

1 NUMASK: High Performance Scalable Skip List 2 for NUMA

3 **Henry Daly**

4 Lehigh University, Bethlehem, PA, USA
5 hwd219@lehigh.edu

6 **Ahmed Hassan**

7 Alexandria University, Alexandria, Egypt
8 ahmed.hassan@alexu.edu.eg

9 **Michael F. Spear**

10 Lehigh University, Bethlehem, PA, USA
11 spear@lehigh.edu

12 **Roberto Palmieri**

13 Lehigh University, Bethlehem, PA, USA
14 palmieri@lehigh.edu

15 — Abstract —

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

26 **2012 ACM Subject Classification** Information systems Data structures

27 **Keywords and phrases** Skip list, NUMA, Concurrent Data Structure

28 **Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.18

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

32 **Acknowledgements** Authors would like to thank anonymous reviewers for the insightful com-
33 ments, Maged Michael and Dave Dice for the early feedback on the paper, and Vincent Gramoli
34 for agreeing to integrate NUMASK into Synchrobench.

35 **1** Introduction

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



© Henry Daly and Ahmed Hassan and Michael F. Spear and Roberto Palmieri;
licensed under Creative Commons License CC-BY

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



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

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

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

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

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

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

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

- 85 - We define independent index layers (one per NUMA zone) for the skip list. Each operation
 86 traverses the index layer that is local to the thread that executes it. This way, operations

87 do not need to traverse the interconnection between NUMA zones during the index layer
88 traversal. Importantly, we do not keep these index layers consistent with each other; we
89 allow them to be different. In fact, having different index layers in different NUMA zones
90 does not affect correctness because the actual data (which resides in the lowest level of
91 the skip list) is still synchronized.

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

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

103 We implement NUMASK in C++ and integrate into Sychrobench [17], a comprehensive
104 suite of data structures implemented in the same optimized software infrastructure. The
105 implementation of NUMASK has been enriched with specific optimizations, such as an
106 efficient NUMA memory allocator, developed on top of `libnuma` [1], to avoid bottleneck.
107 We compare the performance of NUMASK with three state-of-the-art concurrent skip lists:
108 Fraser [15], No Hotspot [8], and Rotating Skip List [12]. Performance shows up to 16x speed
109 up for write workloads and improvements up to 40% in read-intensive workloads. In summary,
110 NUMASK hits an important performance goal: in low-contention workloads, NUMASK adds
111 no overhead to the high-performance concurrent data structures; and in high-contention
112 workloads, NUMASK outperforms all other competitors and keeps scaling (we tested up to
113 160 threads) while other competitors stop earlier (at 64 threads in our experiments).

114 NUMASK is part of the core release of Sychrobench [17] available at <https://github.com/gramoli/sychrobench>.
115

116 2 Related Work

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

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

131 The impact of NUMA organization on the performance of software components (e.g. data
132 structures and thread synchronization) is an important topic. Interestingly, the last decade

133 saw the proposal of many NUMA-aware building blocks to improve application performance.
 134 Examples include NUMA-aware lock implementations [11, 5], thread placement policy [23],
 135 and smart data arrays [27]. Although helpful, the applicability of these components in linked
 136 data structures is limited due to the memory organization required by data structures in
 137 order to implement their operations while preserving the asymptotic complexity.

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

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

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

163 **3 Terminology, NUMA & Linked Data Structures**

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

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

178 Linked data structures are particularly affected by the memory latency variation intro-

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

185 **4 NUMASK: A Concurrent Skip List Designed for NUMA**

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

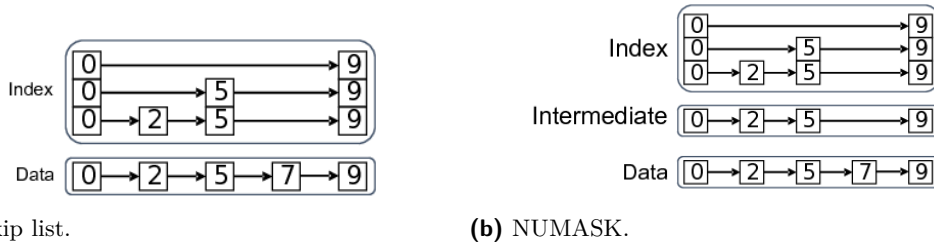
194 In the rest of the paper we will use the term *base skip list* to indicate an implementation
195 of a skip list that is wrapped (and improved) by the NUMASK architecture. The *base skip*
196 *list* is a concurrent skip list whose API are *insert*, *remove*, and *contains* operations, with
197 their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

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

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

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

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



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

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

230 4.1 Per-NUMA zone index layers

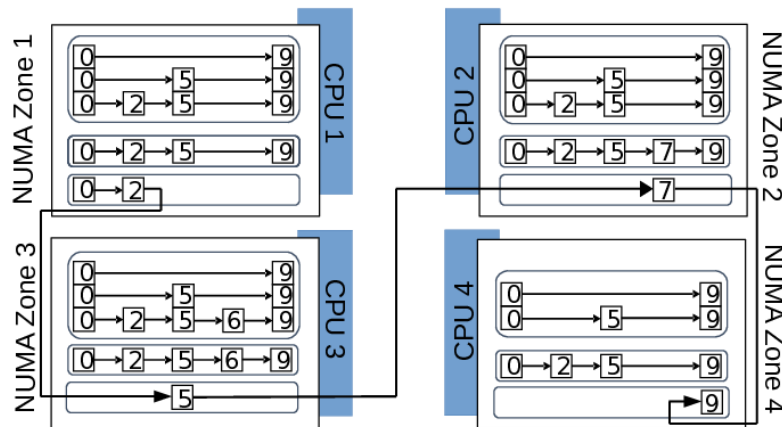
231 In skip lists, most of the traversed nodes exist in the index layer; therefore, creating as many
 232 index layers as the number of NUMA zones allows application threads to perform mostly
 233 NUMA-local accesses. Given that the *base skip list* defers updates to the index layer to a
 234 helper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

237 4.2 Per-NUMA zone intermediate layers

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

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

253 The peculiarity of the intermediate layer is that it need not be an exact replica of the
 254 data layer (e.g., it is enough to be eventually synchronized with the data layer). In fact, any
 255 inaccuracy in an index layer instance, which could happen due to a temporarily out-dated
 256 intermediate layer, affects only the skip list performance and not its correctness. This is the
 257 same rationale that led previous skip list designs [8, 12, 17] to lazily update the index layer.
 258 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: $\{0;2;5;7;9\}$.

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

266 4.3 Propagation of Data Layer Modifications.

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

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

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

283 4.4 Example of NUMASK deployment

284 In Figure 2 we deployed NUMASK on a server with 4 processor sockets and 4 NUMA zones.
 285 In the example, the abstract state of the skip list is $\{0;2;5;7;9\}$. By looking at the data layer

286 we assume that the elements 0 and 2 have been inserted by an application thread executing
 287 on CPU1, element 5 by a thread on CPU3, and so on. Each NUMA zone has its own
 288 intermediate and index layer instance. The composition of the different intermediate layer
 289 instances is different because the data layer modifications are not propagated at the same
 290 time to all intermediate later instances. For example, in the figure the element 6 has been
 291 removed, but the intermediate layer of NUMA zone 3 still has not applied this modification.
 292 Also, in the figure the four index layer instances differ from each other since helper threads
 293 work independently and do not proceed synchronously.

294 4.5 Design Trade-offs

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

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

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

312 5 NUMASK: Protocol Details

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

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

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

Algorithm 1 Abstract Base Skip List

```

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

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

```

331 **5.1 NUMASK: Data Structure Operations**

332 NUMASK's **Insert**, **Remove**, and **Contains** operations (Algorithm 2) can be summarized in
333 the following steps: *i*) each operation traverses the local index layer instance until it retrieves
334 a pointer to a node in the local intermediate layer; *ii*) this intermediate layer node is used
335 as an indirection to reach a pointer to a data layer node; *iii*) this pointer is then used to
336 perform the actual operation on the data layer. Importantly, the operations terminate right
337 after updating the data layer, since all further updates in both intermediate and index layers
338 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

```

339 The details of Algorithm 2 are as follows. In typical skip lists, index layer traversal starts
340 from a known sentinel node. In NUMASK, each NUMA zone has its own index layer instance
341 and therefore its own sentinel node as well (Algorithm 2:2). When a NUMASK traversal is
342 invoked (Algorithm 2:18), the local thread starts from the sentinel node of the local NUMA
343 zone. From this point, all memory accesses of **NUMASK_Traversal** will be NUMA-local. The
344 traversal operates similar to that of the base skip list: it moves to a node on its right in the
345 same level (using the *next* field) as long as its key is less than or equal to the target key

346 (say k), and it moves to the next lower index level (using the *down* field) otherwise. If there
 347 is no lower index level to traverse, the traversal exits by returning the pointer to the node
 348 in the intermediate layer. Each node in the intermediate layer has a (*down*) pointer to its
 349 respective data layer node, from which `Base-DoOperation` can begin.

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

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

362 5.2 Data-Layer-Helper

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

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

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

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

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

392 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

```

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

399 5.3 Per-NUMA-Helper

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

405 The `Update-Intermediate-Layer` procedure (Algorithm 3:28) is responsible for achieving
406 the first step. In this procedure, the Per-NUMA-Helper thread fetches jobs from the queue in
407 the local NUMA zone and applies them to the local intermediate layer. To do that, Per-NUMA-
408 Helper calls `NUMASK-Traversal` to reach the interested location of the local intermediate layer

409 in logarithmic time. After that, the intermediate layer instance is updated by simply calling
 410 `NUMASK-Operation` using the intermediate node pointer returned by `NUMASK-Traversal`.

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

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

430 5.4 Correctness Arguments

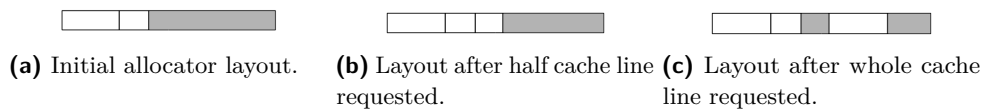
431 One of the advantages of NUMASK’s design is its ability to reuse already-implemented basic
 432 operations to manipulate the data (and not metadata) of the data structure. None of our
 433 modifications needs to address how to insert or remove a node in the skip list data layer.
 434 Even the basic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

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

439 6 NUMASK Optimization

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

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



■ **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.

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

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

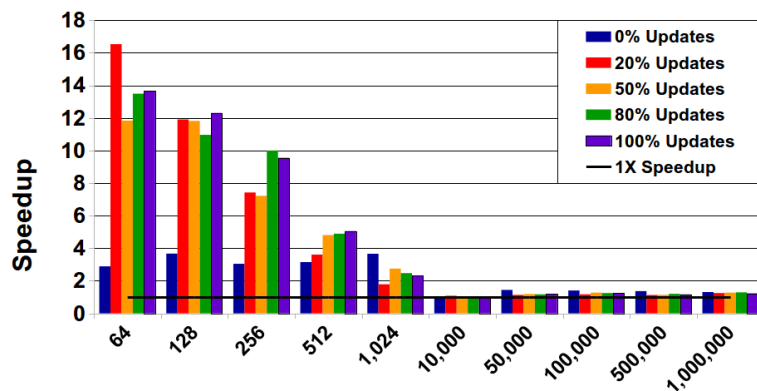
462 The final addition to our allocator deals with request alignment. Since the allocator is
 463 only used to create index and intermediate nodes, and their sizes are less than and greater
 464 than a half cache line, respectively, the requests are automatically aligned to either a half or
 465 whole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

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

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

489 7 Evaluation

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



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

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

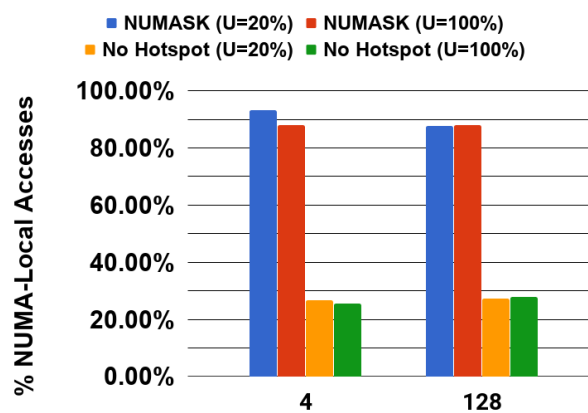
499 Our testbed consists of a server with 4 Intel Xeon Platinum 8160 processors (2.1GHz,
 500 24/48 cores/threads per CPU). The machine provides 192 hardware threads. There are
 501 4 sockets hosting the 4 processors, via 4 NUMA zones (one per socket), and 768 GB of
 502 memory. In our experiments we ran up to 160 application threads (the actual number of
 503 executing threads is higher because of the helper threads used by each competitor) to leave
 504 enough resources to the operating system to execute without creating bottlenecks. In our
 505 experiments we distribute application threads evenly across NUMA zones.

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

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

523 NUMASK’s speedup grows significantly when the data structure size decreases. This
 524 is mostly due to its capability of exploiting NUMA-local accesses and leveraging cache

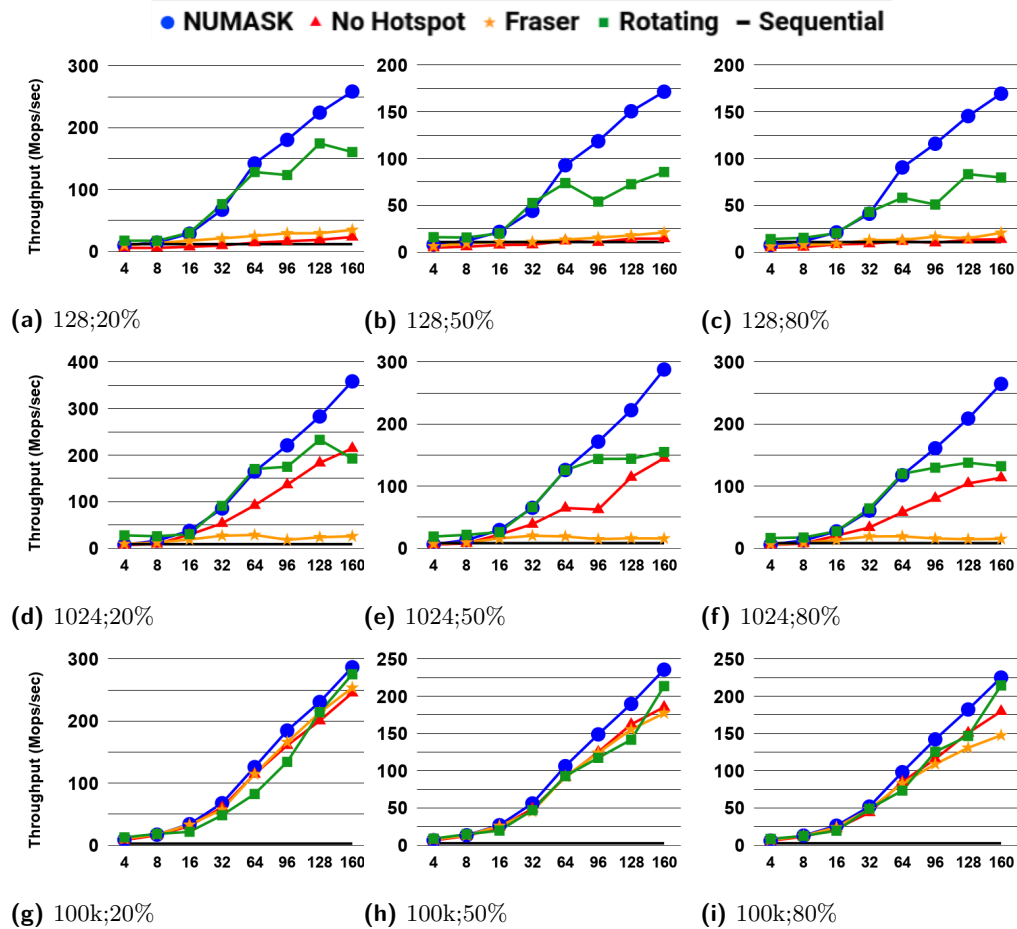
525 locality. In fact, with sizes less than 10k elements, most of the data structure will likely
 526 fit in processors' caches, but the presence of updates forces frequent cache refreshing. This
 527 refreshing requires loading memory locations from main memory. In No Hotspot, this is
 528 likely to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



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

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

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



■ **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.

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

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

566 8 Conclusion

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

References

- 571 ———
- 572 **1** *numa(3) Linux Programmer's Manual*, second edition, December 2007. <https://linux.die.net/man/3/numa>.
- 573
- 574 **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.
- 575
- 576
- 577
- 578
- 579
- 580
- 581 **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.
- 582
- 583
- 584
- 585
- 586 **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.
- 587
- 588
- 589
- 590
- 591
- 592 **5** Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware Reader-writer Locks. In *PPoPP '13*, 2013.
- 593
- 594 **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.
- 595
- 596
- 597
- 598
- 599
- 600 **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.
- 601
- 602
- 603
- 604
- 605 **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.
- 606
- 607
- 608
- 609 **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.
- 610
- 611
- 612
- 613
- 614 **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.
- 615
- 616
- 617
- 618
- 619
- 620

- 621 **11** David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique
622 for Designing NUMA Locks. In *PPoPP '12*, 2012.
- 623 **12** Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and*
624 *Computation: Practice and Experience*, 29(4), 2017. URL: [https://doi.org/10.1002/](https://doi.org/10.1002/cpe.3876)
625 [cpe.3876](https://doi.org/10.1002/cpe.3876), doi:10.1002/cpe.3876.
- 626 **13** Jason Evans. jemalloc memory allocator. <https://github.com/jemalloc/jemalloc>.
- 627 **14** Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of*
628 *the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC
629 '04, pages 50–59, New York, NY, USA, 2004. ACM. URL: [http://doi.acm.org/10.1145/](http://doi.acm.org/10.1145/1011767.1011776)
630 [1011767.1011776](http://doi.acm.org/10.1145/1011767.1011776), doi:10.1145/1011767.1011776.
- 631 **15** Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, September 2003.
- 632 **16** Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova,
633 Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management
634 on modern numa systems. *Commun. ACM*, 58(12):59–66, November 2015. URL: [http://doi.acm.org/10.1145/](http://doi.acm.org/10.1145/2814328)
635 [2814328](http://doi.acm.org/10.1145/2814328), doi:10.1145/2814328.
- 636 **17** Vincent Gramoli. More than you ever wanted to know about synchronization: syn-
637 chrobench, measuring the impact of the synchronization on concurrent algorithms. In
638 Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Sym-*
639 *posium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco,*
640 *CA, USA, February 7-11, 2015*, pages 1–10. ACM, 2015. URL: [http://doi.acm.org/10.](http://doi.acm.org/10.1145/2688500.2688501)
641 [1145/2688500.2688501](http://doi.acm.org/10.1145/2688500.2688501), doi:10.1145/2688500.2688501.
- 642 **18** Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less
643 balanced tree. In *Distributed Computing - 29th International Symposium, DISC 2015,*
644 *Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.
- 645 **19** Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the
646 synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM*
647 *Symposium on Parallelism in Algorithms and Architectures, SPAA '10*, pages 355–364, New
648 York, NY, USA, 2010. ACM. URL: [http://doi.acm.org/10.1145/](http://doi.acm.org/10.1145/1810479.1810540)
649 [1810479.1810540](http://doi.acm.org/10.1145/1810479.1810540), doi:10.1145/1810479.1810540.
- 650 **20** M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann,
651 2008.
- 652 **21** Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist
653 algorithm. In *Structural Information and Communication Complexity, 14th International*
654 *Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings*, pages
655 124–138, 2007.
- 656 **22** Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–
657 40:51, July 2013. URL: <http://doi.acm.org/10.1145/2508834.2513149>, doi:10.1145/
658 [2508834.2513149](http://doi.acm.org/10.1145/2508834.2513149).
- 659 **23** Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory place-
660 ment on NUMA systems: Asymmetry matters. In Shan Lu and Erik Riedel, editors,
661 *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara,*
662 *CA, USA*, pages 277–289. USENIX Association, 2015. URL: [https://www.usenix.org/](https://www.usenix.org/conference/atc15/technical-session/presentation/lepers)
663 [conference/atc15/technical-session/presentation/lepers](https://www.usenix.org/conference/atc15/technical-session/presentation/lepers).
- 664 **24** Zoltan Majo and Thomas R. Gross. Memory management in numa multicore sys-
665 tems: Trapped between cache contention and interconnect overhead. In *Proceedings of*
666 *the International Symposium on Memory Management, ISMM '11*, pages 11–20, New
667 York, NY, USA, 2011. ACM. URL: [http://doi.acm.org/10.1145/](http://doi.acm.org/10.1145/1993478.1993481)
668 [1993478.1993481](http://doi.acm.org/10.1145/1993478.1993481), doi:10.1145/1993478.1993481.
- 669 **25** Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. On
670 designing numa-aware concurrency control for scalable transactional memory. In Rafael

- 671 Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on*
672 *Principles and Practice of Parallel Programming, PPOPP 2016, Barcelona, Spain, March*
673 *12-16, 2016*, pages 45:1–45:2. ACM, 2016. URL: <http://doi.acm.org/10.1145/2851141.2851189>, doi:10.1145/2851141.2851189.
- 675 **26** Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot.
676 Scale-out NUMA. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors,
677 *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14,*
678 *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.
- 680 **27** Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi,
681 and Timothy L. Harris. Analytics with smart arrays: adaptive and efficient language-
682 independent data. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings*
683 *of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*,
684 pages 17:1–17:15. ACM, 2018. URL: <http://doi.acm.org/10.1145/3190508.3190514>,
685 doi:10.1145/3190508.3190514.
- 686 **28** William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*,
687 33(6):668–676, 1990. URL: <http://doi.acm.org/10.1145/78973.78977>, doi:10.1145/
688 78973.78977.
- 689 **29** Nikita Shamgunov. The memsql in-memory database system. In Justin J. Levandoski and
690 Andrew Pavlo, editors, *Proceedings of the 2nd International Workshop on In Memory Data*
691 *Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014.*, 2014.
- 692 **30** Dmitry Vyukov. Unbounded SPSC Queue, 2018. [http://www.1024cores.net/home/](http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue)
693 [lock-free-algorithms/queues/unbounded-spsc-queue](http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue).