2016 45th International Conference on Parallel Processing

An Unbounded Nonblocking Double-ended Queue

Matthew Graichen Joseph Izraelevitz Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY, USA
matt.graichen@gmail.com, {jhi1, scott}@cs.rochester.edu

***Abstract*—We introduce a new algorithm for an unbounded concurrent double-ended queue (deque). Like the bounded deque of Herlihy, Luchangco, and Moir on which it is based, the new algorithm is simple and obstruction free, has no pathological long-latency scenarios, avoids interference between operations at opposite ends, and requires no special hardware support beyond the usual compare-and-swap. To the best of our knowledge, no prior concurrent deque combines these properties with unbounded capacity, or provides consistently better performance across a wide range of concurrent workloads.**

***Index Terms*—parallel processing; parallel algorithms; nonblocking algorithms;**

I. Introduction

Container classes—stacks, queues, sets, and maps—are among the most widely used abstractions in both sequential and parallel computing. Parallel containers are generally considered correct only if their operations (method calls) are *linearizable* [1], meaning that they appear, from the perspective of all threads, to take effect atomically at some point between their call and return.

Atomicity of container operations may be achieved using locks or via *nonblocking* programming techniques. An object (class instance) is said to be nonblocking if no thread, through inaction, can prevent other threads from making progress [2]. Nonblocking objects often scale better than those based on coarse-grain locks. They are also much less prone to performance anomalies caused by preemption.

Nonblocking objects can be classified in terms of liveness properties. A *wait-free* object guarantees that any thread t executing one of its methods will complete the operation in a bounded number of steps. A *lock-free* object guarantees that *some* thread will complete an operation in a bounded number of t's time steps. Finally, an *obstruction–free* [3] object guarantees that t will complete its operation in a bounded number of steps in the absence of contention. Wait-free objects preclude starvation. Lock-free objects preclude livelock. Obstruction-free objects require some sort of external mechanism to prevent livelock; the "randomization" inherent in most schedulers typically suffices.

Double ended queues (or *deques*) are lists that support `push` and `pop` operations at both ends. In this work, we introduce a new concurrent, nonblocking deque that avoids most of the limitations of existing implementations. Our work builds on the array-based, bounded *HLM deque* of Herlihy, Luchangco, and Moir [3]. Like that deque, ours is obstruction free, and allows operations on opposite ends to execute concurrently without interference. We allocate additional arrays as needed, however, to eliminate the bound on capacity. We also incorporate an optional elimination optimization [4], [5] to combine matching `push`es and `pop`s. Experimental results confirm that our deque maintains low latency while providing higher throughput than that of any known alternative.

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study. The authors would also like to thank Sean Brennan for his early exploration on this topic.

A. Related Work

Nonblocking stacks and queues have been studied extensively, beginning with the Treiber stack [6] and the Michael & Scott queue [7]. Subsequent breakthroughs include the introduction of *elimination*, which allows matching operations to complete without serializing [4], [8], [9]; *flat combining* [19], which improves cache locality but reintroduces blocking; and optimistic *fetch-and-increment* [10], which greatly reduces contention.

Work-stealing queues [11] are commonly used to dispatch tasks to worker threads. Access to one end of the structure is limited to `push`es and `pop`s by a distinguished *owner* thread. The other end supports concurrent access, but only for `pop`s. These restrictions are important, and can be leveraged to significantly enhance performance [12], [13].

The first fully functional nonblocking concurrent deque is due to Michael [14]. It is surprisingly complex, and suffers from contention between the two ends of the queue. The lock-free deque of Sundell and Tsigas [15] avoids this contention, but it can require lengthy helping operations if threads stall at inopportune times. The recent time-stamp deque of Dodds et al. [5], [16] has significantly better throughput, but at the expense of intentionally elevated latency, introduced to facilitate elimination. Finally, the obstruction-free array-based deque of Herlihy, Luchangco, and Moir [3] (the *HLM deque*) is refreshingly simple, but bounded. We employ this deque as the basis of a new, unbounded alternative.

II. Unbounded Double Ended Queue

A. Overview

At a global level, our deque consists of a doubly-linked list of HLM linear bounded deques, with careful handling of linking and unlinking operations. Consequently, we begin by

2332-5690/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPP.2016.32

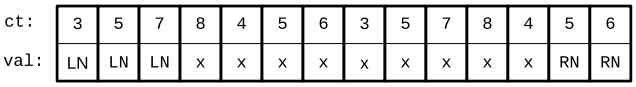

Fig. 1: HLM Bounded Linear Deque structure. Counter values are above, with data values below. The value x represents an arbitrary valid data object.

reviewing the behavior of the HLM deque. In our pseudocode, the keyword `tuple` indicates a single CAS-able value comprising several fields. We use the angle brackets ⟨ and ⟩ to indicate creating a tuple from individual field values.

1) HLM Bounded Linear Deque: Figure 1 shows the structure of the HLM bounded linear deque, and Figure 2 shows its type declarations [3]. The deque consists of a single array of special `hlm_slots`. Each slot contains a single 64-bit tuple comprising a 32-bit data value and a 32-bit counter. The linear array stores nontrivial data tuples contiguously: the reserved `LN` (left `NULL`) and `RN` (right `NULL`) tuples on either side of the contiguous data tuples indicate empty slots. A simple obstruction-free two-CAS protocol is used to `push` and `pop` values at the *edge* of the contiguous span of data values. To `push_left`, for example, a thread first finds the left edge—the boundary between the rightmost `LN` tuple and the data tuple to its right. It then performs a pair of CASes, first to increment the count of the leftmost data tuple and then to replace the rightmost `LN` tuple with a new data tuple (see Figure 3). The two-CAS protocol verifies both that the operation was enacted on the left edge and that any concurrent left-edge–changing operation would fail. A similar two-CAS protocol is used for `pop_left` (see Figure 3), and operations on the right side are symmetric. An arbitrary `oracle` function is used to find edges, but the expectation is that the structure will store hints to the edges that are periodically updated.

```
1 Object LN, RN; //reserved 32 bit consts
2 int HLM_SIZE;
3 tuple hlm_slot{
4   Object val; // 32 bits
5   int ct; // 32 bits
6 };
7 class hlm_linear_deque {
8   hlm_slot[HLM_SIZE] array;
9 };
```

Fig. 2: HLM Linear Deque Declarations

2) Unbounded Deque Structure: Figure 4 shows the structure of our deque and Figure 5 declares the types. Within each array-based `node` in our doubly linked list, we store a linear array of `slots` similar to the HLM deque. Once again, the linear array stores data tuples contiguously: the reserved `LN` and `RN` tuples on either side of the contiguous data tuples indicate empty slots, but the overall deque may span multiple buffers. Our slots on the interior of a buffer, like those of the HLM deque, contain either a `NULL` or a datum; we call these *data slots*. The two slots at the ends of the buffer (the *borders*), however, contain either a `NULL` or a pointer to an adjacent `node`; we call these *link slots*.

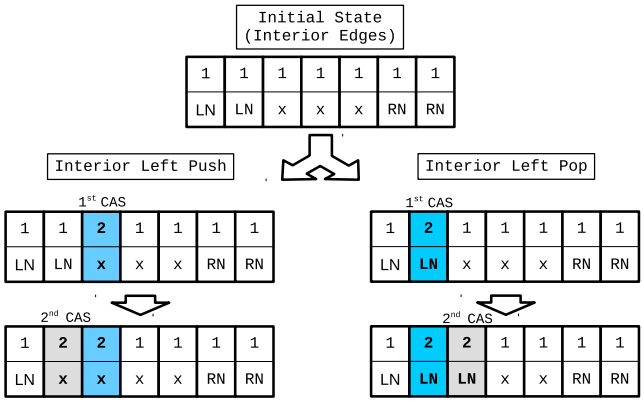

Fig. 3: Interior deque operations. Slot counters are all set to one initially—this is an unlikely scenario but simplifies exposition. We also omit hint values, which are generally updated after a successful transition.

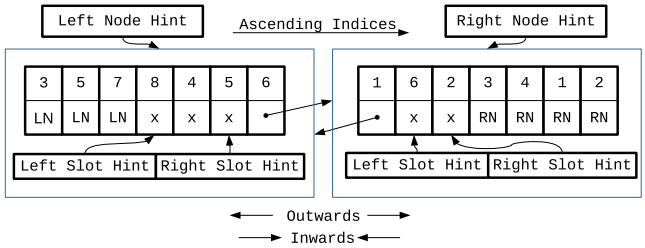

Fig. 4: Unbounded deque structure, consisting of two nodes, their slot arrays, and hints (both node local and deque global).

Given that the deque's contents are contiguous and accessed at both ends, we should like to discuss the algorithm in an end-agnostic way. Consequently, we use *inward* to mean the direction towards the center of the contiguous span of data values, and we use *outward* to mean the direction away from it. For example, the left edge of the contiguous span of data values lies between the *innermost* `LN` and the *outermost* data value (Figure 4). Similarly for the right edge.

We use explicit *hints* to find edges, and we update these upon completion of each operation. To accommodate temporarily incorrect hints, we use two auxiliary functions: `left_oracle()` and `right_oracle()`. As in the HLM algorithm, these functions return an index that was correct sometime during the call [3]; in our code, they also return a pointer to the appropriate node.

The reserved values `LS` and `RS` are "sealed" values, added to facilitate node reclamation. Nodes are sealed in their innermost data slot. A sealed node acts as a "cap" on its end of the deque and prevents outward growth in that direction; e.g., a left seal (`LS`) may be added in the rightmost data slot of a node (see Figure 9). Once a node has been sealed it can be removed from the deque and subsequently garbage collected.

3) Transitions: Modifications to the deque structure are made via a limited set of transition actions, each of which uses a short protocol to ensure that the transition's preconditions are

```
10 // reserved 32 bit constants
11 Object LN, RN, LS, RS;
12 int SZ; // length of node buffers
13
14 tuple slot {
15   union{ // 32 bits
16     Object value;
17     node* ptr;
18   };
19   int ct; // 32 bits
20 };
21
22 class node {
23   int left_slot_hint;
24   int right_slot_hint;
25   slot[] buffer;
26
27   node(int split) {
28     buffer = new slot[SZ];
29     for (int i = 0; i < split; i++)
30       buffer[i] = ⟨LN, 0⟩;
31     for (int i = split; i < SZ; i++)
32       buffer[i] = ⟨RN, 0⟩;
33     left_slot_hint = split-1;
34     right_slot_hint = split;
35   }
36   slot& operator[](int i){return buffer[i];}
37 };
38
```

```
39 tuple node_hint {
40   node *buffer; // 32 bits
41   int ct; // 32 bits
42 };
43
44 class deque {
45   node_hint left_node_hint, right_node_hint;
46   deque() {
47     left_node_hint.buffer = new node(SZ/2);
48     left_node_hint.ct = 0;
49     right_node_hint = left_node_hint;
50   }
51
52   // trace from hint to left/right edge
53   // (code not shown)
54   ⟨node*, int⟩ l_oracle(node_hint hint);
55   ⟨node*, int⟩ r_oracle(node_hint hint);
56
57   // update left/right hint from old to
58   // new node + index; return the new version
59   // (code not shown)
60   node_hint hint_l (node_hint old, node* nw_nd,
61    int nw_idx);
62   node_hint hint_r (node_hint old, node* nw_nd,
63    int nw_idx);
64
65   // frees node after no longer accessible
66   // by any other thread (code not shown)
67   retire(node* nd);
68 };
```

Fig. 5: Unbounded deque globals and data types

not violated by a concurrent transition. As in the `push` and `pop` operations of the HLM deque, our transitions rely on slot counters.

Unlike the HLM deque, our algorithm must accommodate node linking. Consequently, edges may arise in any of three places, as illustrated in Figure 7. The simplest case is said to be *interior*, where the edge occurs on the interior of a buffer. In the most complex, an edge *straddle*s a pair of adjacent nodes, that is, it aligns with the link between them. In the third, intermediate case, the edge may coincide with a node *boundary*, where there is no adjacent node, yet the edge is at the border of the buffer between the outer link slot and the outer data slot.

We can group the transitions into three categories: list modifications, empty-checks, and the hint-update. List modifications (e.g., inserting a value) use a two-CAS protocol similar to that of the HLM deque to modify the deque state. Empty checks, which are read-only, employ a multi-step snapshot. Hint update transitions are performance optimizations; they do not affect the underlying list structure. The transitions are:

L1 An interior push (line 92 & Figure 3).
L2 An interior pop (line 170 & Figure 3).
L3 A straddling push (line 125 & Figure 9).
L4 A boundary pop (line 225 & Figure 10).
L5 Sealing a node (line 198 & Figure 10).
L6 Appending a new node (line 106 & Figure 9).
L7 Removing a sealed node (lines 132 and 209 & Figures 9 and 10).
E1 An interior empty check (line 166 & Figure 11).
E2 A straddling empty check (lines 194 and 205 & Figure 11).
E3 A boundary empty check (line 221 & Figure 11).
H Updating the hint (line 60).

Our implementation uses these transitions to ensure that the deque is always in a valid state and that transitions are applied when the thread's perspective of the deque indicates that they are appropriate.

In general, threads attempting an operation first use the hint to find the edge node and slot for their operation (e.g., a `push_left`ing thread looks for the leftmost value among the contiguous data slots). Once the thread finds the edge data value (called `in`), it views the next slot outward (called `out`) and expects either a `NULL` value or a pointer. If it finds a pointer, it accesses the innermost data slot of its neighboring node (called `far`—effectively three slots outward from `in`), and verifies that its neighbor points back at it by checking the neighbor's link slot (`back`). Based on these few values (see Figure 8), the thread can determine if it has found an edge and what sort of edge it is (interior, straddling, or boundary). Based on the edge type, it applies the appropriate transitions(s) to complete its overall operation.

B. Implementation Details

Figure 5 shows how the deque structure is initialized. To start the doubly-linked list we allocate an initial node. The left half of this node's buffer is filled with `LN` values and the right half with `RN` values. Finally, we point both the `left_node_hint` and `right_node_hint` to this node and set the slot hints appropriately.

Since the `push` and `pop` operations are side-agnostic, from here on we will only discuss the details of `push_left` (Figure 6) and `pop_left` (Figure 12).

```
69 Object deque:push_left(Object o) {
70
71   while (true) {
72     // find edge
73     int edge_idx;
74     node edge_nd;
75     node_hint hint_cpy = left_node_hint;
76     ⟨edge_nd, edge_idx⟩ = l_oracle(hint_cpy);
77
78     slot* in = &edge_nd[edge_idx];
79     slot in_cpy = *in;
80     slot* out = &edge_nd→[edge_idx-1];
81     slot out_cpy = *out;
82
83     // check oracle's edge
84     if ((in_cpy.val == LN || in_cpy.val == RS)
85      ||(edge_idx ≠ 1 && out_cpy.val ≠ LN)
86      ||(edge_idx == SZ - 1 && in_cpy.val ≠ RN))
87       continue;
88
89     // interior push
90     if (edge_idx ≠ 1) {
91       if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
92        && CAS(out, out_cpy, ⟨o, out_cpy.ct+1⟩)) {
93         hint_l(hint_cpy, edge_nd, edge_idx-1);
94         return OK; }
95     } // end interior push
96
97     // edge is either straddling or boundary
98     else {
99       // check state for boundary edge (append)
100      if (out_cpy.val == LN) {
101        // create new node
102        node* nw_nd = new node(SZ);
103        nw_nd→[SZ-2] = o;
104        nw_nd→[SZ-1] = edge_nd;
105        if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
106         && CAS(out, out_cpy, ⟨nw_nd, out_cpy.ct+1⟩)) {
107          hint_l(hint_cpy, nw_nd, SZ-2);
108          return OK; }
109      }
110      // edge is straddling, so either straddle push or
111      // help remove sealed node on left
112      else {
113        node* out_nd = out_cpy.val;
114        slot* far = &(out_nd→[SZ-2]);
115        slot far_cpy = *far;
116
117        // ensure left neighbor points back
118        slot* back = &(out_cpy.val→[SZ-1]);
119        slot back_cpy = *back;
120        if (back.val ≠ edge_nd) {continue;}
121
122        // check state for straddling push
123        if (far_cpy.val == LN) {
124          if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
125           && CAS(far, far_cpy, ⟨o, far_cpy.ct+1⟩)) {
126            hint_l(hint_cpy, out_nd, SZ-2);
127            return OK; }
128        }
129        // remove sealed node on left
130        else if (far_cpy.val == LS) {
131          if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
132           && CAS(out, out_cpy, ⟨LN, out_cpy.ct+1⟩)) {
133            // for memory reclamation update both hints
134            hint_l(hint_cpy, edge_nd, 1);
135            hint_r(oracle_r(right_node_hint));
136            retire(out_nd); }
137        }
138      } // end straddling edge
139    } // end boundary or straddling edge
140  } // end while
141 } // end method
```

Fig. 6: Unbounded deque `push_left()` (symmetric code for `push_right()`)

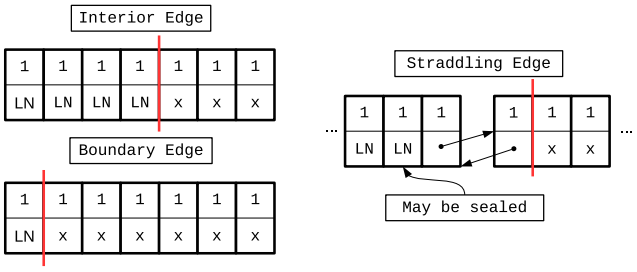

Fig. 7: Edge types

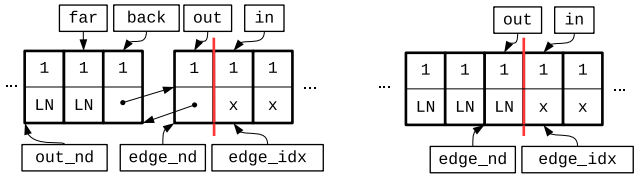

Fig. 8: Thread local values from snapshot

1) `push_left`: Our `push_left` operation (Figure 6) illustrates the general pattern for operations: find the edge, read the `in`, `out`, and `far` pointers, then apply the appropriate transitions.

If `push_left` finds an interior edge based on the `edge_idx`, it simply attempts to apply an interior push which follows exactly the two-CAS protocol of the HLM deque (Figure 3). If the interior push fails, then the edge may have moved, so the entire operation retries (including re-finding the edge).

If `push_left` instead discovers that the outermost data value is in an outermost data slot, it is possible that the deque's left edge is in the straddling or boundary case. In the boundary case, the thread uses the append transition (Figure 9) to add a new node to the left of the current deque. The append transition is effectively an HLM push but the pushed "value" is a pointer in the link slot. By using the two-CAS protocol for append, the thread ensures that the boundary edge has not changed.

Alternatively, if the deque's left edge is in the straddling case, then `edge_nd` has a left neighbor. We copy the inner-most data slot of our left neighbor (`far`) and verify that the right link slot of our neighbor (`back`) indeed points back to our starting node. If the edge is a valid straddling edge, `far` slot is either `LS` or `LN`. If it is `LN` we can push our value into it using the straddling push transition (Figure 9). However, if it is `LS` then we must remove our left neighbor to progress using the remove transition (Figure 9); then we begin the operation again. Both operations may be interrupted—and forced to start over—by a concurrent straddling operation.

2) `pop_left`: Our `pop_left` method (Figure 12), like `push_left`, follows the established pattern: find the edge, read the `in`, `out`, and `far` pointers, then apply the appropriate

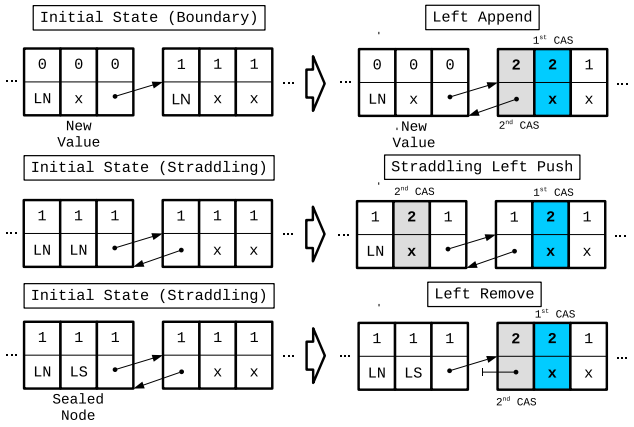

Fig. 9: Non-interior `push` operations

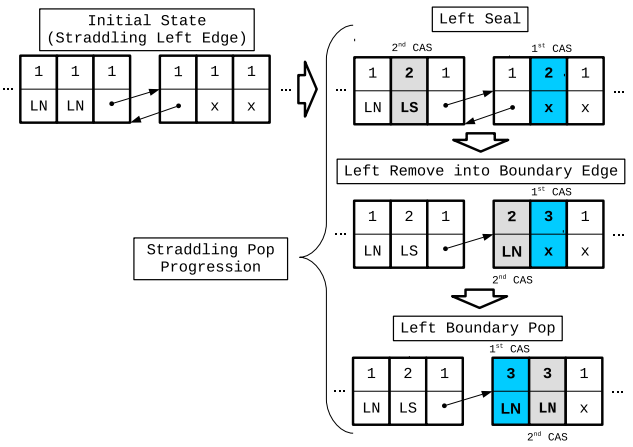

Fig. 10: Non-interior `pop` operations

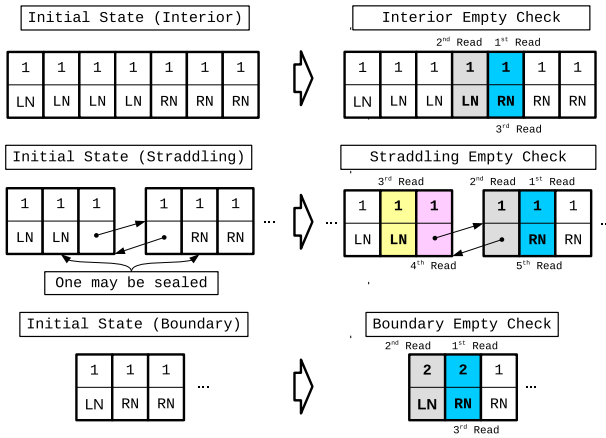

Fig. 11: Empty check operations

transitions.

If `pop_left` finds an interior edge based on the `edge_idx`, it applies the interior pop of the HLM deque (Figure 3).

If `pop_left` finds that the outermost data value is in an outermost data slot, it is again possible that the deque's left edge is in the straddling or boundary case.

If the deque's left edge is in the straddling case, `edge_nd` has a left neighbor pointed to by the left link slot (`out`). A straddling edge for a `pop` operation triggers a series of transitions we call the "straddling pop progression" (see Figure 10). Assuming that we truly have a straddling edge, we read the innermost data slot of our left neighbor (`far`), and verify that the left neighbor points back (`back`). If the `far` slot is not sealed, we seal it using the seal transition. Now that our left neighbor is sealed (by us or another thread), we remove the left neighbor node with the remove transition. Finally, since we have no left neighbor, we have a boundary edge.

In the boundary edge case, whether because the current thread found a boundary edge or because it followed the straddling pop progression, the thread uses the boundary pop transition to remove from the outermost data slot (Figure 10).

For all transitions in `pop_left`, if we notice that the leftmost data value (`in`) is either `RN` or `RS`, we use a snapshot empty check to verify the deque is empty and that our edge remains valid (Figure 11). This empty check also prevents us from having two sealed nodes pointing at each other, since the second node to get sealed will instead return `EMPTY`.

C. Memory Management

Our pseudocode for the most part ignores memory management, except for the `retire` method. After a thread has detached a node from the deque, it keeps it in a thread local *retirement list*. After no threads are able to access the node, we know that the node can be freed.

Our memory reclamation scheme leverages the invariant that removed nodes cannot be accessed from active (unremoved) nodes; the remove operation breaks the link in this direction (Figure 10). Consequently, when a node is retired, we update the global hints to any active node using a CAS. Any future thread that reads these new hints cannot trace to our retired node, and, if all `node_hint` updates use a CAS, our node will be inaccessible from all future hinted nodes. We use hazard pointers [17] to track threads with earlier hints as they traverse the chain in the `oracle` function.

D. Optimizations

We enacted one key optimization not discussed in the algorithm description. As noted by Dodds et al. [5], deques, like stacks, can use elimination arrays [4] on each end—a like-sided push and pop never need to actually access the deque structure and instead can "cancel out" if they overlap in time.

We use modified elimination arrays (Figure 13) to optimize our algorithm, one on each side of the deque (`l_elim` and `r_elim`). The `elimination` class exports three methods: an `insert` method, which stores details of the operation in a thread-local elimination slot, a `scan` method, which scans the entire elimination array searching for opposite operations to eliminate with, and a `remove` method, which removes the

```
142 Object deque:pop_left() {
143
144   while (true) {
145
146   // find edge
147   int edge_idx;
148   node edge_nd;
149   node_hint hint_cpy = left_node_hint;
150   ⟨edge_nd, edge_idx⟩ = l_oracle(hint_cpy);
151
152   slot* in = &edge_nd→[edge_idx];
153   slot in_cpy = *in;
154   slot* out = &edge_nd→[edge_idx-1];
155   slot out_cpy = *out;
156
157   // check oracle's edge
158   if ((in_cpy.val == LN || in_cpy.val == RS)
159    || (edge_idx ≠ 1 && out_cpy.val ≠ LN)
160    || (edge_idx == SZ - 1 && in_cpy.val ≠ RN))
161     continue;
162
163   // interior edge,
164   // so interior pop or empty check
165   if (edge_idx ≠ 1) {
166     if (in_cpy.val == RN && *in == in_cpy) {
167       return EMPTY;
168     }
169     if (CAS(out, out_cpy, ⟨LN, out_cpy.ct+1⟩)
170      && CAS(in, in_cpy, ⟨LN, in_cpy.ct+1⟩)) {
171       hint_l(hint_cpy, edge_nd, edge_idx+1);
172       return in_cpy.val;
173     }
174   } // end interior pop
175
176   // edge is on border of array, so follow straddling
177   // pop progression as necessary: seal left node,
178   // remove left node, then boundary pop
179     else {
180       // check if we have a straddling edge
181       if (out_cpy.val ≠ LN) {
182         node* out_nd = out_cpy.val;
183         slot* far = &(out_nd→[SZ-2]);
184         slot far_cpy = *far;
185
186         // ensure left neighbor points back
187         slot* back = &(out_cpy.val→[SZ-1]);
188         slot back_cpy = *back;
189         if (back.val ≠ edge_nd) {continue;}
190
191         // check for straddled edge and seal
192         if (far_cpy.val == LN) {
193           if ((in_cpy.val == RN || in_cpy.val == RS)
194            && *in == in_cpy) {
195             return EMPTY;
196           }
197           if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
198            && CAS(far, far_cpy, ⟨LS, far_cpy.ct+1⟩)) {
199             far_cpy = ⟨LS, far_cpy.ct+1⟩;
200             in_cpy = ⟨in_cpy.val, in_cpy.ct+1⟩;
201           }
202         }
203         // check for sealed left node and remove it
204         if (far_cpy.val == LS) {
205           if (in_cpy.val == RN && *in == in_cpy) {
206             return EMPTY;
207           }
208           if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)
209            && CAS(out, out_cpy, ⟨LN, out_cpy.ct+1⟩)) {
210             // for memory reclamation update both hints
211             hint_cpy = hint_l(hint_cpy, edge_nd, 1);
212             hint_r(oracle_r(right_node_hint));
213             retire(out_nd);
214             in_cpy = ⟨in_cpy.val, in_cpy.ct+1⟩;
215             out_cpy = ⟨LN, out_cpy.ct+1⟩;
216           }
217         }
218       }
219       // check for boundary edge, then boundary pop
220       if (out_cpy.val==LN) {
221         if (in_cpy.val==RN && *in==in_cpy) {
222           return EMPTY;
223         }
224         if (CAS(out, out_cpy, ⟨LN, out_cpy.ct+1⟩)
225          && CAS(in, in_cpy, ⟨LN, in_cpy.ct+1⟩)) {
226           hint_l(hint_cpy, edge_nd, 2);
227           return in_cpy.val;
228         }
229       }
230     } // end boundary or straddling edge
231   } // end while
232 } // end method
```

Fig. 12: Unbounded deque `pop_left()` (symmetric code for `pop_right()`)

operation from the slot, possibly after it has combined with an opposite operation.

Figure 13 shows the use of our elimination arrays for the `pop_left` operation (other methods are similar). When they begin an operation, threads `insert` the details of their operation into a thread local slot in the appropriate elimination array. This information remains in the array while the thread searches for an edge in the `oracle` function. Once a thread finds an edge, it `removes` itself from its slot. Conceivably, the thread has already eliminated with an overlapping opposite operation, in which case the thread has completed. If the thread does not eliminate, it begins to attempt transitions on the actual deque. Should the thread fail to complete its operation on the actual deque, it `scans` the elimination array for opposite operations to combine with. Should this scan fail, the thread reinserts itself into the elimination array and retries the entire operation from the beginning.

Our elimination design moves the high overhead elimination scan off the critical path, reduces contention on the main deque, and leverages the latency of the `oracle` function as a strength. Furthermore, it relieves us of the need to pick an appropriate period for threads to wait in the elimination array before trying the actual deque.

III. Correctness

We here present an informal proof of our algorithm's correctness, discussing both safety and liveness guarantees. For clarity of exposition, we elide discussion of garbage collection in the proof.

A. Safety

To prove our algorithm correct, we must explain the desired sequential semantics, identify linearization points for our operations, and, finally, demonstrate that any realizable parallel execution has the same observable behavior as a sequential execution performed in linearization order.

1) Sequential Semantics: A sequential deque exports `push_left`, `pop_left`, `push_right`, and `pop_right` methods with the usual semantics. At any given point in

```
233 class elimination {
234   // occupy thread local elimination slot
235   // using release store
236   void insert(int operation, Object val, int thread_id);
237   // remove self from thread local elimination slot
238   // using CAS; return partner's value if eliminated
239   Object remove(int thread_id);
240   // scan for elimination partner
241   // notify partner using CAS on their slot
242   // return partner's value if successfully combined
243   Object scan(int operation, Object val, int thread_id);
244 };
245
246 class deque {
247   ...
248   elimination l_elim;
249   elimination r_elim;
250 };
251
252 Object deque:pop_left() {
253   l_elim.insert(POP, NULL, thread_id);
```

```
254   while (true) {
255     ...
256     ⟨edge_nd, edge_idx⟩ = l_oracle(left_node_hint);
257     Object retval;
258     if ((retval = l_elim.remove(thread_id)) ≠ NULL)
259       return retval;
260     // check oracle's edge
261     if ((in_cpy.val == LN || in_cpy.val == RS)
262      ||(edge_idx ≠ 1 && out_cpy.val ≠ LN)
263      ||(edge_idx == SZ - 1 && in_cpy.val ≠ RN))
264       goto try_elim;
265     // transitions
266     if (edge_idx≠ 1) {...}
267     else {...}
268
269     try_elim: // there appears to be contention
270     if ((retval = l_elim.scan(POP, NULL, thread_id))
271      ≠ NULL)
272       return retval;
273     l_elim.insert(POP, NULL, thread_id);
274   } // end while
275 } // end method
```

Fig. 13: Elimination optimization of `popleft()`, modifications bolded

time, the deque has an abstract state comprising a finite sequence of elements; initially this sequence is empty. A `push_right(`x`)` operation moves the abstract state from S to $S.x$, by concatenating x to the end of S. A `push_left(`x`)` operation moves the abstract state from S to $x.S$, by concatenating x to the beginning of S. When S is empty, `pop_right` and `pop_left` return `EMPTY` and leave S unchanged. If the abstract state is $S.x$, `pop_right` moves to S and returns x. Similarly, `pop_left` moves from $x.S$ to S and returns x. A sequential history is valid if we can identify abstract states $S_i, \forall i$, such that S_0 is empty and the ith operation in the history moves from S_{i-1} to S_i.

2) Linearization points: To prove the safety of our deque, it is sufficient to show that in any realizable concurrent history it is possible to identify linearization points (each between the call and return of its operation) such that the history has the same observable behavior (i.e., return values) as a sequential execution that performs the same operations in linearization order on a sequential deque.

1) A `push` operation linearizes when the second CAS of its insertion transition succeeds (line 92 for an interior push, line 125 for a straddling push, and line 106 for a node append).
2) A `pop` operation that finds a nonempty container linearizes when the second CAS of its removal protocol succeeds (line 170 for an interior pop and line 225 for a boundary pop).
3) A `pop` operation that finds an empty container linearizes when the final read of its empty check succeeds (line 166 for an interior empty check, line 194 or line 205 for a straddling empty check, and line 221 for a boundary empty check).

Theorem 1 (Linearizability)**.** *Any realizable well-formed history of our concurrent deque containing only completed operations is equivalent to a valid well-formed history of a sequential deque.*

Proof. By induction. In the base case, our deque begins in a quiescent state with a single empty node. The left half of the node's array contains `LN` values, while the right side of the array contains `RN` values. Both exterior hints point to the single node, and both of the node's interior hints point to the center.

Our induction invariant requires that the deque is well-formed as described below. All non-removed (active) nodes are doubly-linked to their neighbors and form a contiguous chain. For the active nodes, any empty slot on the left contains the `LN` value, and any empty slot on the right contains the `RN` value. The innermost non-value slots may be an `LS` or `RS` if the associated edge is a straddling edge. Interior slots of active nodes between the edges must contain stored values (non-`NULL` non-`SEALED` values). We call the contiguous chain of active nodes the *active chain*.

Sealed nodes are sealed on either the right or left side (*right sealed* or *left sealed*) and lie on either side of the active chain. Sealed nodes cannot interrupt the active chain, but may be on one or both ends. A sealed node, on its inward side, may be singly linked inward toward the active chain (its inward side neighbor might not point back to it). A sealed node, on its inward side, is the neighbor of either an active node, another sealed node which has been sealed on the same side, or another sealed node which has been sealed on the opposite side but does not point back.

Examination of the code confirms that the deque's state can be changed by a small set of transitions (Section II-A3). These transitions (or concurrent sets of them) are our induction steps; they transition the deque from one well-formed state to another.

All transitions act on one side of the deque and have analogous operations on the other side. While the empty check transitions do not actually change the underlying structure,

they can still be considered a valid "action" on the deque.

The non-structural edge transitions (L1–L5) use the double CAS protocol to change the edge of the deque without changing the underlying doubly linked list. For a given side, these operations will all conflict with each other: only one of these transitions can succeed at once. If the deque contains zero or one elements, these transitions also conflict with the opposite side. The empty check transitions (E1–E3) use a snapshot read protocol to ensure that the edge does not change: they conflict with the non-structural edge transitions (L1–L5) but not with other empty check transitions (E1–E3).

The linking transitions (L6–L7) use a slightly different double CAS protocol, yet this modified double CAS protocol also conflicts with non-structural transitions (L1–L5) and empty check transitions (E1–E3). The append transition (L6) will fail if any other thread changes the edge: moving the edge inward will cause the first CAS to fail, while moving the edge outward will cause the second CAS to fail. The boundary empty check's triple read protocol will be aborted if it overlaps with an append.

The remove transition (L7) also conflicts with the non-structural transitions (L1–L5) and all empty checks (E1–E3). Based on the initial condition for the remove transition, a non-structural transition (L1–L5) which attempts to move the edge outward or inward will fail due to the sealed slot as will any empty check (E1–E3). Append and remove transitions conflict with each other as they have different initial conditions and they both trivially conflict with themselves. Finally, append and remove transitions will never conflict with their opposite side analogues if the node buffer size is sufficiently large.

The hint update (H) can occur simultaneously with any other transition. Examination of the code confirms that the hint always points to some valid node, either active or inactive. From either node type, a traversal of a well formed deque can always find the edges of an active chain. Thus a hint update can safely occur simultaneously with any other transitions.

By induction, we know that the deque is always well-formed. The linearized history is the induced order of transitions with non-exported transitions (sealing, detaching, hint updating) dropped. The linearized history is correct since the contents of the active chain are always equivalent to a sequential deque with an equivalent history. □

B. Liveness and Contention Freedom

Theorem 2 (Obstruction Freedom)**.** *The presented unbounded concurrent deque is obstruction-free.*

Proof. Only three loops exist in the code, one each in the `push` and `pop` methods, and one in the oracle method. In the absence of contention, the snapshots taken by threads from the local variables `in`, `out`, and `far` remain valid. Examination of the code confirms that, for any well-formed unbounded deque state, some transition will be chosen. For a `push` operation, assuming no contention, only a `SEALED` neighbor will cause a retry. After removing the neighbor, a subsequent append will complete the push. For a `pop` operation, assuming no contention, a series of one or more transitions will be performed, ending in a successful pop.

In the absence of contention, assuming a well-formed deque, we can always follow sealed nodes back to the active chain since they always point to nodes that were sealed after them. Eventually the oracle function will find the edge node and its slot by following this chain. □

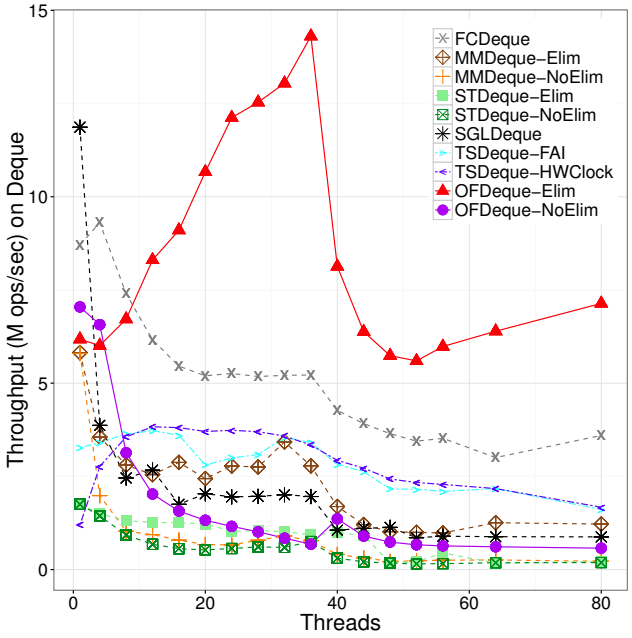

Fig. 14: Throughput for deque access pattern

IV. Performance Results

For data structure evaluation, we used an Intel machine with two eighteen-core, two-way hyper-threaded Intel Xeon E5-2699 v3 processors at 3.6 GHz (i.e., with up to 72 hardware threads). Every core's L1 and L2 caches are private to that core (shared between hyper-threads); the L3 cache (45 MB) is shared across all cores of a single processor. The machine runs Fedora Core 19 Linux. Tests were performed in a controlled environment when we were the sole users of the machine. Threads were pinned to cores in a consistent order for all experiments: one thread per physical core on the first processor (1–18), then one thread for each additional hyper-thread on that processor (19–36), then one thread per core (37–54) and one per additional hyper-thread (55–72) on the second processor. Code was written in C++ and compiled at the `-O3` optimization level using g++ 4.8.2. When a nonblocking memory allocator would improve performance, we used one adapted from the Rochester Software Transactional Memory (RSTM) package [18].

Our test comprises a micro-benchmark, run for a fixed period of time, in which every thread repeatedly executes some method of the deque, uniformly and randomly choosing the method at each iteration. We ran experiments in which threads use the deque in a **Stack**, **Queue**, and **Deque** access pattern

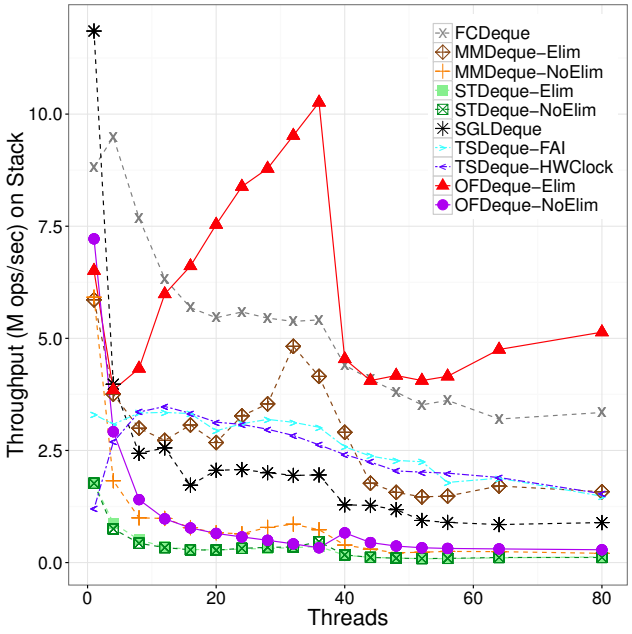

Fig. 15: Throughput for stack access pattern

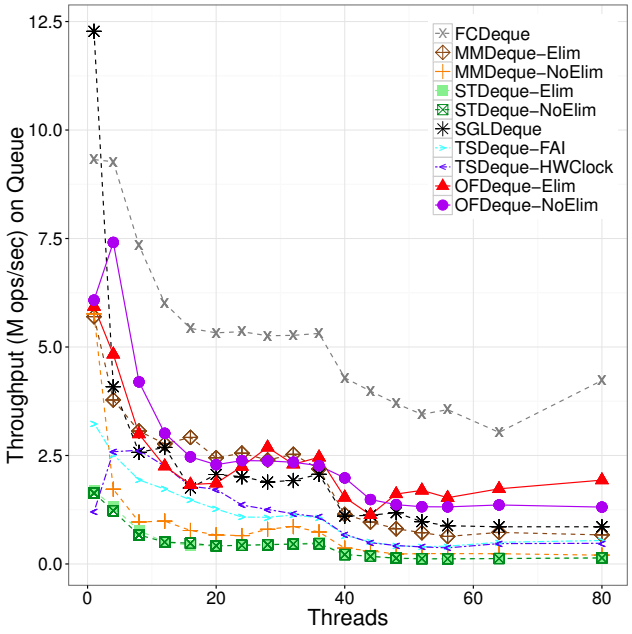

Fig. 16: Throughput for queue access pattern

(e.g., under **Stack**, threads chose only between `push_left` and `pop_left`). Each configuration was tested five times; we report the average.

We tested several different deques—**SGLDeque**: a deque protected by a single global test-and-test_and_set lock; **FCDeque**: a concurrent deque using flat combining with an exponential backoff lock [19]; **MMDeque**, **STDeque**: the lock free deque of Maged Michael [14] and the lock free deque of Sundell and Tsigas [15] respectively, both deques with and without exponential backoff elimination arrays [4]; **TSDeque-FAI**, **TSDeque-HW**: the time stamping deque of Dodds et al. [5], using a fetch-and-increment counter and the hardware cycle counter respectively — both versions use an elimination style optimization; and **OFDeque**: the obstruction free deque which is the topic of this paper, shown with and without the elimination optimization. We chose 1024 as a representative number of slots in each buffer; no significant performance impact was noted for different buffer sizes.

As can be seen in Figures 14, 15, and 16, our new concurrent **OFDeque** generally outperforms prior art across the range of thread counts and for most access patterns. Its single-thread throughput also exceeds that of all the nonblocking alternatives.

The elimination technique provides a significant boost for the **Deque** and **Stack** access patterns, allowing performance of the **OFDeque** within one socket to scale with the number of threads. Elimination is a general technique, and can be applied to both the **STDeque** and **MMDeque** (the **TSDeque** already incorporates an elimination mechanism), however, due to their slow single threaded latency, the elimination does not help these deques as much.

For the **Queue** access pattern, where elimination is not generally feasible, flat combining achieves the best performance by maximizing cache locality and reducing contention, but the presented obstruction free deque generally outperforms all nonblocking alternatives. Note that on the **Queue** access pattern, elimination will generally hurt performance, since operations will never combine in this test, as seen in the **OFDeque**'s performance, unless it acts as a contention manager, as seen in the **MMDeque**.

V. Conclusion

In conclusion, our algorithm provides a novel unbounded and obstruction free double ended queue construction. Our structure outperforms across the range of thread counts other state of the art nonblocking solutions and, for certain access patterns, also outperforms blocking solutions.

References

[1] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[2] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, Nov. 1993.

[3] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, ser. ICDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 522–.

[4] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proc. of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, ser. SPAA '04. Barcelona, Spain: ACM, 2004, pp. 206–215.

[5] M. Dodds, A. Haas, and C. M. Kirsch, "Fast concurrent data-structures through explicit timestamping," Department of Computer Sciences, Universitt Salzburg, Tech. Rep. Technical Report 2014-03, Feb. 2014.

[6] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, Apr. 1986.

[7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. of the 1996 ACM Symp. on Principles of Distributed Computing*, ser. PODC '96. Philadelphia, PA, USA: ACM, 1996, pp. 267–275.

[8] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proc. of the 17th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, ser. SPAA '05. Las Vegas, NV, USA: ACM, 2005, pp. 253–262.

[9] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Intl. Conf. on Principles of Distributed Systems*, ser. OPODIS '07, 2007, pp. 401–414.

[10] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 103–112.

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proc. of the Thirty-fifth Intl. Symp. on Computer Foundations of Computer Science (FOCS)*, Santa Fe, NM, Nov. 1994, pp. 356–368.

[12] D. Hendler, Y. Lev, M. Moir, and N. Shavit, "A dynamic-sized nonblocking work stealing deque," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2005.

[13] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems*, vol. 34, no. 2, pp. 115–144, 2001.

[14] M. M. Michael, "CAS-based lock-free algorithm for shared deques," in *Proc. of the 9th Intl. European Conf. on Parallel and Distributed Computing*, ser. Euro-Par 2003. Klagenfurt, Austria: Springer Berlin Heidelberg, 2003, pp. 651–660.

[15] H. Sundell and P. Tsigas, "Lock-free and practical doubly linked list-based deques using single-word compare-and-swap," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, T. Higashino, Ed. Springer Berlin Heidelberg, 2005, vol. 3544, pp. 240–255.

[16] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," in *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, ser. POPL '15. Mumbai, India: ACM, 2015, pp. 233–246.

[17] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 8, pp. 491–504, Aug. 2004.

[18] "Reconfigurable Software Transactional Memory; Release 7," https://github.com/mfs409/rstm (Accessed 2015).

[19] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, ser. SPAA '10, Santorini, Greece, Jun. 2010, pp. 355–364.