SmartPQ: An Adaptive Concurrent Priority Queue for NUMA Architectures

*Christina Giannoula[†] *Foteini Strati^{‡†} Dimitrios Siakavaras[†] Georgios Goumas[†] Nectarios Koziris[†]

[†]National Technical University of Athens [‡]ETH Zürich

Concurrent priority queues are widely used in important workloads, such as graph applications and discrete event simulations. However, designing scalable concurrent priority queues for NUMA architectures is challenging. Even though several NUMA-oblivious implementations can scale up to a high number of threads, exploiting the potential parallelism of insert operation, NUMA-oblivious implementations scale poorly in deleteMin-dominated workloads. This is because all threads compete for accessing the same memory locations, i.e., the highest-priority element of the queue, thus incurring excessive cache coherence traffic and non-uniform memory accesses between NUMA nodes. In such scenarios, NUMA-aware implementations are typically used to improve system performance on a NUMA system.

In this work, we propose an adaptive priority queue, called SmartPQ. SmartPQ tunes itself by switching between a NUMA-oblivious and a NUMA-aware algorithmic mode to achieve high performance under all various contention scenarios. SmartPQ has two key components. First, it is built on top of NUMA Node Delegation (Nuddle), a generic lowoverhead technique to construct efficient NUMA-aware data structures using any arbitrary concurrent NUMA-oblivious implementation as its backbone. Second, SmartPQ integrates a lightweight decision making mechanism to decide when to switch between NUMA-oblivious and NUMA-aware algorithmic modes. Our evaluation shows that, in NUMA systems, SmartPQ performs best in all various contention scenarios with 87.9% success rate, and dynamically adapts between NUMA-aware and NUMA-oblivious algorithmic mode, with negligible performance overheads. SmartPQ improves performance by 1.87× on average over SprayList, the state-of-theart NUMA-oblivious priority queue.

1. Introduction

Concurrent data structures are widely used in the software stack, i.e., kernel, libraries and applications. Prior works [9, 10, 14, 65] discuss the need for efficient and scalable concurrent data structures for commodity Non-Uniform Memory Access (NUMA) architectures. Pointer chasing data structures such as linked lists, skip lists and search trees have inherently low contention, since concurrent threads search for different elements during their operations. Recent works [10,16,70] have shown that lock-free algorithms [22,24,29,36,52,55] of such data structures can scale to hundreds of cores. On the other hand, data structures such as queues and stacks typically incur high contention, when accessed by many threads. In these data structures, concurrent threads compete for the *same* memory elements (locations), incurring excessive traffic and

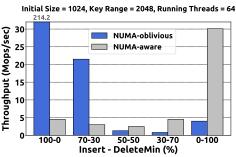


Figure 1: Throughput achieved by a *NUMA-oblivious* [2, 34] and a *NUMA-aware* [65] priority queue, both initialized with 1024 keys. We use 64 threads that perform a mix of *insert* and *deleteMin* operations in parallel, and the key range is set to 2048 keys. We use *all* NUMA nodes of a 4-node NUMA system, the characteristics of which are presented in Section 4.

non-uniform memory accesses between nodes of a NUMA system.

In this work, we focus on priority queues, which are widely used in a variety of applications, including task scheduling in real-time and computing systems [79], discrete event simulations [49, 75] and graph applications [39, 43, 76], e.g., Single Source Shortest Path [12] and Minimum Spanning Tree [60]. Similarly to skip-lists and search trees, priority queues have two main operations: *insert* and *deleteMin*. The *insert* operation, concurrent priority queues typically exhibits high levels of parallelism and low-contention, since threads may work on different parts of the data structure. Therefore, concurrent *NUMA-oblivious* implementations [6, 45, 47, 64, 67, 74, 77, 80] can scale up to a high number of threads. In contrast, in deleteMin operation, all threads compete for deleting the highest-priority element of the queue, thus competing for the same memory locations (similarly to queues and stacks), and creating a contention spot. In deleteMin-dominated workloads, concurrent priority queues typically incur high-contention and low parallelism. To achieve higher parallelism, relaxed priority queues have been proposed in the literature [2, 30], in which deleteMin operation returns an element among the first few (high-priority) elements of the priority queue. However, such NUMA-oblivious implementations are still inefficient in NUMA architectures, as we demonstrate in Section 4. Therefore, to improve performance in NUMA systems, NUMAaware implementations have been proposed [10,65].

We examine *NUMA-aware* and *NUMA-oblivious* concurrent priority queues with a wide variety of contention scenarios in NUMA architectures, and find that the performance of a priority queue implementation is becoming increasingly dependent on both the contention levels of the workload and the underlying computing platform. This is illustrated in Figure 1,

^{*} Christina Giannoula and Foteini Strati are co-primary authors.

which shows the throughput achieved by a *NUMA-oblivious* and a *NUMA-aware* priority queue using a 4-node NUMA system. Even though in a *insert*-dominated scenario, e.g., when having 100% *insert* operations, the *NUMA-oblivious* implementation achieves significant performance gains over the *NUMA-aware* one, when contention increases, i.e., the percentage of *deleteMin* operations increases, the *NUMA-oblivious* implementation incurs non-negligible performance slowdowns over the *NUMA-aware* priority queue. We conclude that none of the priority queues performs best across *all* contention workloads.

Our **goal** in this work is to design a concurrent priority queue that (i) achieves *the highest performance under all various contention scenarios, and (ii) performs best* even when the contention of the workload *varies* over time.

To this end, our contribution is twofold. First, we introduce *NUMA Node Delegation* (*Nuddle*), a generic technique to obtain *NUMA-aware* data structures, by effectively transforming *any* concurrent *NUMA-oblivious* data structure into the corresponding *NUMA-aware* implementation. In other words, *Nuddle* is a framework to wrap any *concurrent NUMA-oblivious* data structure and transform it into an efficient *NUMA-aware* one. *Nuddle* extends *ffwd* [65] by enabling multiple server threads, instead of only one, to execute operations in parallel on behalf of client threads. In contrast to *ffwd*, which aims to provide single threaded data structure performance, *Nuddle* targets data structures which are able to scale up to a number of threads such as priority queues.

Second, we propose *SmartPQ*, an adaptive concurrent priority queue that achieves the *highest* performance under *all* contention workloads and *dynamically* adapts itself over time between a *NUMA-oblivious* and a *NUMA-aware* algorithmic mode. *SmartPQ* integrates (i) *Nuddle* to *efficiently* switch between the two algorithmic modes with *very low* overhead, and (ii) a simple decision tree *classifier*, which predicts the bestperforming algorithmic mode given the expected contention levels of a workload.

Figure 2 presents an overview of *SmartPQ*, where we use the term *base algorithm* to denote *any* arbitrary concurrent *NUMA-oblivious* data structure. *SmartPQ* relies on three key ideas. First, client threads can execute operations using either *Nuddle* (*NUMA-aware* mode) or its underlying *NUMA-oblivious* base algorithm (*NUMA-oblivious* mode). Second, *SmartPQ* incorporates a decision making mechanism to decide upon transitions between the two modes. Third, *SmartPQ* exploits the fact that the actual underlying implementation of *Nuddle* is a *concurrent NUMA-oblivious* data structure. Client threads in both algorithmic modes access the data structure with the same concurrency strategy, i.e., with no actual change in the way data is accessed, and synchronization is implemented. Therefore, *SmartPQ* switches from one mode to another with *no* synchronization points between transitions.

We evaluate a wide range of contention scenarios and compare *Nuddle* and *SmartPQ* with state-of-the-art *NUMA-oblivious* [2, 47] and *NUMA-aware* [65] concurrent priority queues. We also evaluate *SmartPQ* using synthetic benchmarks that *dynamically* vary their contention workload over time. Our evaluation shows that *SmartPQ* adapts between its two algorithmic modes with negligible performance over-

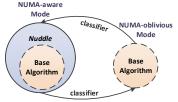


Figure 2: High-level overview of *SmartPQ*. *SmartPQ* dynamically adapts its algorithm to the contention levels of the workload based on the prediction of a simple classifier.

heads, and achieves the highest performance in *all* contention workloads with 87.9% success rate.

This paper makes the following contributions:

- We propose *Nuddle*, a generic technique to obtain *NUMA-aware* concurrent data structures.
- We design a simple classifier to predict the best-performing implementation among *NUMA-oblivious* and *NUMA-aware* priority queues given the contention levels of a workload.
- We propose SmartPQ, an adaptive concurrent priority queue that achieves the highest performance, even when contention varies over time.
- We evaluate Nuddle and SmartPQ with a wide variety of contention scenarios, and demonstrate that SmartPQ performs best over prior state-of-the-art concurrent priority queues.

2. NUMA Node Delegation (Nuddle)

2.1. Overview

NUMA Node Delegation (*Nuddle*) is a generic technique to obtain *NUMA-aware* data structures by automatically transforming *any* concurrent *NUMA-oblivious* data structure into an efficient *NUMA-aware* implementation. *Nuddle* extends *ffwd* [65], a client-server software mechanism which is based on the delegation technique [8, 38, 48, 57, 73].

Figure 3 left shows the high-level overview of ffwd, which has three key design characteristics. First, all operations performed by multiple client threads are delegated to one single dedicated thread, called server thread. Server thread performs operations in the data structure on behalf of its client threads. This way, the data structure remains in the memory hierarchy of a *single* NUMA node, avoiding non-uniform memory accesses to remote data. Second, ffwd eliminates the need for synchronization, since the shared data structure is no longer accessed by multiple threads: only a single server thread directly modifies the data structure, and therefore, ffwd uses a serial asynchronized implementation of the underlying data structure. Third, ffwd provides an efficient communication protocol between the server thread and client threads that minimizes cache coherence overheads. Specifically, ffwd reserves dedicated cache lines to exchange request and response messages between the client threads and sever thread. Multiple client threads are grouped together to minimize the response messages from the server thread: one response cache line is shared among multiple client threads belonging at the same client thread group. For more details, we refer the reader to the original paper [65].

Figure 3 right presents the high-level overview of *Nuddle*, which is based on three key ideas. First, *Nuddle* deploys

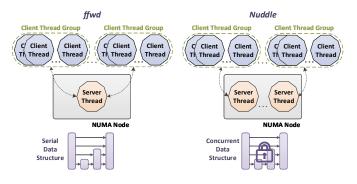


Figure 3: High-level design of ffwd [65] and Nuddle. Nuddle locates all server threads at the same NUMA node to design a NUMA-aware scheme, and associates each of them to multiple client thread groups. Nuddle uses the communication protocol proposed in ffwd [65].

multiple servers to perform operations on behalf of multiple client threads. Specifically, client threads are grouped in client thread groups, and each sever thread serves multiple client thread groups. This way, multiple server threads *concurrently* perform operations on the data structure, achieving high levels of parallelism up to a number of server threads. Second, Nuddle locates all server threads to the same NUMA node to keep the data structure in the memory hierarchy of one *single* NUMA node, and propose a *NUMA-aware* approach. Client threads can be located at any NUMA node. Third, since multiple servers can *concurrently* update the *shared* data structure, 18 **struct** server { *Nuddle* uses the *concurrent NUMA-oblivious* implementation (i.e., which includes synchronization primitives when accessing the shared data) of the underlying data structure to ensure correctness. Third, Nuddle employs the same client-server communication protocol with ffwd to carefully manage memory accesses and minimize cache coherence traffic and latency.

ffwd targets inherently serial data structures, whose concurrent performance cannot be better than that of single threaded performance. In contrast, Nuddle targets data structures that can scale up to a number of concurrent threads. Priority queue is a typical example of such a data structure. In insert operation, priority queue can scale up to multiple threads which can concurrently update the shared data. In contrast, deleteMin operation is inherently serial: at each time only one thread can update the shared data, since all threads compete for the highest-priority element of the queue. However, as we mentioned, in relaxed priority queues (e.g., SprayList [2]), even deleteMin operation can be parallelized to some extent.

2.2. Implementation Details

Figures 4, 5 and 6 present the code of a priority queue implementation using *Nuddle*. We denote with red color the core operations of the *base algorithm*, which is used as the underlying concurrent *NUMA-oblivious* implementation of *Nuddle*. Note that even though in this work we focus on priority queues, *Nuddle* is a *generic* framework for any type of concurrent data structure.

Helper Structures. *Nuddle* includes three *helper* structures (Figure 4), which are needed for client-server communication. First, the main structure of *Nuddle*, called *struct nuddle_pq*, wraps the *base algorithm* (*nm_oblv_set*), and

includes a few additional fields, which are used to associate client thread groups to server threads in the initialization step. Second, each client thread has its own <code>struct client</code> structure with a dedicated request and a dedicated response cache line. The request cache line is exclusively written by the client thread and read by the associated server thread, while the response cache line is exclusively written by the server thread and read by all client threads that belong in the same client thread group. Third, each server thread has its own <code>struct server</code> structure that includes an array of requests (<code>my_clients</code>), each of them is shared with a client thread, and an array of responses (<code>my_responses</code>), each of them is shared with all client threads of the <code>same</code> client thread group.

```
#define cache_line_size 128
typedef char cache_line[cache_line_size];

struct nuddle_pq {
    mm_oblv_set *base_pq;
    int servers, groups, clnt_per_group;
    int server_cnt, clients_cnt, group_cnt;
    cache_line *requests[groups][clnt_per_group];
    cache_line *responses[groups];
    lock *global_lock;
};

struct client {
    cache_line *request, *response;
    int clnt_pos;
    int clnt_pos;
};

struct server {
    mm_oblv_set *base_pq;
    cache_line *my_clients[], *my_responses[];
    int my_groups, clnt_per_group;
};
```

Figure 4: Helper structures of Nuddle.

Initialization Step. Figure 5 describes the initialization functions of Nuddle. initPQ() initializes (i) the underlying data structure using the corresponding function of the base algorithm (line 25), and (ii) the additional fields of struct nuddle_pq. For this function, programmers need to specify the number of server threads and the maximum number of client threads to properly allocate cache lines needed for communication among them. Programmers also specify the size of the client thread group (line 27), which is typically 7 or 15, if the cache line is 64 or 128 bytes, respectively. As explained in *ffwd* [65], assuming 8-byte return values, a dedicated 64-byte (or 128-byte) response cache line can be shared between up to 7 (or 15) client threads, because it also has to include one additional toggle bit for each client thread. After initializing struct nuddle_pq, each running thread calls either initClient() or initServer() depending on its role. Each thread initializes its own helper structure (struct client or struct server) with request and response cache lines of the corresponding shared arrays of struct nuddle_pq. Server threads undertake client thread groups with a round robin fashion, such that the load associated with client threads is balanced among them. In function initServer(), it is the programmer's responsibility to properly pin software server threads to hardware contexts (line 56), such that server threads are located in the *same* NUMA node,

and the programmer fully benefits from the *Nuddle* technique. Moreover, given that client threads of the *same* client thread group share the same response cache line, the programmer could pin client threads of the *same* client thread group to hardware contexts of the *same* NUMA node to minimize cache coherence overheads. Finally, since the request and response arrays of *struct nuddle_pq* are *shared* between all threads, a global lock is used when updating them to ensure mutual exclusion.

```
23 struct nuddle_pq *initPQ(int servers, int
       max_clients) {
24
     struct nuddle_pq *pq = allocate_nuddle_pq();
2.5
     __base_init(pq->base_pq);
26
     pq->servers = servers;
27
     pq->clnt_per_group = client_group(
       cache_line_size);
28
     pq->groups = (max_clients +
29
       pq->clnt_per_group-1) / pq->clnt_per_group;
30
     pq->server_cnt = 0;
31
    pq->client_cnt = 0;
32
     pq->group_cnt = 0;
33
     pq->requests = malloc(groups * clnt_per_group);
34
    pq->responses = malloc(groups);
35
     init_lock(pq->global_lock);
36
     return pq;
37 }
38
  struct client *initClient(struct nuddle_pq *pq) {
39
40
     struct client *cl = allocate_client();
41
     acquire_lock(pq->global_lock);
42
     cl->request = &(pq->requests[group_cnt][
       clients_cnt]);
43
     cl->response = &(pq->responses[group_cnt]);
44
     cl->pos = pq->client_cnt;
45
     pq->client_cnt++;
46
     if (pq->client_cnt % pq->clnt_per_group == 0) {
47
       pq->clients_cnt = 0;
48
       pq->group_cnt++;
49
50
     release_lock(pq->global_lock);
51
     return cl:
52 }
53
  struct server *initServer(struct nuddle pg *pg.
       int core)
55 {
     set_affinity(core);
56
57
     struct server *srv = allocate_server();
58
     srv->base_pq = pq->base_pq;
59
     srv->my\_groups = 0;
60
     srv->clnt_per_group = pq->clnt_per_group;
61
     acquire_lock(pq->global_lock);
     int j = 0;
63
     for(i = 0; i < pq \rightarrow groups; i++)
64
       if(i % pq->servers == pq->server_cnt) {
65
         srv->my_clients[j] = pq->requests[i][0..
       ar clntl:
66
         srv->my_responses[j++] = pq->responses[i];
67
         srv \rightarrow my\_groups ++;
68
69
     pq->server_cnt++;
70
     release_lock(pq->global_lock);
71
     return srv;
72 }
```

Figure 5: Initialization functions of Nuddle.

Main API. Figure 6 shows the core functions of *Nuddle*, where we omit the corresponding functions for *deleteMin* operation, since they are very similar to that of *insert* operation. Both *insert* and *deleteMin* operations of *Nuddle* have similar

API with the classic API of prior state-of-the-art priority queue implementations [2, 45, 47, 67]. However, we separate the corresponding functions for client threads and server threads. A client thread writes its request to a dedicated request cache line (line 75) and then waits for the server thread's response. In contrast, a server thread directly executes operations in the data structure using the core functions of the base algorithm (line 82). Moreover, a server thread can serve client threads using the serve_requests() function. A server thread iterates over its own client thread groups and executes the requested operations in the data structure. The server thread buffers individual return values for clients to a local cache line (resp in lines 92 and 94) until it finishes processing all requests for the current client thread group. Then, it writes all responses to the shared response cache line of that client thread group (line 96), and proceeds to its next client thread group.

```
73 int insert_client(struct client *cl,
                             int key, int64_t value)
74
75
     cl->request = write_req("insert", key, value);
76
     while (cl->response[cl->pos] == 0);
77
     return cl->response[cl->pos];
78 }
79
  int insert_server(struct server *srv,
                             int key, int64_t value)
81
  {
82
     return __base_insrt(srv->base_pq, key, value);
83 }
84
85
  void serve_requests(struct server *srv) {
     for(i = 0; i < srv->mygroups; i++) {
86
87
       cache_line resp;
88
       for(j = 0; j < srv->clnt_per_group; j++) {
89
         key = srv->my_clients[i][j].key;
90
         value = srv->my_clients[i][j].value;
91
         if (srv->my_clients[i][j].op == "insert")
92
           resp[j] = __base_insrt(srv->base_pq, key,
       value);
93
         else if (srv->my_clients[i][j].op == "
       deleteMin")
94
           resp[j] = __base_delMin(srv->base_pq);
95
96
       srv->my_responses[i] = resp;
97
     }
98 }
```

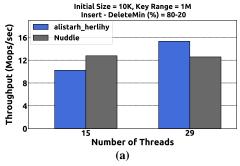
Figure 6: Functions used by server threads and client threads to perform operations using *Nuddle*.

3. SmartPO

We propose *SmartPQ*, an adaptive concurrent priority queue which tunes itself by *dynamically* switching between *NUMA-oblivious* and *NUMA-aware* algorithmic modes, in order to perform best in *all* contention workloads and at *any* point in time, even when contention varies over time.

Designing an adaptive priority queue involves addressing two major challenges: (i) how to switch from one algorithmic mode to the other with *low overhead*, and (ii) *when* to switch from one algorithmic mode to the other.

To address the first challenge, we exploit the fact that the actual underlying implementation of *Nuddle* is a *concurrent NUMA-oblivious* implementation. We select *Nuddle*, as the *NUMA-aware* algorithmic mode of *SmartPQ*, and its underly-



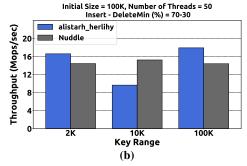


Figure 7: Throughput achieved by *Nuddle* (using 8 server threads) and its underlying *NUMA-oblivious base algorithm*, i.e., *alistarh_herlihy* [2,34], when we vary (a) the number of threads that perform operations in the shared data structure, and (b) the key range of the workload.

ing base algorithm, as the NUMA-oblivious algorithmic mode of SmartPQ. Threads can perform operations in the data structure using either Nuddle or its underlying base algorithm, with no actual change in the way data is accessed. As a result, SmartPQ can switch between the two algorithmic modes without needing a synchronization point between transitions, and without violating correctness.

To address the second challenge, we design a simple decision tree classifier (Section 3.1.2), and train it to select the best-performing algorithmic mode between *Nuddle*, as the *NUMA-aware* algorithmic mode of *SmartPQ*, and its underlying *base algorithm*, as the *NUMA-oblivious* mode of *SmartPQ*. Finally, we add a *lightweight* decision making mechanism in *SmartPQ* (Section 3.2) to dynamically tune itself over time between the two algorithmic modes. We describe more details in next sections.

3.1. Selecting the Algorithmic Mode

3.1.1. The Need for a Machine Learning Approach

Selecting the best-performing algorithmic mode can be solved in various ways. For instance, one could take an empirical exhaustive approach: measure the throughput achieved by the two algorithmic modes for all various contention scenarios on the target NUMA system, and then use the algorithmic mode that achieves the highest throughput on future runs of the same contention workload on the target NUMA system. Even though this is the most accurate method, it (i) incurs substantial overhead and effort to sweep over all various contention workloads, and (ii) would need a large amount of memory to store the best-performing algorithmic mode for all various scenarios. Furthermore, it is not trivial to construct a statistical model to predict the best-performing algorithmic mode, since the performance of an algorithm is also affected by the characteristics of underlying computing platform. Figure 7 summarizes these observations by comparing *Nuddle* with its underlying base algorithm in a 4-node NUMA system. For the base algorithm, we use alistarh_herlihy priority queue [2, 34], since this is the NUMA-oblivious implementation that achieves the highest performance, according to our evaluation (Section 4).

Figure 7a demonstrates that the best-performing algorithmic mode depends on multiple parameters, such as the number of threads that perform operations in the shared data structure. We find that the algorithmic also depends on the size of the

data structure, and the operation workload, i.e., the percentage of *insert/deleteMin* operations. Specifically, when the number of threads increases, we may expect that the performance of the *NUMA-oblivious alistarh_herlihy* degrades due to higher contention. In contrast, with 80% *insert* operations when increasing the number of threads to 29, *alistarh_herlihy* outperforms *Nuddle*. This is because the size of the priority queue and the range of keys used in the workload are relatively large, while the percentage of *deleteMin* operations is low. In this scenario, threads may not compete for the same elements, working on different parts of the data structure, and thus, the *NUMA-oblivious alistarh_herlihy* achieves higher throughput compared to the *NUMA-aware Nuddle*.

Figure 7b demonstrates that the best-performing algorithmic mode cannot be straightforwardly predicted, and also depends on the characteristics of the workload and of the underlying hardware. In insert-dominated workloads, as the key range increases, threads may update different parts of the shared data structure. We might, thus, expect that after a certain point of increasing the key range, the NUMA-oblivious alistarh_herlihy will always outperform Nuddle, since the contention decreases. However, we note that, even though the performance of *Nuddle* remains constant, as expected, the performance of alistarh herlihy highly varies as the key range increases due to the hyperthreading effect. When using more than 32 threads, hyperthreading is enabled in our NUMA system (Section 4). The hyperthreading pair of threads shares the L1 and L2 caches, and thus, these threads may either thrash or benefit from each other depending on the characteristics of L1 and L2 caches (e.g., size, eviction policy), and the elements accessed in each operation.

Considering the aforementioned non-straightforward behavior, we resort to a machine learning approach as the basis of our prediction mechanism.

3.1.2. Decision Tree Classifier

We formulate the selection of the algorithmic mode as a classification problem, and leverage supervised learning techniques to train a simple classifier to predict the best-performing algorithmic mode for each contention workload. For our classifier, we select decision trees, since they are commonly used in classification models for multithreaded workloads [3, 17, 19, 21, 51, 59, 69, 72], and incur low training and inference overhead. Moreover, they are easy to interpret and thus, be incorporated to our proposed priority queue (Sec-

tion 3.2). We generate the decision tree classifier using the scikit-learn machine learning toolkit [56].

1) Class Definition: We define the following classes: (a) the **NUMA-oblivious** class that stands for the NUMA-oblivious algorithmic mode, (b) the *NUMA-aware* class that stands for the NUMA-aware algorithmic mode, and (c) the neutral class that stands for a tie, meaning that either a NUMA-aware or a NUMA-oblivious implementation can be selected, since they achieve similar performance. We include a neutral class for two reasons: (i) when using only one socket of a NUMA system, NUMA-aware implementations deliver similar throughput with NUMA-oblivious implementations, and (ii) in an adaptive data structure, which dynamically switches between the two algorithmic modes, we want to configure a transition from one algorithmic mode to another to occur when the difference in their throughput is relatively high, i.e., greater than a certain threshold. Otherwise, the adaptive data structure might continuously oscillate between the two modes, without delivering significant performance improvements or even causing performance degradation.

2) Extracted Features: Table 1 explains the four features of the contention workload which are used in our classifier targeting priority queues. We assume that the contention workload is known a priori, and thus, we can easily extract the features needed for classification. Section 5 discusses how to on-the-fly extract these features.

Feature	Definition
#Threads	The number of active threads that perform operations in the data structure
Size	The current size of the priority queue
Key_range % insert/deleteMin	The range of keys used in the workload The percentage of <i>insert/deleteMin</i> operations

Table 1: The features of the contention workload which are used for classification.

3) Generation of Training Data: To train our classifier, we develop microbenchmarks, in which threads repeatedly execute random operations on the priority queue for 5 seconds. We select Nuddle, as the NUMA-aware implementation, and alistarh_herlihy, as its underlying NUMA-oblivious implementation, since this is the best-performing NUMA-oblivious priority queue (Section 4). We use a variety of values for the features needed for classification (Table 1). Our training data set consists of 5525 different contention workloads. Finally, we pin software threads to hardware contexts of the evaluated NUMA system in a round-robin fashion, and thus, the classifier is trained with this thread placement. We leave the exploration of the thread placement policy for future work.

4) Labeling of Training Data: Regarding the labeling of our training data set, we set the threshold for tie between the two algorithmic modes to an empirical value of 1.5 Million operations per second. When the difference in throughput between the two algorithmic modes is less than this threshold, the neutral class is selected as label. Otherwise, we select the class that corresponds to the algorithmic mode that achieves the highest throughput.

The final decision tree classifier has only 180 nodes, half of which are leaves. It has a very low depth of 8, that is the length

of the longest path in the tree, and thus, a *very low* traversal cost (2-4 ms in our evaluated NUMA system).

3.2. Implementation Details

Figure 8 presents the modified code of Nuddle adding the decision making mechanism (using green color) to implement SmartPQ. We extend the main structure of Nuddle, renamed to struct smartpq, by adding an additional field, called algo, to keep track the current algorithmic mode, (either NUMA-oblivious or NUMA-aware). Similarly, struct client and struct server structures are extended with an additional algo field (e.g., line 111), which is a pointer to the algo field of struct smartpg. Each active thread initializes this pointer either in initClient() or initServer() depending on its role (e.g., line 119). This way, all threads share the same algorithmic mode at any point in time. In struct client, we also add a pointer to the shared data structure (line 110), which is used by client threads to directly perform operations in the data structure in case of NUMA-oblivious mode. Specifically, we modify the core functions of client threads, i.e., insert_client() and deleteMin_client(), such that client threads either directly execute their operations in the data structure (e.g., line 126), or delegate them to server threads (e.g., line 127-128), with respect to the current algorithmic mode. In contrast, the core functions of server threads do not need any modification. Finally, we wrap the code of serve requests function, i.e., the lines 86-97 of Figure 6, with an if/else statement on the algo field (lines 133, 146 in Fig. 8), such that server threads poll at client threads' requests only in NUMA-aware mode. In NUMA-oblivious mode, serve requests function returns without doing nothing. This way, programmers do not need to take care of calls on this function in their code, when the NUMA-oblivious mode is selected.

The decisionTree() function describes the interface with our proposed decision tree classifier, where the input arguments are associated with its features. In frequent time lapses, one or more threads may call this function to check if a transition to another algorithmic mode is needed. If this is the case, the algo field of struct smartpq is updated (line 154 in Fig. 8), and SmartPQ switches algorithmic mode, i.e., all active threads start executing their operations using the new algorithmic mode. If the classifier predicts the neutral class (line 153), the algo field is not updated, and thus SmartPQ remains at the currently selected algorithmic mode.

4. Experimental Evaluation

In our experimental evaluation, we use a 4-socket Intel Sandy Bridge-EP server equipped with 8-core Intel Xeon CPU E5-4620 processors providing a total of 32 physical cores and 64 hardware contexts. The processor runs at 2.2GHz and each physical core has its own L1 and L2 cache of sizes 64KB and 256KB, respectively. A 16MB L3 cache is shared by all cores in a NUMA socket and the RAM is 256GB. We use GCC 4.9.2 with -O3 optimization flag enabled to compile all implementations.

Our evaluation includes the following concurrent priority queue implementations:

```
99 struct smartpq {
100
     nm_oblv_set *base_pq;
101
      int servers, groups, clnt_per_group;
      int server_cnt, clients_cnt, group_cnt;
102
103
      cache_line *requests[groups][clnt_per_group];
      cache_line *responses[groups];
104
105
      lock *global_lock;
106
      int *algo; // 1: NUMA-oblivious (default),
        2: NUMA-aware
107 };
108
109 struct client {
110
     nm_oblv_set *base_pq;
111
      int *algo;
112
      cache_line *request, *response;
113
      int clnt_pos;
114 };
115
116 struct client *initClient(struct smartpq *pq) {
      ... lines 40-49 of Fig. 5 ...
117
      cl->base_pq = pq->base_pq;
118
119
      cl->algo = &(pq->algo);
120
      release_lock(pq->global_lock);
121
      return cl;
122 }
123
124 int insert_client(struct client *cl,
                       int key, float value) {
125
      if(*(cl->algo) == 1) {
        return __base_insert(cl->base_pq,key,value);
126
127
      } else { // *(cl->algo) == 2
128
        ... lines 75-77 of Fig. 6 ...
129
130 }
131
132 void serve_requests(struct server *srv) {
133
     if(*(srv->algo) == 2){
134
        for(i = 0; i < srv->mygroups; i++) {
135
          cache_line resp;
136
          for(j = 0; j < srv->clnt_per_group; j++) {
137
            key = srv->my_clients[i][j].key;
138
            value = srv->my_clients[i][j].value;
139
            if (srv->my_clients[i][j].op == "insert")
140
              resp[j] = __base_insrt(srv->base_pq, key,
        value);
141
            else if (srv->my_clients[i][j].op == "
        deleteMin")
142
              resp[j] = __base_delMin(srv->base_pq);
143
          }
144
          srv->my_responses[i] = resp;
145
        }
146
      } else
147
        return;
148 }
149
150 void decisionTree(struct server /
                    struct client *str, int nthreads,
                      int size, int key_range,
                     double insert\_deleteMin) {
151
      int algo = 0;
152
      ... code for decision tree classifier ...
153
      if (algo != 0) // 0: neutral
154
        *(str->algo) = algo;
155 }
```

Figure 8: The modified code of *Nuddle* adding the decision making mechanism to implement *SmartPQ*.

- alistarh_fraser [2, 24]: A NUMA-oblivious, relaxed priority queue [2] based on Fraser's skip-list [24] available at ASCYLIB library [16].
- alistarh_herlihy [2,34]: A NUMA-oblivious, relaxed priority queue [2] based on Herlihy's skip-list [34] available at ASCYLIB library [16].
- lotan_shavit [47]: A NUMA-oblivious priority queue available at ASCYLIB library [16].
- ffwd [65]: A NUMA-aware priority queue based on the delegation technique [8, 38, 48, 57, 73], which includes only one server thread to perform operations on behalf of all client threads.
- Nuddle: Our proposed NUMA-aware priority queue, which uses alistarh_herlihy as the underlying base algorithm.
- SmartPQ: Our proposed adaptive priority queue, which uses Nuddle as the NUMA-aware mode, and alistarh_herlihy as the NUMA-oblivious base algorithm.

We evaluate the concurrent priority queue implementations in the following way:

- Each execution lasts 5 seconds, during which each thread performs randomly chosen operations. We also tried longer durations and got similar results.
- Between consecutive operations in the data structure each thread executes a delay loop of 25 pause instructions. This delay is intentionally added in our benchmarks to better simulate a real-life application, where operations in the data structure are intermingled with other instructions in the application.
- At the beginning of each run, the priority queue is initialized with elements the number of which is described at each figure.
- Each software thread is pinned to a hardware context. Hyperthreading is enabled when using more than 32 software threads. When exceeding the number of available hardware contexts of the system, we oversubscribe software threads to hardware contexts.
- We pin the first 8 threads to the first NUMA node, and consecutive client thread groups of 7 client threads each, to NUMA nodes in a round-robin fashion.
- In NUMA-oblivious implementations, any allocation needed in the operation is executed on demand, and memory affinity is determined by the first touch policy.
- In NUMA-aware implementations, since our NUMA system has 64-byte cache lines, the response cache line is shared between up to 7 client threads, using 8-byte return values.
- In Nuddle, the first 8 threads represent server threads. Server threads repeatedly execute the serve_requests function, and then a randomly chosen operation until time is up.
- We have disabled the automatic Linux Balancing [42] to get consistent and stable results.
- All reported results are the average of 10 independent executions with no significant variance.

4.1. Throughput of *Nuddle*

Figure 9 presents the throughput achieved by concurrent priority queue implementations for various sizes and operation workloads. *NUMA-aware* priority queue implementations, i.e., *ffwd* and *Nuddle*, achieve high throughput in *deleteMin-*dominated workloads: *Nuddle* performs best in

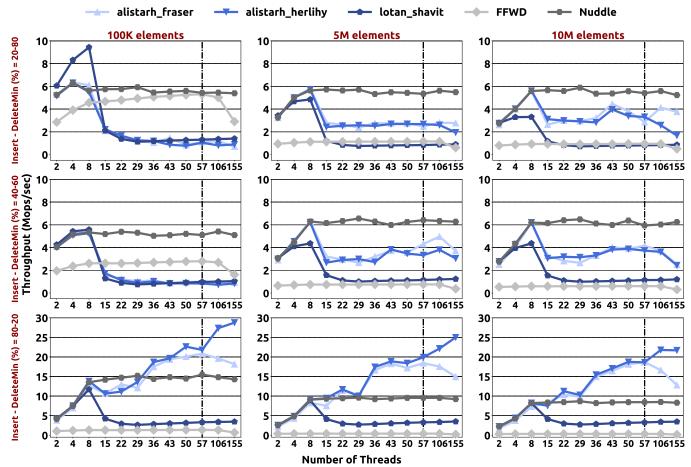


Figure 9: Throughput of concurrent priority queue implementations. The columns show different priority queue sizes using the key range of double the elements of each size. The rows show different operation workloads. The vertical line in each plot shows the point after which we oversubscribe software threads to hardware contexts.

all deleteMin-dominated workloads, while ffwd outperforms NUMA-oblivious implementations in the small-sized priority queues (e.g., 100K elements). In large-sized priority queues, insert operations have a larger impact on the total execution time (due to a longer traversal), and thus Nuddle and NUMAoblivious implementations perform better than ffwd, since they provide higher thread-level parallelism. Note that ffwd has single-threaded performance, since at any point in time only one (server) thread performs operations in the data structure. Moreover, as it is expected, the performance of both ffwd and Nuddle saturates at the number of server threads used (e.g., 8 server threads for *Nuddle*) to perform operations in the data structure. Finally, we note that the communication between server and client threads used in NUMA-aware implementations has negligible overhead; when the number of client threads increases, even though the communication traffic over the interconnect increases, there is no performance drop. Overall, we conclude that *Nuddle* achieves the highest throughput in all deleteMin-dominated workloads, and is the most efficient NUMA-aware approach, since it provides high thread-level parallelism.

On the other hand, *NUMA-oblivious* implementations incur high performance degradation in *high-contention* scenarios, such as *deleteMin*-dominated workloads, when using more than one NUMA node (i.e., after 8 threads). As already dis-

cussed in prior works [5, 15, 25, 48, 54, 81], the non-uniformity in memory accesses and cache line invalidation traffic significantly affects performance in high-contention scenarios. In insert-dominated workloads, which incur lower contention, even though *lotan_shavit* priority queue still incurs performance degradation when using more than one NUMA nodes of the system, the relaxed NUMA-oblivious implementations, i.e., alistarh fraser and alistarh herlihy priority queues, achieve high scalability. This is because relaxed priority queues decrease both (i) the contention among threads, and (ii) the cache line invalidation traffic: the deleteMin operation returns (with a high probability) an element among the first few (highpriority) elements of the queue, and thus, threads do not frequently compete for the same elements. Finally, we observe that alistarh_herlihy priority queue achieves higher performance benefits over alistarh_fraser priority queue, when we oversubscribe software threads to the available hardware contexts of our system. Overall, we find that in *insert*-dominated workloads, the relaxed NUMA-oblivious implementations significantly outperform the NUMA-aware ones.

To sum up, we conclude that there is no one-size-fits-all solution, since none of the priority queues performs best across *all* contention workloads. *Nuddle* achieves the highest throughput in high contention scenarios, while *alistarh_herlihy* performs best in low and medium contention scenarios. It is thus

desirable to design a new approach for a concurrent priority queue to perform best under *all* various contention scenarios.

4.2. Throughput of SmartPQ

4.2.1. Classifier Accuracy

We evaluate the efficiency of our proposed classifier (Section 3.1.2) using two metrics: (i) accuracy, and (ii) misprediction cost. First, we define the accuracy of the classifier as the percentage of *correct* predictions, where a prediction is considered correct, if the classifier predicts the algorithmic mode (either the NUMA-aware Nuddle or the NUMAoblivious alistarh herlihy) that achieves the best performance between the two. We use a test set of 10780 different contention workloads, where we randomly select the values of the features in each workload. In the above test set, our classifier has 87.9% accuracy, i.e., it mispredicts 1300 times in 10780 different contention workloads. Second, we define the misprediction cost as the performance difference between the correct (best-performing) algorithmic mode and the wrong predicted mode normalized to the performance of the wrong predicted mode. Specifically, assuming the throughput of the wrong predicted and correct (best-performing) algorithmic mode is Y and X respectively, the misprediction cost is defined as ((X - Y)/Y) * 100%. In 1300 mispredicted workloads, the geometric mean of misprediction cost for our classifier is 30.2%. We conclude that the proposed classifier has high accuracy, and in case of misprediction, incurs low performance degradation.

4.2.2. Varying the Contention Workload

We present the performance benefit of SmartPQ in synthetic benchmarks, in which we vary the contention workload over time, and compare it with Nuddle and its underlying base algorithm, i.e., alistarh_herlihy priority queue. In all benchmarks, we change the contention workload every 25 seconds. In SmartPQ, we set one dedicated sever thread to call the decision tree classifier every second, in order to check if a transition to another algorithmic mode is needed. Figure 10 and Figure 11 show the throughput achieved by all three schemes, when we vary one and multiple features in the contention workload, respectively. Table 2 and Table 3 show the features of the workload as they vary during the execution for the benchmarks evaluated in Figure 10 and Figure 11, respectively. Note that the current size of the priority queue changes during the execution due to successful insert and *deleteMin* operations.

We make three observations. First, as already shown in Section 4.1, there is no one-size-fits-all solution, since neither *Nuddle* nor *alistarh_herlihy* performs best across all various contention workloads. For instance, in Figure 10b, even though the performance of *Nuddle* remains constant, it outperforms *alistarh_herlihy*, when having 15 running threads, i.e., using 2 NUMA nodes of the system. Second, we observe that *SmartPQ* successfully adapts to the best-performing algorithmic mode, and performs best in *all* contention scenarios. In Figure 11, even when multiple features in the contention workload vary during the execution, *SmartPQ* outperforms *alistarh_herlihy* and *Nuddle* by 1.87× and 1.38× on average, respectively. Note that any of the contention workloads

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	1149	100K	50	75-25
25	812	2K	50	75-25
50	485	1M	50	75-25
75	2860	10K	50	75-25
100	2256	50M	50	75-25

(a) Varying the key range in the workload.

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	1166	20M	57	65-35
25	15567	20M	29	65-35
50	15417	20M	15	65-35
75	15297	20M	43	65-35
100	15346	20M	15	65-35

(b) Varying the number of threads that perform operations in the data structure.

Time (sec)	Current Size	Key Range	Number of Threads	Insert - Delete $\operatorname{Min}\left(\%\right)$
0	1M	5M	22	50-50
25	140	5M	22	100-0
50	7403	5M	22	30-70
75	962	5M	22	100-0
100	8236	5M	22	0-100

(c) Varying the percentage of insert/deleteMin operations.

Table 2: Features of the contention workload for benchmarks evaluated in Figure 10. We use bold font on the features that change in each execution phase.

evaluated in Figures 10 and 11 belongs in the training data set used for training our classifier. Third, we note that the decision making mechanism of SmartPQ has very low performance overheads. Across all evaluated benchmarks, SmartPQ achieves only up to 5.3% performance slowdown (i.e., when using a range of 50M keys in Figure 10a) over the corresponding baseline implementation (alistarh_herlihy priority queue). Note that since the proposed decision tree classifier has very low traversal cost (Section 3.1.2), we intentionally set a frequent time interval (i.e., one second) for calling the classifier, such that SmartPQ detects the contention workload change on time, and quickly adapts itself to the best-performing algorithmic mode. We also tried large time intervals, and observed that SmartPQ slightly delays to detect the contention workload change, thus achieving lower throughput in the transition points.

Overall, we conclude that *SmartPQ* performs best across *all* contention workloads and at *any* point in time, and incurs negligible performance overheads over the corresponding baseline implementation.

5. Discussion & Future Work

In Section 3.1.2, we assume that the contention workload is known a priori to extract the features needed for classification. To extract these features *on-the-fly*, and *dynamically* detect when contention changes, the main structure of *SmartPQ*, i.e., *struct smartpq*, needs to be enriched with additional fields to keep track of workload statistics (e.g., the number of completed *insert/deleteMin* operations, the number of active threads that perform operations on the data structure, the minimum and/or maximum key that has been requested so far). Active threads that perform operations on the data structure could atomically update these statistics. In frequent time lapses, either a background thread or an active thread could extract the features needed for classification based on the work-

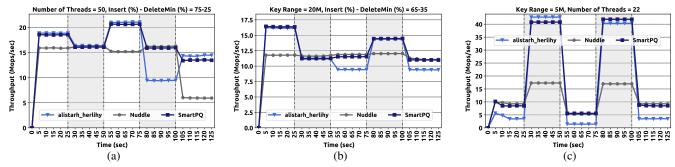


Figure 10: Throughput achieved by *SmartPQ*, *Nuddle* and its underlying *base algorithm* (*alistarh_herlihy*), in synthetic benchmarks, in which we vary a) the key range, b) the number of threads that perform operations in the data structure, and c) the percentage of *insert/deleteMin* operations in the workload.

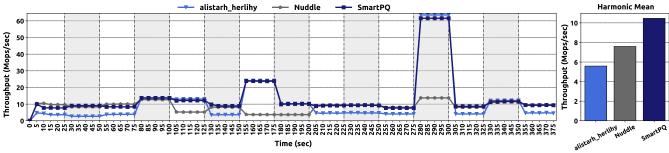


Figure 11: Throughput achieved by SmartPQ, Nuddle and its underlying base algorithm (alistarh_herlihy), in synthetic benchmarks, in which we vary multiple features in the contention workload.

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	1M	10M	57	50-50
25	26	10M	36	70-30
50	12	20M	36	50-50
75	79	20M	36	80-20
100	29K	20M	50	80-20
125	319K	100M	50	50-50
150	13	100M	57	50-50
175	524K	100M	22	100-0
200	524K	100M	22	50-50
225	1142	100M	22	50-50
250	463	200M	57	0-100
275	253	200M	57	100-0
300	33K	20M	57	0-100
325	142	20M	29	80-20
350	25K	20M	29	50-50

Table 3: Features of the contention workload for benchmarks evaluated in Figure 11. We use bold font on the features that change in each execution phase.

load statistics, and call the classifier to predict if a transition to another algorithmic mode is needed. Finally, an additional parameter could be also added in *SmartPQ* to configure how often to collect workload statistics.

In our experimental evaluation, we pin server threads on a single NUMA node and client threads on all nodes. We have chosen to do so (i) for simplicity, given that this approach fits well with our microbenchmark-based evaluation, and (ii) because this is par with prior works on concurrent data structures [2, 7, 9–11, 14, 16, 22, 27, 37, 46, 65–67, 71, 78]. In a real-life scenario, where *SmartPQ* is used as a part of an application, client threads do *not* need to be pinned in hardware contexts, and they can be allowed to run in any core of the system. However, for our approach to be meaningful server threads need to be limited on a single NUMA node. This can easily be done by creating the server threads when *SmartPQ*

is initialized, and pinning them to hardware contexts that are located at the same NUMA node. In this case, server threads are background threads that only accept and serve requests from various client threads, which are part of the high-level application.

Finally, even though we focus on a microbenchmark-based evaluation to cover a *wide variety* of contention scenarios, it is one of our future directions to explore the efficiency of *SmartPQ* in real-life applications, such as web servers [28,63], graph traversal applications [12,67] and scheduling in operating systems [1]. As future work, we also aim to investigate the applicability of our approach in other data structures, that may have similar behavior with priority queues (e.g., skip lists, search trees), and extend our proposed classifier (e.g., adding more features) to cover a variety of NUMA CPU-centric systems with different architectural characteristics.

6. Related Work

To our knowledge, this is the first work to propose an adaptive priority queue for NUMA systems, which performs best under *all* various contention workloads, and even when contention varies over time. We briefly discuss prior work.

Concurrent Priority Queues. A large corpus of work proposes concurrent algorithms for priority queues [2, 6, 9, 45, 47, 62, 64, 66–68, 74, 77, 80], or generally for skip lists [11, 13, 18, 23, 24, 34, 35, 46, 61]. Recent works [45, 47] designed lock-free priority queues that separate the logical and the physical deletion of an element to increase parallelism. Alistarh et al. [2] design a relaxed priority queue, called *SprayList*, in which *deleteMin* operation returns with a high probability, an element among the *first* $\mathcal{O}(p \log 3p)$ elements of the priority queue, where p is the number of threads.

Sagonas et al [66] design a contention avoiding technique, in which deleteMin operation returns the highest-priority element of the priority queue under *low* contention, while it enables relaxed semantics when high contention is detected. Specifically, under high-contention a few *deleteMin* operations are queued, and later several elements are deleted from the head of the queue at once via a combined deletion operation. Heidarshenas et al. [30] design a novel architecture for relaxed priority queues. These prior approaches are NUMA-oblivious implementations. Thus, in NUMA systems, they incur significant performance degradation in high-contention scenarios (e.g., deleteMin-dominated workloads in Section 4.1). In contrast, Calciu et al. [9] propose a NUMA-friendly priority queue employing the combining and elimination techniques. Elimination allows the complementary operations, i.e., *insert* with deleteMin, to complete without updating the data structure, while combining is a technique similar to the delegation technique [8, 38, 48, 57, 73] of *Nuddle* and *ffwd* [65]. Finally, Daly et al. [14] propose an efficient technique to obtain NUMAaware skip lists, which however, can only be applied to skip list-based data structures. In contrast, Nuddle is a generic technique to obtain *NUMA-aware* data structures.

Black-Box Approaches. Researchers have proposed blackbox approaches: any data structure can be made wait-free or NUMA-aware without effort or knowledge on parallel programming or NUMA architectures. Herlihy [33] provides a universal method to design wait-free implementations of any sequential object. However, this method remains impractical due to high overheads. Hendler et al. [31] propose flat combining; a technique to reduce synchronization overheads by executing multiple client threads' requests at once. Despite significant improvements [32], this technique provides high performance only for a few data structures (e.g., synchronous queues). ffwd [65] is black-box approach, which uses the delegation technique [8, 38, 48, 57, 73] to eliminate cache line invalidation traffic over the interconnect. However, ffwd is limited to single threaded performance. Calciu et al. [10] propose a black-box technique, named Node Replication, to obtain concurrent NUMA-aware data structures. In Node Replication, every NUMA node has replicas of the shared data structure, which are synchronized via a shared log. Although ffwd and *Node Replication* are generic techniques to obtain *NUMA*aware data structures, similarly to Nuddle, both of them use a serial asynchronized implementation as the underlying base algorithm. Thus, if they are used as the NUMA-aware algorithmic mode in an adaptive data structure, which dynamically switches between a NUMA-oblivious and a NUMA-aware mode, both ffwd and Node Replication need a synchronization point between transitions to ensure correctness. Consequently, they would incur high performance overheads, when transitions between algorithmic modes happen at a non-negligible frequency.

Machine learning in Data Structures. Even though machine learning is widely used to improve performance in many emerging applications [3, 4, 17, 21, 26, 41, 44, 50, 51, 53, 58, 69, 82], there is a handful of works [20, 40] that leverage machine learning to design *highly-efficient* concurrent data

structures. Recently, Eastep et al. [20] use reinforcement learning to on-the-fly tune a parameter in the flat combining technique [31, 32], which is used in skip lists and priority queues. Kraska et al. [40] demonstrate that machine learning models can be trained to predict the position or existence of elements in key-value lookup sets, and discuss under which conditions learned index models can outperform the traditional indexed data structures (e.g., B-trees).

7. Conclusion

We propose SmartPQ, an adaptive concurrent priority queue for NUMA architectures, which performs best under all various contention scenarios, and even when contention varies over time. SmartPQ has two key components. First, it is built on top of *Nuddle*; a generic low-overhead technique to obtain efficient NUMA-aware data structures using any concurrent NUMA-oblivious implementation as its backbone. Second, SmartPQ integrates a lightweight decision making mechanism, which is based on a simple decision tree classifier, to decide when to switch between Nuddle, i.e., a NUMA-aware algorithmic mode, and its underlying base algorithm, i.e., a NUMA-oblivious algorithmic mode. Our evaluation over a wide range of contention scenarios demonstrates that SmartPQ switches between the two algorithmic modes with negligible overheads, and significantly outperforms prior schemes, even when contention varies over time. We conclude that SmartPQ is an efficient concurrent priority queue for NUMA systems, and hope that this work encourages further study on adaptive concurrent data structures for NUMA architectures.

References

- L. A. Torrey, J. Coleman, and B. Miller, "A Comparison of Interactivity in the Linux 2.6 Scheduler and an MLFQ Scheduler," Software Practice and Experience., 2007.
- [2] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: a Scalable Relaxed Priority Queue," in PPoPP, 2015.
- [3] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU," ICPP, 2016.
- [4] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU," in ICPP, 2016.
- [5] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in OSDI, 2010.
- [6] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A Parallel Priority Queue with Constant Time Operations," J. Parallel Distrib. Comput., 1998.
- [7] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A Practical Concurrent Binary Search Tree," in PPoPP, 2010.
- [8] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir, "Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores," in *OPODIS*, 2013.
- [9] I. Calciu, H. Mendes, and M. Herlihy, "The Adaptive Priority Queue with Elimination and Combining," in DISC, 2014.
- [10] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box Concurrent Data Structures for NUMA Architectures," ASPLOS, 2017.
- [11] J. Choe, A. Huang, T. Moreshet, M. Herlihy, and R. I. Bahar, "Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation," in SPAA, 2019.
- [12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Third Edition. The MIT Press, 2009.
- [13] T. Crain, V. Gramoli, and M. Raynal, "No Hot Spot Non-blocking Skip List," in ICDCS, 2013.
- [14] H. Daly, A. Hassan, M. F. Spear, and R. Palmieri, "NUMASK: High Performance Scalable Skip List for NUMA," in DISC, 2018.

- [15] T. David, R. Guerraoui, and V. Trigonakis, "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask," in SOSP, 2013.
- [16] T. A. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," ASPLOS, 2015.
- [17] L. Dhulipala, C. Hong, and J. Shun, "ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms," in VLDB Endowment, 2020.
- [18] I. Dick, A. Fekete, and V. Gramoli, "A Skip List for Multicore," Concurrency and Computation: Practice and Experience, 2017.
- [19] L. Doddipalli and K. Rani, "Ensemble Decision Tree Classifier For Breast Cancer Data," *International Journal of Information Technology Convergence and Services*, 2012.
- [20] J. Eastep, D. Wingate, and A. Agarwal, "Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures," in ICAC, 2011.
- [21] A. Elafrou, G. I. Goumas, and N. Koziris, "Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors" in ICPP, 2017
- [22] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking Binary Search Trees," in PODC, 2010.
- [23] M. Fomitchev and E. Ruppert, "Lock-free Linked Lists and Skip Lists," in PODC, 2004
- [24] K. Fraser., "Practical Lock-Freedom," PhD thesis, University of Cambridge, 2004.
- [25] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in HPCA, 2021
- [26] P. Grönquist, C. Yao, T. Ben-Nun, N. Dryden, P. Dueben, S. Li, and T. Hoefler, "Deep Learning for Post-Processing Ensemble Weather Forecasts," *Philosophical Transactions of the Royal Society A*, 2021.
- [27] R. Guerraoui and V. Trigonakis, "Optimistic Concurrency with OPTIK," in PPoPP, 2016.
- [28] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based Scheduling to Improve Web Performance," TOCS, 2003.
- [29] T. Harris, "A Pragmatic Implementation of Non-Blocking Linked Lists," in *DISC*,
- [30] A. Heidarshenas, T. Gangwani, S. Yesil, A. Morrison, and J. Torrellas, "Snug: Architectural support for relaxed concurrent priority queueing in chip multiprocessors," in ICS, 2020.
- [31] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat Combining and the Synchronization-parallelism Tradeoff," in SPAA, 2010.
- [32] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Scalable Flat-Combining Based Synchronous Queues," in DISC, 2010.
- [33] M. Herlihy, "Wait-free Synchronization," TOPLAS, 1991.
- [34] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A Simple Optimistic Skiplist Algorithm," in SIROCCO, 2007.
- [35] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., 2008.
- [36] S. V. Howley and J. Jones, "A Non-blocking Internal Binary Search Tree," in SPAA, 2012.
- [37] S. V. Howley and J. Jones, "A Non-Blocking Internal Binary Search Tree," in SPAA, 2012.
- [38] D. Klaftenegger, K. Sagonas, and K. Winblad, "Delegation Locking Libraries for Improved Performance of Multithreaded Programs," in *EuroPar*, 2014.
- [39] V. Kolmogorov, "Blossom V: A new Implementation of a Minimum Cost Perfect Matching Algorithm," 2009.
- [40] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in SIGMOD, 2018.
- [41] A. Kusum, I. Neamtiu, and R. Gupta, "Safe and Flexible Adaptation via Alternate Data Structure Representations," in CC, 2016.
- [42] L. T. Schermerhorn. (2007) Automatic Page Migration for Linux [A Matter of
- [43] D. Lasalle and G. Karypis, "Multi-threaded Graph Partitioning," in IPDPS, 2013.
- [44] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," in SIGMOD, 2020.
- [45] J. Lindén and B. Jonsson, "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention," in OPODIS, 2013.
- [46] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent Data Structures for Near-Memory Computing," in SPAA, 2017.
- [47] I. Lotan and N. Shavit., "Skiplist-Based Concurrent Priority Queues," in *IPDPS*,

- [48] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications," in USENIX ATC, 2012.
- [49] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A Non-Blocking Priority Queue for the Pending Event Set," in SIMUTOOLS, 2016.
- [50] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic, "Using Meta-Heuristics and Machine Learning for Software Optimization of Parallel Computing Systems: A Systematic Literature Review," Computing, 2019.
- [51] K. Meng, J. Li, G. Tan, and N. Sun, "A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs," in PPoPP, 2019.
- [52] M. M. Michael, "High Performance Dynamic Lock-free Hash Tables and List-based Sets," in SPAA, 2002.
- [53] D. Michie, "'Memo' Functions and Machine Learning," in Nature, 1968.
- [54] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in PACT, 2009
- [55] A. Natarajan and N. Mittal, "Fast Concurrent Lock-free Binary Search Trees," in PPoPP, 2014.
- [56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," J. Mach. Learn. Res., 2011.
- [57] D. Petrović, T. Ropars, and A. Schiper, "On the Performance of Delegation over Cache-Coherent Shared Memory," in *ICDCN*, 2015.
- [58] J. C. Pichel and B. Pateiro-López, "A New Approach for Sparse Matrix Classification Based on Deep Learning Techniques," in CLUSTER, 2018.
- [59] K. Polat and S. Güneş, "A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-against-All Approach for Multi-Class Classification Problems," Expert Syst. Appl., 2009.
- [60] R. C. Prim, "Shortest Connection Networks and some Generalizations," The Bell Systems Technical Journal, 1957.
- [61] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," Commun. ACM, 1990.
- [62] M. Rab, R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "NUMA-Aware Non-Blocking Calendar Queue," in DS-RT, 2020.
- [63] M. Rawat and A. D. Kshemkalyani, "SWIFT: Scheduling in Web Servers for Fast Response Time," in NCA, 2003.
- [64] H. Rihani, P. Sanders, and R. Dementiev, "MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues," 2015.
- [65] S. Roghanchi, J. Eriksson, and N. Basu, "Ffwd: Delegation is (Much) Faster Than You Think," in SOSP, 2017.
- [66] K. Sagonas and K. Winblad, "The Contention Avoiding Concurrent Priority Queue," in LCPC, 2016.
- [67] K. Sagonas and K. Winblad, "The Contention Avoiding Concurrent Priority Queue," in LCPC, 2017.
- [68] P. Sanders, "Randomized Priority Queues for Fast Parallel Access," J. Parallel Distrib. Comput., 1998.
- [69] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic Selection of Sparse Matrix Representation on GPUs," in ICS, 2015.
- [70] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "RCU-HTM: A Generic Synchronization Technique for Highly Efficient Concurrent Search Trees," CCPE, 2021.
- [71] D. Siakavaras, K. Nikas, G. I. Goumas, and N. Koziris, "RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees," PACT 2017.
- [72] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra," in DSN, 2012.
- [73] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in ASPLOS, 2009.
- [74] H. Sundell and P. Tsigas, "Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems," *Journal of Parallel and Distributed Computing*, 2005.
- [75] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation," *TOMACS.*, 2005.
- [76] M. Thorup, "Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem," in STOC, 2003.
- [77] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The Lock-free k-LSM Relaxed Priority Queue," in PPoPP, 2015.
- [78] K. Winblad, K. Sagonas, and B. Jonsson, "Lock-Free Contention Adapting Search Trees," in SPAA, 2018.

- [79] Y. Xu, K. Li, and J. Hu, "A Genetic Algorithm for Task Scheduling on Heterogeneous Computing Systems using Multiple Priority Queues," *Information Sciences*, 2014.
- [80] D. Zhang and D. Dechev, "A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists," *TPDS*, 2016.
- [81] M. Zhang, H. Chen, L. Cheng, F. C. M. Lau, and C. Wang, "Scalable Adaptive NUMA-Aware Lock," *TPDS*, 2017.
- [82] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the Gap between Deep Learning and Sparse Matrix Format Selection," in *PPoPP*, 2018.