offset by the size of the request and return the old value; thus requests are served in constant time without overhead, making the allocator fast.

Our allocator consists of a basic linear allocator plus three additions to fit our needs. The first addition is to allow the allocator to allocate new buffers (linear allocators usually do not reallocate memory). The second addition is to allocate the buffer in a specific NUMA zone, so that all the returned memory addresses reside in the same NUMA zone. With that, intermediate and index layers are formed of NUMA-local memory.

The final addition to our allocator deals with request alignment. Since the allocator is only used to create index and intermediate nodes, and their sizes are less than and greater than a half cache line, respectively, the requests are automatically aligned to either a half or whole cache line. The allocator keeps track of the previous request's alignment internally and aligns the current request based on the previous alignment and the size of the current request. This internal bookkeeping allows the allocator to fit two index nodes in a cache line, which in turn results in faster index traversal, for two nodes in the same cache line will likely be near each other in the index layer, thus reducing necessary memory accesses.

Figure 3 details how the allocator aligns requests in different scenarios. The example begins in Figure 3a; the previous two requests resulted in a whole cache-line alignment and a half cache-line alignment. Depending on the next request, the allocator could result in two separate layouts. If the next request is an index node (size less than half a cache-line), the allocator can squeeze it in the half cache-line free space. Figure 3b shows the result in this case. However, if an intermediate node is the next memory allocation, the allocator will move the offset to the beginning of the next cache-line to keep the intermediate node from spilling over two cache lines. Figure 3c depicts this. Note that the free space skipped over in Figure 3c will not be used.

Avoiding Synchronization When Updating Intermediate Layer. In Section 5.3 we discussed how each Per-NUMA-Helper thread updates the local intermediate layer. In the pseudo-code we do that by invoking `NUMASK-Operation`, which uses synchronization primitives, since it is the same function used by application threads to operate on the data layer. This task can be changed to let Per-NUMA-Helper modify the intermediate layer without any atomic operations as follows. In order to make updates on an intermediate layer instance synchronization-free, we need to disallow `NUMASK_Operation` from using the intermediate layer to access the data layer (see Algorithm 2:14). To do so, in our implementation we store the pointer to the data layer directly in the index nodes so that application threads never need to access the intermediate layer.

7 Evaluation

We implemented NUMASK in C++, and integrated it into Synchrobench [17], a benchmark suite for concurrent data structures. In addition to providing a common software

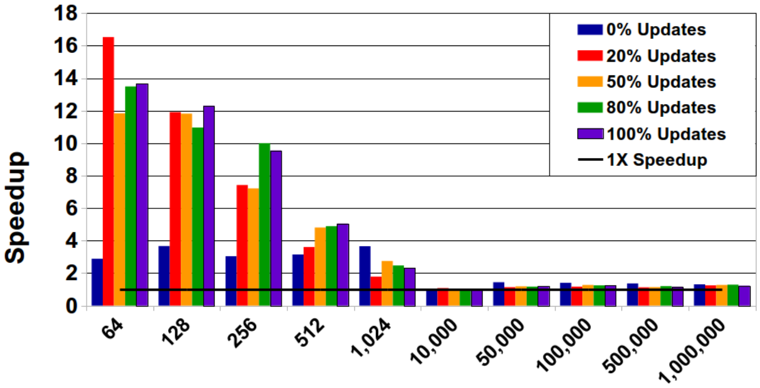


Figure 4 Speedup NUMASK over No Hotspot varying data structure size.

architecture to configure and test different data structure implementations, Synchrobench already implements many state-of-the-art high performance solutions that we used to compare against NUMASK. Specifically, we selected three concurrent skip list implementations: No Hotspot [8], Fraser [15], and Rotating skip list [12]. We also included a sequential skip list implementation [17]. As specified earlier in the paper, NUMASK has been built using No Hotspot as a base skip list implementation for two reasons: it is among the fastest concurrent skip lists of which we are aware, and it alleviates contention by deferring index layer updates.

Our testbed consists of a server with 4 Intel Xeon Platinum 8160 processors (2.1GHz, 24/48 cores/threads per CPU). The machine provides 192 hardware threads. There are 4 sockets hosting the 4 processors, via 4 NUMA zones (one per socket), and 768 GB of memory. In our experiments we ran up to 160 application threads (the actual number of executing threads is higher because of the helper threads used by each competitor) to leave enough resources to the operating system to execute without creating bottlenecks. In our experiments we distribute application threads evenly across NUMA zones.

The workloads we use to test competitors perform insert/remove/contains operations. Note that in order to keep the size of the data structure consistent, during removal the application attempts to pick elements that have previously been inserted successfully. Each test has a warm-up phase where the skip list is populated and the index is built. This phase is also used to fill out L1/L2/L3 caches. After that, the application runs for 10 seconds while collecting statistics. In the experiments we use a range of key elements that is twice the data structure size; and all elements have integer keys. All results are averages of five test runs.

Before showing the throughput of all competitors, we report two plots that summarize the advantages of NUMASK over the base skip list, which is No Hotspot in our case. Figure 4 demonstrates the speedup of NUMASK over No Hotspot by varying the initial size of the data structure, in the range 64 to 1M elements. To improve clarity, a line is drawn to show when speedup equals 1. We test different percentages of update operations and we record the value for the best performance among all thread ranges. Although for clarity we cannot include the number of threads corresponding to each data point in the plot, it is worth noting that, in our evaluation settings, NUMASK is most effective when the number of threads exceeds 64, as it will be clear analyzing Figure 6. As a result, for all data points in Figure 4, the number of application threads is always in the range of 64 to 160.

NUMASK's speedup grows significantly when the data structure size decreases. This is mostly due to its capability of exploiting NUMA-local accesses and leveraging cache locality. In fact, with sizes less than 10k elements, most of the data structure will likely fit in processors' caches, but the presence of updates forces frequent cache refreshing. This refreshing requires loading memory locations from main memory. In No Hotspot, this is likely to be in a remote NUMA zone given that the machine has 4 NUMA zones. However, NUMASK was designed to keep most of the needed memory locations in the local NUMA zone. This is also confirmed by the result using 0% updates; here the speed up is significantly less than in write-intensive workloads because both competitors can benefit from cache locality. Considering 50% updates and 128 elements NUMASK is 11x faster than No Hotspot; and at 100K elements NUMASK is 27% faster. Interestingly, the plots in Figures 6g-6i, meaning when the data structure size is set at 100k, show how NUMASK's performance does not degrade with respect to competitors. In these cases, the most dominant cost for all is poor cache locality, which brings down performance.

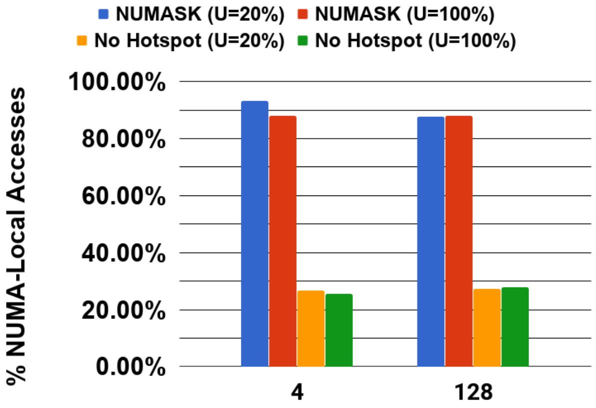
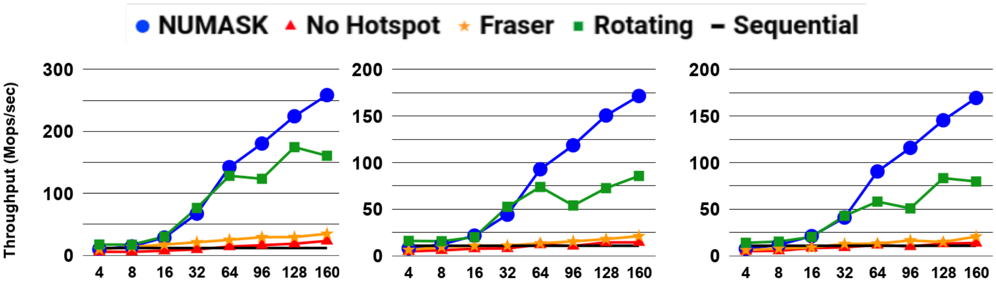


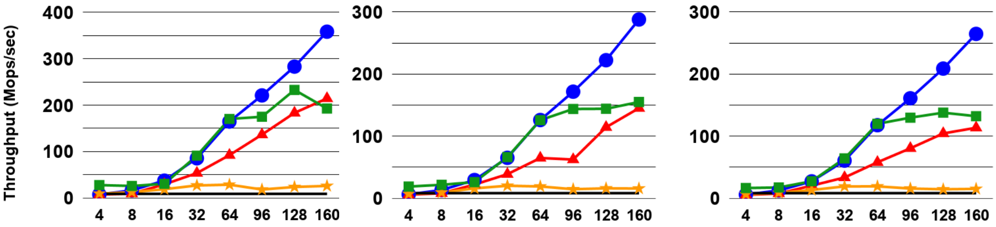
Figure 5 NUMA-local accesses in NUMASK and No Hotspot using {4,128} application threads.

Figure 5 shows the key reason for the performance improvement of NUMASK: its NUMA-local accesses. To collect statistics, we monitored memory accesses performed by application threads and contrasted the application thread's local NUMA zone with the NUMA zone in which the memory location resides. Here the initial size of the data structure is 100K, and we configured the system to run with 4 and 128 application threads. No Hotspot hovers around 25%, which is the immediate consequence of having uniform distribution of data structure accesses and 4 NUMA zones; NUMASK is around 90% because of its NUMA-aware design. An observation that is not shown in the plots is that the percentage of NUMA-local accesses for the Per-NUMA-Helper threads is consistently slightly lower than 100% (recall that each Per-NUMA-Helper can occasionally access some NUMA-remote location as detailed in Section 5.3).

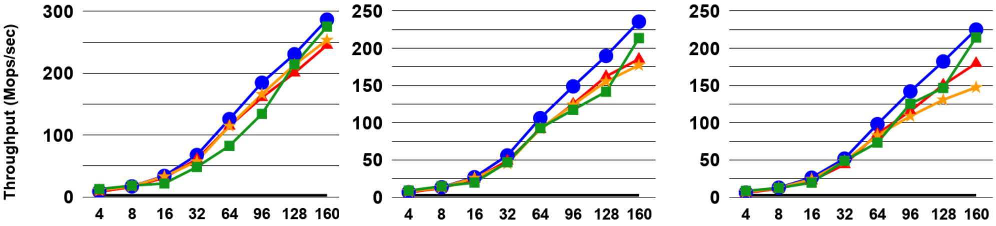
Figure 6 shows the throughput of NUMASK against the Fraser, Rotating, and No Hotspot skip lists by varying the number of application threads, data structure size, and percentage of update operations. Throughput is measured in millions of operations successfully completed per second. A specially relevant case is the one where the data structure is 1K elements. In the read-intensive scenario, all competitors scale well except for Fraser, with NUMASK demonstrating the highest performance. With 50% and 80% of updates, all competitors stop scaling beyond 64 threads while NUMASK continues scaling, hitting the remarkable performance of 300 million operations per second with 50% updates. In this configuration, at 160 threads NUMASK outperforms rotating skiplist and No Hotspot by 2x.



(a) 128;20% **(b)** 128;50% **(c)** 128;80%



(d) 1024;20% **(e)** 1024;50% **(f)** 1024;80%



(g) 100k;20% **(h)** 100k;50% **(i)** 100k;80%

Figure 6 Throughput of NUMASK against other skip list implementations varying data structure size and the percentage of update operations. Throughput is in Millions operations per second.

Reducing the data structure size improves the gap between NUMASK and the other competitors. This is reasonable since our NUMA design avoids synchronization across NUMA zones, which would generate many NUMA-remote accesses.

At 100k element size, the gaps among competitors is reduced. Sill, NUMASK is the fastest at 50% updates and 160 threads by gaining 10% over Rotating and 27% over No Hotspot. As mentioned before and confirmed by the analysis of the cache hits/misses, the dominant cost here is repeatedly loading new elements into the cache. This cost obfuscates the effort in improving performance made by NUMASK's design. No Hotspot's performance evaluation also discusses similar findings with large data structure sizes.

8 Conclusion

In this paper we presented NUMASK, a high-performance concurrent skip list that uses a combination of distributed design and eventual synchronization to improve performance in NUMA architectures. Our evaluation study shows unquestionably high throughput and remarkable speedups: up to 16x in write-intensive workloads and in the presence of contention.

References

1 *numa(3) Linux Programmer's Manual*, second edition, December 2007. `https://linux.die.net/man/3/numa`.

2 Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In Larry Rudolph and Anoop Gupta, editors, *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, pages 117–128. ACM Press, 2000. Source code available at `https://github.com/emeryberger/Hoard`. URL: `http://doi.acm.org/10.1145/356989.357000`, `doi:10.1145/356989.357000`.

3 Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1854273.1854350`, `doi:10.1145/1854273.1854350`.

4 Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 121–132. ACM, 2016. URL: `http://doi.acm.org/10.1145/2935764.2935796`, `doi:10.1145/2935764.2935796`.

5 Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware Reader-writer Locks. In *PPoPP '13*, 2013.

6 Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 207–221. ACM, 2017. URL: `http://doi.acm.org/10.1145/3037697.3037721`, `doi:10.1145/3037697.3037721`.

7 Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 161–170. ACM, 2012. URL: `http://doi.acm.org/10.1145/2145816.2145837`, `doi:10.1145/2145816.2145837`.

8 Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 196–205. IEEE Computer Society, 2013. URL: `https://doi.org/10.1109/ICDCS.2013.42`, `doi:10.1109/ICDCS.2013.42`.

9 Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 381–394. ACM, 2013. URL: `http://doi.acm.org/10.1145/2451116.2451157`, `doi:10.1145/2451116.2451157`.

10 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 631–644. ACM, 2015. URL: `http://doi.acm.org/10.1145/2694344.2694359`, `doi:10.1145/2694344.2694359`.

11 David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12*, 2012.

12 Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), 2017. URL: `https://doi.org/10.1002/cpe.3876`, `doi:10.1002/cpe.3876`.

13 Jason Evans. jemalloc memory allocator. `https://github.com/jemalloc/jemalloc`.

14 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM. URL: `http://doi.acm.org/10.1145/1011767.1011776`, `doi:10.1145/1011767.1011776`.

15 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, September 2003.

16 Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Commun. ACM*, 58(12):59–66, November 2015. URL: `http://doi.acm.org/10.1145/2814328`, `doi:10.1145/2814328`.

17 Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 1–10. ACM, 2015. URL: `http://doi.acm.org/10.1145/2688500.2688501`, `doi:10.1145/2688500.2688501`.

18 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less balanced tree. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.

19 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1810479.1810540`, `doi:10.1145/1810479.1810540`.

20 M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

21 Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings*, pages 124–138, 2007.

22 Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013. URL: `http://doi.acm.org/10.1145/2508834.2513149`, `doi:10.1145/2508834.2513149`.

23 Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 277–289. USENIX Association, 2015. URL: `https://www.usenix.org/conference/atc15/technical-session/presentation/lepers`.

24 Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/1993478.1993481`, `doi:10.1145/1993478.1993481`.

25 Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. On designing numa-aware concurrency control for scalable transactional memory. In Rafael

Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 45:1–45:2. ACM, 2016. URL: `http://doi.acm.org/10.1145/2851141.2851189`, `doi:10.1145/2851141.2851189`.

26 Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 3–18. ACM, 2014. URL: `http://doi.acm.org/10.1145/2541940.2541965`, `doi:10.1145/2541940.2541965`.

27 Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Timothy L. Harris. Analytics with smart arrays: adaptive and efficient language-independent data. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 17:1–17:15. ACM, 2018. URL: `http://doi.acm.org/10.1145/3190508.3190514`, `doi:10.1145/3190508.3190514`.

28 William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. URL: `http://doi.acm.org/10.1145/78973.78977`, `doi:10.1145/78973.78977`.

29 Nikita Shamgunov. The memsql in-memory database system. In Justin J. Levandoski and Andrew Pavlo, editors, *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014.*, 2014.

30 Dmitry Vyukov. Unbounded SPSC Queue, 2018. `http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue`.