Theory Comput. Systems **35**, 391–432 (2002)
DOI: 10.1007/s00224-002-1046-6

Theory of Computing Systems

Black-Box Correctness Tests for Basic Parallel Data Structures*

Phillip B. Gibbons,[1] John L. Bruno,[2] and Steven Phillips[3]

[1]Intel Research Pittsburgh, 417 South Craig Street,
Pittsburgh, PA 15213, USA
phillip.b.gibbons@intel.com

[2]Office of the Vice Provost - Information and Educational Technology,
University of California, Davis, CA 95616, USA
jlbruno@ucdavis.edu

[3]AT&T Labs–Research, AT&T Shannon Laboratory,
180 Park Avenue, Florham Park, NJ 07932, USA
phillips@research.att.com

Abstract. Operations on basic data structures such as queues, priority queues, stacks, and counters can dominate the execution time of a parallel program due to both their frequency and their coordination and contention overheads. There are considerable performance payoffs in developing highly optimized, asynchronous, distributed, cache-conscious, parallel implementations of such data structures. Such implementations may employ a variety of tricks to reduce latencies and avoid serial bottlenecks, as long as the semantics of the data structure are preserved. The complexity of the implementation and the difficulty in reasoning about asynchronous systems increases concerns regarding possible bugs in the implementation.

In this paper we consider postmortem, *black-box* procedures for testing whether a parallel data structure behaved correctly. We present the first systematic study of algorithms and hardness results for such testing procedures, focusing on queues, priority queues, stacks, and counters, under various important scenarios. Our results demonstrate the importance of selecting test data such that distinct values are inserted into the data structure (as appropriate). In such cases we present an $O(n)$ time algorithm for testing linearizable queues, an $O(n \log n)$ time algorithm for testing linearizable priority queues, and an $O(np^2)$ time algorithm for testing sequentially consistent queues, where n is the number of data structure operations and p is the

* This work was performed while the first author was with Bell Laboratories. This work was performed while the second author was visiting Bell Laboratories.

number of processors. In contrast, we show that testing such data structures for executions with arbitrary input values is NP-complete. Our results also help clarify the thresholds between scenarios that admit polynomial time solutions and those that are NP-complete. Our algorithms are the first nontrivial algorithms for these problems.

1. Introduction

Operations on basic data structures such as shared queues, priority queues, stacks, and counters can often dominate the execution time of a parallel program. This dominance arises due to the large number of operations on the data structure by the processors, including multiple operations contending for the shared data structure at the same time. An example would be a shared queue used to hold tasks available for execution by the processors, where multiple processors grab tasks from the head of the queue and deposit new tasks to the tail of the queue. There are considerable performance gains arising from developing highly optimized, asynchronous, distributed, cache-conscious, parallel implementations of such data structures. Such implementations may employ a variety of tricks to reduce latencies and avoid serial bottlenecks, including servicing multiple requests simultaneously or even out-of-order. Examples include implementations based on counting networks [AHS], elimination trees [ST], diffracting trees [SUZ], or combining funnels with elimination [SZ]. In fact, the only requirement of the implementation is that it preserves the (serial) semantics of the data structure, as observed by the processors interacting with the data structure. The complexity of the implementation and the difficulty in reasoning about asynchronous parallel systems increases concerns regarding possible bugs in the implementation.

In this paper we consider *black-box* procedures for testing whether a parallel data structure behaved correctly in an execution of a parallel program. Black-box testing procedures rely solely on the outcomes of the data structure operations, as recorded at the individual processors requesting the operations. Since they are external to the implementation, they work for any implementation, and do not interfere with the implementation (such interference can mask possible timing-dependent errors).

We record a trace for each processor of its operations on the shared data structure over the course of a program execution, and then perform a post-execution/postmortem audit of the per-processor traces. This audit either detects one or more errors in the implementation or validates the correctness of the execution by determining that there is a merging of the per-processor traces (i.e., a topological sort of the operations in these traces) that preserves the (serial) semantics of the data structure/object.

Consider the example depicted in Figure 1 of per-processor traces for two executions of a program with three processors, p_1, p_2, and p_3, accessing a shared stack. Push(v) denotes that the processor pushed the value v on the stack. Pop(v) denotes that the processor popped the top of the stack and the value v was returned.[1] Arrows indicate the

[1] For notational uniformity, we use Pop(v); an alternative notation would be v = Pop().

p_1	p_2	p_3	p_1	p_2	p_3
Push(2)	Pop(1)	Pop(3)	Push(2)	Pop(1)	Pop(5)
↓	↓	↓	↓	↓	↓
Push(3)	Pop(2)	Pop(5)	Push(3)	Pop(2)	Pop(3)
↓	↓		↓	↓	
Push(1)	Push(5)		Push(1)	Push(5)	

Fig. 1. Valid versus invalid stack traces.

operation order at the individual processors, so for example in both executions, processor p_1 pushes 2 onto the stack, then pushes 3, then pushes 1.

The per-processor traces on the left correspond to a valid execution of the shared stack, because there exists a merging of the traces that preserves the semantics of a stack. Specifically, there are two such mergings:

Push(2), Push(3), Push(1), Pop(1), Pop(3), Pop(2), Push(5), Pop(5)

and

Push(2), Push(3), Pop(3), Push(1), Pop(1), Pop(2), Push(5), Pop(5).

The reader may verify that in both total orders, the value returned for each pop operation is the value currently at the top of the stack.

The per-processor traces on the right, however, are invalid, because there is no possible merging of the traces such that each pop operation returns the value at the top of the stack. To see this, observe that any merged order in which each push appears before its corresponding pop begins:

Push(2), Push(3), Push(1), Pop(1), Pop(2),

and this last pop operation does not return the value 3 at the top of the stack. The traces on the right might arise in a buggy shared stack implementation in which processor p_3 issues its two pop requests, and the second request bypasses the first (e.g., in the interconnection network).

As a second example, consider the per-processor traces depicted in Figure 2. In this example there are time intervals associated with each operation. Push(v, t_1, t_2) denotes that the processor pushed the value v on the stack during the time interval $[t_1, t_2]$. Pop(v, t_1, t_2) denotes that the processor popped the top of the stack during the time interval $[t_1, t_2]$ and the value v was returned. The time intervals for the trace on the left are depicted in Figure 3, together with the partial order defined by these intervals (called

p_1	p_2	p_3	p_1	p_2	p_3
Push(2, 1, 3)	Pop(2, 4, 9)	Pop(3, 2, 13)	Push(2, 1, 3)	Pop(2, 7, 9)	Pop(3, 11, 13)
Push(3, 5, 6)	Push(5, 10, 16)	Pop(5, 15, 18)	Push(3, 5, 6)	Push(5, 10, 16)	Pop(5, 15, 18)

Fig. 2. Valid versus invalid linearizable stack traces.

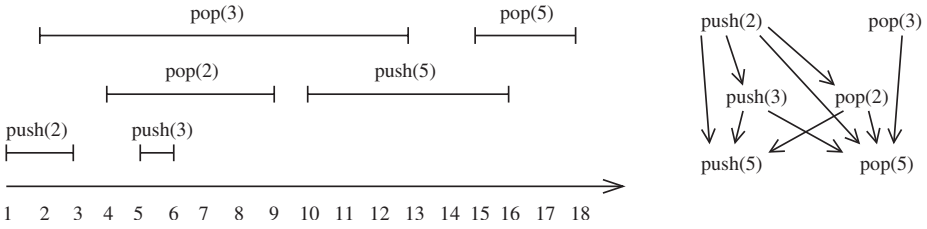

Fig. 3. An interval order. On the left, the interval representation of the left trace in Figure 2. On the right, the corresponding partial order.

an *interval order*). Two operations with non-overlapping time intervals must appear in the merged total order in the order of their intervals.

The per-processor traces on the left of Figure 2 correspond to a valid execution of the shared stack (respecting the time intervals). Namely,

Push(2,1,3), Pop(2,4,9), Push(3,5,6), Pop(3,2,13), Push(5,10,16), Pop(5,15,18)

and

Push(2,1,3), Push(3,5,6), Pop(3,2,13), Pop(2,4,9), Push(5,10,16), Pop(5,15,18)

are both valid executions. The per-processor traces on the right, however, are invalid, because there is no possible merging of the traces respecting the time intervals such that each pop operation returns the value at the top of the stack. Specifically, any trace respecting the time intervals must begin with Push(2, 1, 3), Push(3, 5, 6), Pop(2, 7, 9), and this last pop does not return the value 3 at the top of the stack.

The general combinatorial problem we define and consider is the following. Each operation on the data structure is represented as a (processor ID, operation description) pair, e.g., (P3, Pop(5)). We are given a trace comprising a set of operation pairs and a partial order on these pairs that reflects ordering constraints that must be preserved. The goal is to determine if there is a topological sort of the operations (i.e., a total order that is consistent with the given partial order) such that the sorted sequence of operations is a legal serial execution of the data structure, in order to distinguish valid executions from invalid executions. We denote this problem as the *Testing Parallel Executions* problem.

1.1. *A Systematic Study of Testing Parallel Executions of Basic Data Structures*

This paper presents the first systematic study of algorithms and hardness results for the testing parallel executions problem, focusing on queues, priority queues, stacks, and counters. The results of our study are summarized in Table 1. We consider varying restrictions on both the data value types and the class of partial orders being considered. For value types, we consider both *distinct values*, where we are guaranteed that each value inserted into the data structure is distinct, and *arbitrary values*, where there is no such guarantee.

Table 1. Results for *testing parallel executions* of various data structures, with distinct and arbitrary value types and various partial orders.*

Data object	Value type	Partial order	Result	Citation
Queue	distinct	interval orders	$O(n)$ time	Theorem 3.1
Priority queue	distinct	interval orders	$O(n \log n)$ time	Theorem 3.2
Queue	distinct	p chains	$O(np^2)$ time	Theorem 3.5
Queue, Priority queue	distinct	any	$O(n^3)$ time	Theorem 3.6
Stack	distinct	any	poly-time	Theorem 3.7
Queue, Priority queue	arbitrary	interval orders	NP-complete	Theorem 4.2
Stack	arbitrary	interval orders	NP-complete	Theorem 4.4
Queue, Priority queue, Stack	arbitrary	chains	NP-complete	Theorem 4.1
Counter	arbitrary	interval orders	$O(n)$ time	Theorem 5.1
Counter	arbitrary	chains, series-parallel	$O(n \log n)$ time	[MS]
Counter	arbitrary	any	NP-complete	[AW]
Counter	distinct	any	NP-complete	Corollary 5.4
Fetch&Add counter	arbitrary	interval orders, chains	NP-complete	[GK]
Counter with reads	arbitrary	interval orders	NP-complete	Theorem 5.8
Counter with reads	arbitrary	chains	NP-complete	Theorem 5.7
Arbitrary	arbitrary	interval orders	NP-complete	[WG]

*As discussed in the text, we focus on interval orders and chains, due to their correspondence to linearizable data structures and "sequentially consistent" data structures, respectively. Sequential running times are shown, where n is the number of operations on the data structure. Three of the results are immediate corollaries of known results on other problems, and hence the citation is given to the known result. The previous result specifically addressing this problem is given in the last row of the table. All other results are new.

Our study focuses on two partial orders of practical importance: (1) unions of chains and (2) interval orders. A union of chains arises, as in Figure 1, when there is a total order for each processor, but no a priori ordering between processors. This is the partial order that must be preserved in *sequentially consistent* [La] shared memory multiprocessors. An interval order arises, as in Figure 3, when there are time intervals for each operation defining an order that must be preserved in the merging. This is the partial order that must be preserved for a *linearizable* [HW] data object.[2] Linearizable data objects (also known as *atomic* objects) are well-studied (see, e.g., Chapter 13 of [Ly]), as they have a number of desirable properties. For comparison, we also consider more general partial orders such as a series-parallel order (modeling some form of fork-join parallelism) or even an arbitrary partial order.

We present fast algorithms for testing parallel executions under a number of important scenarios, and NP-completeness results for other related scenarios. Thus our results help clarify the threshold between scenarios that admit polynomial time solutions and those that are NP-complete. They also show the advantages of inserting distinct values into the data structure (as appropriate). Our algorithms are the first nontrivial algorithms for these problems, and require several new techniques that may be useful to other testing scenarios and other data structures.

[2] We use *data structure* and *data object* interchangeably. Also, *processor* can be viewed as a synonym for *process*.

1.2. *Previous Work*

Wing and Gong [WG] studied the problem of *testing parallel executions* of arbitrary linearizable shared data structures. The problem of testing arbitrary linearizable data structures is shown to be NP-complete, and an exponential time algorithm is devised. They also developed a simulation environment for implementing their testing algorithms. Sullivan and coworkers [SM1], [SM2], [BS] defined and studied *certification trails* for testing sequential executions of balanced binary trees, priority queues, union-find structures, and mergeable priority queues. In this approach the data structure code is modified to output additional information to assist in testing. Other work on sequential testing and related issues includes [HA], [DRT], and [Ra]. Note that unlike testing sequential executions, testing parallel executions focuses on topological sorting since it does not assume a centralized serialization point or a central module implementing the data structure. The sequential trace work, in contrast, focuses on testing procedures that are more efficient in time and/or space than the implementation being tested. These are also the concerns of the works on sequential program checking (e.g., [BK1], [BLR], and [BEG$^+$]). In a recent independent such work, Finkler and Mehlhorn [FM] presented algorithms for checking sequential priority queues. Their algorithms observe the *sequential* stream of operations at the data structure, and check to see if this stream is legal; in contrast, our main task is to determine if there exists a legal sequential stream corresponding to the collection of parallel streams that we observe.

Papadimitriou [Pa] proved that testing the serializability of database transactions is NP-complete. Gibbons and Korach [GK] studied testing a shared memory for sequential consistency or linearizability under a range of scenarios. They did not explicitly consider testing executions of data structures. Rubinfeld [Ru] presented a number of results on efficient parallel program checking. The model studied is a natural parallelization of the sequential program checking model [BK1]. The checker for a parallel program P may invoke P on other inputs, as part of the checking process, and the goal is to have checkers that are more efficient than P in time and/or processors (ignoring the cost of invoking P). Rubinfeld studied various single- and multi-output functions (e.g., majority and all pairs shortest paths), but did not consider parallel streams of data structure operations. Other work on testing and related issues for parallel machines includes [BA], [BB], [BRS$^+$], [AGMT], and [BC].

All these previous works consider testing in the context of a single run of the program, as in this paper. This has the advantage of testing an actual run of the implementation under real conditions, not an abstraction. On the other hand, we learn only about the correctness of the particular run.

There are a variety of well-known tools for bug detection that complement the postmortem testing procedures considered in this paper. One can formalize the implementation at some level of abstraction and then prove the correctness of the abstracted implementation, either by hand or by using formal verification tools. This approach has the advantage of testing all possible executions, but the disadvantage that for complex implementations, proofs by hand are time consuming and error prone and verification tools can suffer from state explosion problems. Moreover, the abstracted implementation omits details of the real implementation and makes assumptions that may fail to hold for the real implementation. Thus bugs may arise even in implementations with "proofs"

of correctness. Finally, one can use programming environment tools to step through the implementation at the program instruction level; however, this approach may be quite involved as well, and it may mask timing errors due to its interfering nature.

The remainder of this paper is organized as follows. Section 2 presents some preliminary definitions. Sections 3 and 4 present results for testing parallel executions of queues, priority queues, and stacks, with distinct values and with arbitrary values, respectively. Section 5 presents results for testing parallel executions of various types of counters. Finally, Section 6 describes extensions to the testing model, and some further results.

A preliminary version of this paper appeared in the *Proceedings of the* 11*th ACM Symposium on Parallel Algorithms and Architectures* [GBP].

2. Preliminaries

A parallel program is any program for a collection of processors. A run of a parallel program defines a history of operations on a shared data structure. This history is represented by a quadruple $(U, <, \pi, op)$, called a *trace*. U is a finite set of elements called *events*, and $<$ is a partial order over U. In particular, $\alpha < \beta$ whenever we know that α happened before β.[3] Processor ids and operations are associated with events by the functions π and op, respectively, with domain U. For simplicity in reporting running times, we assume that all values, timestamps, etc., are $O(\log n)$-bit integers, unless noted otherwise. Extensions to handle more general values and timestamps are straightforward and affect only the running times.

An *event sequence* $\sigma = \alpha_1, \ldots, \alpha_k$ is a sequence of *distinct* events from U. If σ is an event sequence, then $\{\sigma\}$ denotes the set of elements in σ, and $<_\sigma$ is the total order on U where $\alpha_i <_\sigma \alpha_j$ if and only if $i < j$. A *topological sort* (or *schedule*) of the trace $(U, <, \pi, op)$ is an event sequence σ such that $\{\sigma\} = U$ and for all $\alpha, \beta \in U$, $\alpha < \beta$ implies $\alpha <_\sigma \beta$.

We consider various families of partial orders $<$. A *chain* on a set of events is a total order on those events. In a *union of chains*, or simply *chains*, partial order, the events are partitioned into subsets such that the events within each subset are totally ordered and no other events are ordered. An *interval order* is a partial order defined by a set of intervals on a line, where an event α is ordered before an event β if and only if the intervals for α and β are non-overlapping and the interval for α precedes the interval for β. An example was depicted in Figure 3. A *series-parallel* partial order is defined by a series-parallel directed acyclic graph (DAG). A series-parallel DAG is defined inductively, as follows. The graph consisting of a single node is series-parallel. If G_1 and G_2 are series-parallel, then so is the graph obtained by adding to $G_1 \cup G_2$ a directed edge from the sink node of G_1 to the source node of G_2. (A sink node in a graph has outdegree 0; a source node has indegree 0.) Finally, if $G_1, \ldots, G_k$, $k \geq 2$, are series-parallel, then so is the graph obtained by adding to $G_1 \cup \cdots \cup G_k$ a new source node, u, with a directed edge from

[3] We are not concerned in this paper with the program semantics behind the partial order $<$, e.g., the precise definition of "happened before." We only assume that the partial order $<$ is given, and must be respected by the topological sort.

u into each of the source nodes of $G_1, \ldots, G_k$, and a new sink node, v, with a directed edge into v from each of the sink nodes of $G_1, \ldots, G_k$. A series-parallel partial order corresponding to a series-parallel DAG orders any two nodes with a directed path in the DAG.

Events can often be labeled as "insert" or "delete" events. For example, push (enqueue) is an insert event and pop (dequeue) is a delete event for a stack (queue, respectively). We say that a pair of events, (e, e'), is an *event pair* if e is an insert event and e' is a delete event for the same value v.

The "correctness" of a trace depends on the serial semantics of the data structure and, accordingly, we say a trace is *valid* if the semantics of the data object are preserved, i.e., there is a topological sort of the trace that is a legal serial execution for the data object. For simplicity in exposition, we require that in order for a trace with insert and delete events to be valid, there must be a 1–1 correspondence between insert and delete events.[4] Note that only event pairs can be corresponding events. To illustrate the correctness condition, consider the following three examples.

Stacks. A trace for a stack is valid if there exists a topological sort σ and a 1–1 correspondence between push events and pop events with the same value such that (i) for each corresponding pair, the push event a appears before the pop event a' in σ, and (ii) a pop event a' appears before a pop event b' in σ if and only if either

1. a' appears before the push event b corresponding to b' in σ (i.e., we have a precedes a' precedes b precedes b'), or
2. the push event b appears before the push event a (i.e., we have b precedes a precedes a' precedes b').

This captures formally the requirement that a pop operation must return the value at the top of the stack. Examples of valid and invalid stack traces were given in Figures 1 and 2.

Queues. A trace for a FIFO queue is valid if there exists a topological sort σ and a 1–1 correspondence between enqueue events and dequeue events with the same value such that (i) for each corresponding pair, the enqueue appears before the dequeue in σ, and (ii) an enqueue event a appears before an enqueue event b in σ if and only if the dequeue event a' corresponding to a appears before the dequeue event b' corresponding to b in σ. (That is, we have either a precedes a' precedes b precedes b' or a precedes b precedes a' precedes b'.) Examples of valid and invalid FIFO queue traces are given in Figure 4.

The three-processor trace on the left in Figure 4 is valid: consider the unique 1–1 correspondence between enqueues and dequeues with the same value and the topological sort Enqueue(2), Enqueue(3), Dequeue(2), Enqueue(1), Dequeue(3), Dequeue(1), Enqueue(5), and Dequeue(5). The three-processor trace on the right is invalid: there is no topological sort satisfying both properties (i) and (ii).

[4] Extensions to handle insert events appearing without corresponding delete events are straightforward and left to the reader.

$\underline{p_1}$	$\underline{p_2}$	$\underline{p_3}$	$\underline{p_1}$	$\underline{p_2}$	$\underline{p_3}$
Enqueue(2)	Dequeue(2)	Dequeue(3)	Enqueue(2)	Dequeue(2)	Dequeue(5)
↓	↓	↓	↓	↓	↓
Enqueue(3)	Dequeue(1)	Dequeue(5)	Enqueue(3)	Dequeue(1)	Dequeue(3)
↓	↓		↓	↓	
Enqueue(1)	Enqueue(5)		Enqueue(1)	Enqueue(5)	

Fig. 4. Valid versus invalid queue traces.

Priority Queues. A priority queue supports insert and deletemax operations (alternatively, deletemin operations). A trace for a priority queue is valid if there exists a topological sort σ and a 1–1 correspondence between insert events and deletemax events with the same value such that (i) for each corresponding pair, the insert appears before the deletemax in σ, and (ii) if the value, v, for an insert a and its corresponding deletemax a' is greater than the value w for an insert b and its corresponding deletemax b', and b' precedes a' in σ, then b' precedes a in σ. (That is, if v is larger than w but w was removed from the priority queue before v, then w must have been removed before v was even inserted.)

3. Distinct Values

In this section we present results for distinct-values traces. A *distinct-values trace* is a trace $(U, <, \pi, op)$ in which every event in U appears in at most one event pair. As we shall see, testing parallel executions of distinct-values traces is easier than testing parallel executions of traces with arbitrary values since there is no ambiguity in the target 1–1 correspondence between insert and delete events.

3.1. *Testing Linearizable Queues*

Let $(U, <, \pi, op)$ be a distinct-values trace where the parallel data structure is a FIFO linearizable queue. The operations are either enqueue(x) or dequeue(x) for an integer x. Two events $b, b' \in U$ are an event pair (b, b') if and only if b is an enqueue(x) operation and b' is a dequeue(x) operation for the same value x. Associated with each event α is an interval $[t_1, t_2]$ where t_1 and t_2 are integral timestamps with t_1 less than t_2. We denote these timestamps as $[_\alpha$ and $]_\alpha$, respectively. We assume that no two events in U have any timestamps in common (to ensure this across all processors, we can break ties using π). The event intervals implicitly define the partial order (interval order) $<$: for all $\alpha, \beta \in U$, $\alpha < \beta$ if and only if $]_\alpha < [_\beta$. We say α is *active* at t if $[_\alpha \leq t \leq]_\alpha$.

We seek a topological sort of $(U, <, \pi, op)$ which preserves the serial semantics of a FIFO queue. Accordingly, we define a *queue sort* of a distinct-values trace $(U, <, \pi, op)$ to be a topological sort σ of $(U, <, \pi, op)$ satisfying the requirements for queues presented in Section 2. Namely, for all event pairs (a, a'), $a <_\sigma a'$, and for all other event pairs (b, b'), we have $a <_\sigma b$ if and only if $a' <_\sigma b'$.

The following algorithm (Algorithm $\mathcal{A}$) determines whether or not there exists a queue sort, in linear time.

Algorithm $\mathcal{A}$

Given: A distinct-values trace for a (possibly buggy) linearizable FIFO queue.
Question: Does there exist a queue sort σ of the trace?

1. Sort the timestamps in the trace and place in an array A.
2. Match up event pairs. If there exists either an event that is not in an event pair or an event pair (e, e') such that $]_{e'} < [_e$ (i.e., the dequeue event ends before the enqueue event begins), return NO.
3. For all i, if $A[i] =]_{e'}$, then $B[i] := [_e$, where (e, e') is an event pair, and otherwise, $B[i] := 0$.
4. Compute the prefix max of B: For all i, let $C[i] := \max_{1 \le j \le i} B[j]$.
5. If there exist i such that $A[i] = [_{e'}$ and $]_e < C[i]$, where (e, e') is an event pair, return NO. Otherwise, return YES.

Theorem 3.1. *Given a distinct-values trace with n events, Algorithm $\mathcal{A}$ runs in $O(n)$ time and returns YES if and only if there exists a queue sort of the trace.*

Proof. The time complexity is immediate, since integer sorting can be used in step 1. It is also immediate that if step 2 returns NO, there can be no queue sort. The semantics of a queue require that in σ, a value v is enqueued before a value w if and only if v is dequeued before w. Thus for any event pair (e, e'), if there exists an event pair (d, d') such that $]_e < [_d$ and $]_{d'} < [_{e'}$, then $e <_\sigma d$ and $d' <_\sigma e'$, contradicting σ being a queue sort. Equivalently, if for any event pair (e, e'), we consider all event pairs (d, d') such that $]_{d'} < [_{e'}$ and take the maximum $[_d$ among them, then there is no queue sort if $]_e < [_d$. It follows that if step 5 returns NO, there can be no queue sort.

It remains to show that there exists a queue sort if step 5 returns YES. Here the argument is more involved. We first define a graph G representing a subset of the constraints implied by the traces. We then show that a greedy schedule of G that favors enqueues over dequeues can be used to produce a queue sort, as long as G is acyclic. Finally, we show that G is acyclic unless it contains an "anti-time partner" edge, precisely of the type checked for in step 5.

Consider a graph G where each event in U is a node and there is an edge from node α to node β if $\alpha < \beta$, i.e., if $]_\alpha < [_\beta$. Denote these edges as "time" edges. Next, for each pair of event pairs (d, d') and (e, e') such that $]_{d'} < [_{e'}$, we know by the property of queues that d must precede e in any queue sort. Thus, if there is no time edge from node d to node e, then let G have an edge from d to e, denoted a "partner" edge. Note that partner edges are only between enqueue events.

Consider the following greedy schedule. At each step of the schedule, we say an event in G is *eligible* if it is unscheduled and all its predecessors in G have been scheduled. Repeat until all events are scheduled: Schedule an eligible enqueue, adding its value to the tail of the queue. If none, schedule the eligible dequeue for the value that is presently at the head of the queue. If no such eligible dequeue, report FAILURE.

Note that if the greedy schedule succeeds in scheduling all the events in G, then it produces a queue sort. This is because it only schedules eligible events (and hence the interval order is respected), and it only schedules dequeues that correspond to the current head of queue.

We next argue that if there are no cycles in G, then the greedy schedule succeeds. Suppose to the contrary that it reports FAILURE, and consider the state of the scheduling when FAILURE is reported. There are no eligible enqueues and there is no dequeue for a value that is presently at the head of the queue. Let t be the smallest timestamp such that $t =]_\alpha$ for an unscheduled event α. Note that all events β such that $]_\beta < t$ have been successfully scheduled. Thus there are no time edges preventing α (or any other unscheduled event φ such that $[_\varphi < t$) from being eligible. There are two cases:

1. Event α is an enqueue e. Since e is not eligible, it must have at least one unscheduled ancestor. Consider the subgraph, H_e, of unscheduled ancestors of e. From the arguments above, we have that there are no time edges in H_e. Moreover, any partner edges are between enqueues only, and there are no cycles in H_e because there are no cycles in G. It follows that at least one enqueue in H_e has no incoming edges and hence is eligible, a contradiction.
2. Event α is a dequeue d'. Let (d, d') be an event pair. Suppose d is not scheduled, and consider the subgraph, H_d, of unscheduled ancestors of d. Due to step 2 of the algorithm, we know that $[_d < t$. Thus, as argued in the previous case, there must be an eligible enqueue in H_d, a contradiction. So suppose instead that d is scheduled. Let (a, a') be the event pair such a is the (scheduled) enqueue event whose value is at the head of the queue. By assumption, $d \neq a$ and a' is not eligible. Since the value of a is at the head and not the value of d, we know that a was scheduled before d. If $t < [_{a'}$, then $]_{d'} = t < [_{a'}$, so G would have a partner edge from d to a, contradicting a being eligible for scheduling before d. Otherwise, $[_{a'} < t$. As argued above, there are no time edges into a' from unscheduled events. Moreover, a' is a dequeue, so there are no partner edges into a' either. Thus a' is eligible, a contradiction.

Therefore, the greedy schedule succeeds if there are no cycles in G.

It remains to show there are no cycles in G. Suppose to the contrary that there exists a cycle C in G. We will show that this implies the existence of a two-cycle C' of enqueue events formed by one time edge and one (anti-time) partner edge. Note that (i) time edges are transitive, (ii) partner edges are defined by time edges among dequeues, and hence they are also transitive, and (iii) there can be no cycles of only time edges either among the enqueues or the dequeues. It follows that there also exists a cycle C' such that time edges and partner edges alternate. Thus C' has at least one time edge and at least one partner edge. Since partner edges are only among enqueues, no dequeue can be part of any cycle that alternates time edges and partner edges. Thus C' has only enqueues.

Let S be the set of enqueues e in C' with an incoming time edge. Let $r \in S$ be such that $]_{r'} \leq]_{x'}$ for all $x \in S$, where (r, r') and (x, x') are event pairs. Let q be the predecessor of r in C' (by a time edge), and let p be the predecessor of q (by a partner edge). Since the cycle alternates, we have that $p \in S$. Let (p, p') and (q, q') be event pairs. The partner edge from p to q implies that $]_{p'} < [_{q'}$, and hence by the definition of r, $]_{r'} < [_{q'}$. Thus G has a partner edge from r to q that forms a two-cycle with the

time edge from q to r; we denote this an "anti-time partner" edge. Since step 5 returned YES, there are no events resulting in anti-time partner edges in G, a contradiction. Thus there are no cycles in G.

Therefore, Algorithm $\mathcal{A}$ returns YES if and only if there exists a queue sort of the trace. □

Note that in order to maintain linear time, our algorithm did not materialize the edges in G. In fact, it can be shown that even a transitive reduction of the edges in G can still have $\Theta(np)$ edges, using an example similar to that considered in Section 3.3.

3.2. *Testing Linearizable Priority Queues*

Let $(U, <, \pi, op)$ be a distinct-values trace where the parallel data structure is a linearizable priority queue, supporting *insert* and *deletemax* operations, where $v(\alpha)$ is the component of $op(\alpha)$ corresponding to the value inserted into the queue by event $\alpha \in U$. A *priority-queue sort* of a distinct-values trace $(U, <, \pi, op)$ is a topological sort σ of $(U, <, \pi, op)$ satisfying the requirements for priority queues presented in Section 2. Namely, for all event pairs (a, a'), $a <_\sigma a'$, and for all other event pairs (b, b'), if $v(a)$ is greater than $v(b)$ and $b' <_\sigma a'$, then $b' <_\sigma a$.

Priority queues raise additional complications not present with queues. For a priority queue, if (e, e') and (f, f') are both event pairs, and $v(e)$ is greater than $v(f)$, we have either $e' <_\sigma f'$ or $f' <_\sigma e$. The difficulty in extending Algorithm $\mathcal{A}$ and its correctness proof is that partner edges now go from dequeues to enqueues, so we can no longer argue that the partner ancestors of an unscheduled enqueue are all enqueues and hence some enqueue must be eligible. Instead we devise a new approach. We observe that for each event pair (e, e'), if $]_e < [_{e'}$, then $v(e)$ must be in the queue at least from time $]_e$ to time $[_{e'}$, and hence a lower bound on the maximum value in the queue at time t can be determined by computing the maximum $v(e)$ such that $]_e \le t < [_{e'}$. We denote this lower bound over all timestamps as the *minmax skyline*. Then for any event pair (f, f'), f' cannot be scheduled during any time for which $v(f)$ is less than the minmax skyline (since it could never be the max in the queue at that time). We show below that for a priority-queue sort, the necessary condition that there exists a t between $[_{f'}$ and $]_{f'}$ such that the minmax skyline is at most $v(f)$, is also a sufficient condition.

As an example, consider the following trace, sorted by timestamp, and its minmax skyline, where $v(a) < v(b) < v(c)$:

trace:	$[_a$	$[_b$	$[_c$	$]_b$	$[_h$	$[_{h'}$	$]_a$	$[_{b'}$	$]_c$	$[_{a'}$	$]_h$	$]_{h'}$	$[_{c'}$	$]_{a'}$	$]_{c'}$	$]_{b'}$
skyline:	0	0	0	$v(b)$	$v(b)$	$v(b)$	$v(b)$	$v(a)$	$v(c)$	$v(c)$	$v(c)$	$v(c)$	0	0	0	0

The skyline is depicted pictorially in Figure 5. If $v(h) < v(a)$ in the trace, there is no priority-queue sort (the minmax skyline is greater than $v(h)$ throughout the range from $[_{h'}$ to $]_{h'}$). Otherwise, $v(h) > v(a)$ and $b, a, b', h, h', c, c', a'$ is a priority-queue sort.

Algorithm $\mathcal{B}$

Given: A distinct-values trace for a (possibly buggy) linearizable priority queue.
Question: Does there exist a priority-queue sort σ of the trace?

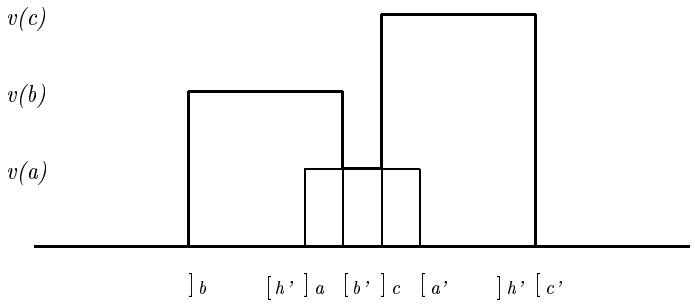

Fig. 5. The minmax skyline for the example trace.

1. Sort the timestamps in the trace and place in an array A.
2. Match up event pairs. If there exists either an event that is not in an event pair or an event pair (e, e') such that $]_{e'} < [_e$, return NO.
3. Compute the minmax skyline B: Scan through A in order, starting with an initially empty queue. For each $A[i]$, insert $v(e)$ into the queue if $A[i] =]_e < [_{e'}$ and delete $v(e)$ if $A[i] = [_{e'} >]_e$, where (e, e') are an event pair. Let $B[i] :=$ the maximum value in the queue after processing $A[i]$.
4. Preprocess B for range minima queries (see below).
5. If there exists an event pair (e, e') such that the minimum value in B in the entire range $\max([_e, [_{e'})$ to $]_{e'}$ is greater than $v(e)$, return NO. Otherwise, return YES.

A *range minima* query for an array $B[1 \cdots n]$ takes as input two endpoints $t_1, t_2 \in [1 \cdots n]$ defining an interval range, and returns the minimal value among $B[t_1]$, $B[t_1 + 1], \ldots, B[t_2]$. Gabow et al. [GBT] have shown that an array can be preprocessed in linear time such that any range minima query can be answered in constant time.

Theorem 3.2. *Given a distinct-values trace with n events, Algorithm $\mathcal{B}$ runs in $O(n \log n)$ time and returns YES if and only if there exists a priority-queue sort of the trace.*

Proof. Clearly, if step 2 returns NO, there is no priority-queue sort. Likewise, in step 5, since e' can be scheduled no earlier than e, then it must be scheduled in the given range. Thus as argued above, if step 5 returns NO, there is no priority-queue sort. If step 5 returns YES, then consider the following scheduling algorithm: Schedule each dequeue e' at the smallest $t \geq \max([_e, [_{e'})$ such that $v(e)$ is at least the minmax skyline at t, where (e, e') is an event pair. Since step 5 returned YES, we know that $t \leq]_{e'}$, as required. (If multiple dequeue events satisfy the criterion for being scheduled at the same t, break ties arbitrarily.) Then schedule each enqueue e at the largest $t \geq [_e$ such that $t \leq]_e$ and e is scheduled before e'. We leave to the reader to verify that this schedule ensures that the actual skyline corresponds to the minmax skyline and hence that the schedule is a priority-queue sort.

As for the time bound, step 3 can be implemented using a sequential heap data structure in $O(n \log n)$ time. All other steps take linear time, using integer sorting for steps 1 and 2, and Gabow et al.'s range minima algorithm for steps 4 and 5. □

3.3. *Testing Sequentially Consistent Queues*

Let $(U, <, \pi, op)$ be a distinct-values trace for a possibly buggy FIFO queue. For the popular scenario in which the partial order $<$ is the union of p chains, we present an $O(np^2)$ time algorithm (Algorithm $\mathcal{C}$ below) that returns a queue sort of $(U, <, \pi, op)$ whenever it exists. For the common case of $p \ll n$, this significantly improves upon our $O(n^3)$ time algorithm (Algorithm $\mathcal{D}$ in Section 3.4 that follows) for arbitrary partial orders. Both algorithms incrementally generate a set of ordering constraints between events, which must be respected by any queue sort. Algorithm $\mathcal{C}$ improves upon Algorithm $\mathcal{D}$ by maintaining only a sparse set of constraints and avoiding repeated transitive closures.

Let an *interchain* constraint be a constraint between two events on different chains. In Algorithm $\mathcal{C}$, steps 2–5 construct a graph G whose edges correspond to a sparse set of scheduling constraints. These constraints are initially only the chains $<$, such that if $v_1 < v_2 < \cdots < v_k$ is a chain, then the chain is represented by k nodes and $k - 1$ directed edges:

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$$

(i.e., without any transitively implied edges). As the algorithm proceeds, additional edges are added to G representing interchain constraints between enqueue events, as implied by other constraints. The key idea is in step 5, where we process the events in G from the leaves, maintaining the invariant that each processed enqueue node w has exactly one outgoing edge per chain reachable from w, such that the edge corresponds to the earliest event on the chain reachable from w. We show that although processing nodes adds new edges, no backtracking is necessary, and moreover the resulting G suffices for a greedy scheduling of G in step 8 that succeeds if and only if we have a valid trace.

Algorithm $\mathcal{C}$

Given: A distinct-values trace for a (possibly buggy) FIFO queue, where the partial order $<$ is the union of p chains.
Question: Does there exist a queue sort σ of the trace?

1. Match up event pairs. If there exists an event that is not in an event pair, return NO.
2. We will construct a graph, G, where the nodes of G are the events and there is a directed edge in G from event α to event β only if we have the constraint $\alpha <_\sigma \beta$. Initialize G to be the p chains of $<$ (without any transitively implied edges, as discussed above).
3. Consider each pair of events a', b on the same chain such that a' is a dequeue event, b is an enqueue event, and b is the event after a' on the chain (i.e., $a' < b$ and there is no event α on the chain such that $a' < \alpha < b$). Let (a, a') be an event pair. If a and b are on different chains, add an interchain edge from a to b. See Figure 6. (This constraint is valid because $a' < b$ implies $a <_\sigma a' <_\sigma b$.) Otherwise, if b precedes a on the same chain, return NO.

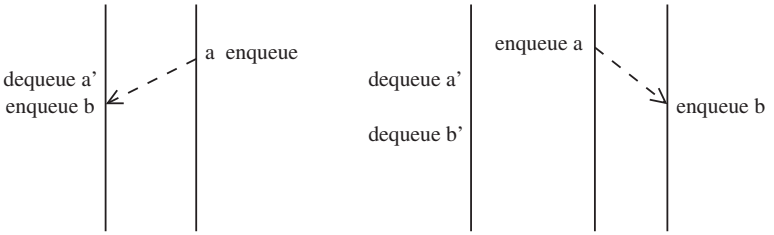

Fig. 6. On the left, step 3 of Algorithm $\mathcal{C}$. On the right, step 4 of Algorithm $\mathcal{C}$. Chains are shown as lines, where ancestors appear above descendants on these chains. The dashed line is the new interchain edge.

4. Consider each pair of dequeue events a', b' on the same chain such that b' is the earliest *dequeue* event after a' on the chain (i.e., $a' < b'$ and there is no dequeue event c' such that $a' < c' < b'$). Let (a, a') and (b, b') be event pairs. If a and b are on different chains, add an interchain edge from a to b. See Figure 6. (This constraint is valid because for a FIFO queue, $a' < b'$ implies $a <_\sigma b$.) Otherwise, if b precedes a on the same chain, return NO.
5. Mark all events as unprocessed. We process G starting with its leaves. We maintain the invariant that each processed enqueue node w has exactly one outgoing edge per chain reachable from w, and that edge is to the earliest node on the chain reachable from w by any path in G.
   Repeat while there is an unprocessed node in G whose children are all processed:
   (a) Select an unprocessed node w in G whose children are all processed. If w is a dequeue event, mark w as processed and repeat. Otherwise, w is an enqueue event. Let chain i be the chain containing w.
   (b) Let $C(w)$ be the set of children of w on other chains together with the earliest enqueue event after w on chain i. Discard the interchain edges outgoing from w, and replace them with edges to the earliest event at each chain j other than i that is either (1) a node in $C(w)$ on chain j or (2) a child in chain j of a node in $C(w)$ not in chain j. See Figure 7. (These new constraints are valid because they are transitively implied by existing constraints.)

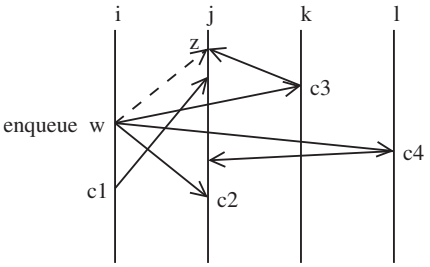

Fig. 7. Step 5(b) of Algorithm $\mathcal{C}$. Four chains are shown as lines labeled i, j, k, l, where ancestors appear above descendants on these chains. Shown are the edges relevant to adding a new edge from w to chain j. $C(w) = \{c1, c2, c3, c4\}$. By construction, $c1$ is the first enqueue succeeding w on its chain, and all of $c1, c2, c3, c4$ have been processed. The new edge from w to z (shown as a dashed edge) replaces the edge from w to $c2$.

(c) For all a such that there is an interchain edge from w to a: Let c' be the latest dequeue event before w on chain i, and let d' be the earliest dequeue event after a on the same chain as a, if they exist. If c and d are on different chains, add an edge from c to d. (This constraint is valid because $c' <_\sigma w <_\sigma a <_\sigma d'$, and for a FIFO queue, $c' <_\sigma d'$ implies $c <_\sigma d$.) Otherwise, if d is earlier than c on the same chain, then return NO.

(d) The previous step may have added a second edge from c to the chain containing d. Given the chain edges, one of these two edges is transitively implied by the other, so we remove it. Specifically, if there is an event x on the same chain as d such that x is a child of c, then discard the edge from c to d if x is earlier than d on the chain and otherwise discard the edge from c to x.

(e) Mark w as processed.

6. If G has an unprocessed node, return NO.
7. Let Q be the empty queue and let σ be an empty topological sort. If all of the parents of an event e have been placed in the topological sort, then e is *eligible* to be added to the topological sort. Let S be the set of eligible enqueue events, i.e., events b with indegree zero in G.
8. Repeat the following until G is empty:
   (a) If S is nonempty, select and remove from S and G an enqueue event a. Add a to the topological sort σ, and append $v(a)$ to Q. For each child b of a such that b is an enqueue event, check to see if b is now eligible, and, if so, add b to S.
   (b) If S is empty, let h be the enqueue event such that $v(h)$ is at the head of Q, and let (h, h') be an event pair. If there is no such h since Q is empty, or h' is not eligible, return NO. Otherwise, remove h' from G, add h' to σ, and dequeue $v(h)$ from Q. Check to see if the successor of h' on its chain (if any) is now eligible, and, if so, add it to S.
9. Return σ.

To establish the correctness of Algorithm $\mathcal{C}$, we first prove the following two lemmas.

Lemma 3.3. *The following three properties hold for Algorithm* $\mathcal{C}$:

(1) *All added edges must be satisfied by all queue sorts of* $(U, <, \pi, op)$.
(2) *All discarded edges are transitively implied by the remaining edges.*
(3) *The order of events added to* σ *in step* 8 *is a topological sort of* G.

Proof. These three properties can be proved by easy inductive arguments, left to the reader. The main ideas are as follows:

(1) This was argued above at each step in the algorithm.

(2) The key step to argue here is step 5(b). In this step, however, each discarded edge from w to a child c is replaced by an edge from w to a child z earlier on the same chain. Hence the discarded edge is transitively implied by the new edge and the chain edges.

(3) This holds because all events in S are eligible, and we only schedule events that are either in S or eligible dequeue events. □

Lemma 3.4. *Throughout step* 5, *each processed enqueue node w has exactly one outgoing edge per chain reachable from w, and that edge is to the earliest node on the chain reachable from w by any path in G. Moreover, w can only reach processed nodes.*

Proof. The proof is by induction on the iterations of step 5. Initially, there are no processed nodes, so the invariant holds. Consider the iteration of step 5 that culminates with w being marked as processed. At step 5(b), all the nodes in $C(w)$ are processed, for otherwise w would not have been selected for this iteration. Thus inductively, we have that each node y in $C(w)$ has outgoing edges to the earliest (processed) node on each chain that can be reached from y by any path in G. Thus it suffices in step 5(b) to consider only $C(w)$ and the children of $C(w)$ in determining the earliest node on each chain that is reachable from w. Thus the desired property holds for w after step 5(b).

Any edge added in step 5(c) is outgoing from an unprocessed node. To see this, consider an edge from c to d added during the processing of w. Let x be the enqueue event after c' on the same chain as w (x is either w or earlier than w on the same chain). Thus previously in step 3, we would have added an edge from c to x, creating a directed path from c to w. By property (2) of Lemma 3.3, there is a directed path from c to w in the current G. Since w is just now being processed, its ancestor c has not yet been processed. Since w's children are processed and can reach only processed nodes, w cannot reach c and hence the new edge does not alter the set of nodes reachable from w. Moreover, step 5(d) involves only outgoing edges from c. Thus the desired property holds for w at the end of the iteration.

It remains to show that processing w did not violate the invariant for an already processed enqueue e. At the start of the iteration, w is unprocessed, so, inductively, e cannot reach w or any other unprocessed node. Step 5(b) adds and removes edges outgoing from w, and steps 5(c) and (d) add and remove edges outgoing from unprocessed nodes (as argued above). Thus the desired property holds for e at the end of the iteration. □

We are now ready to prove the correctness of Algorithm $\mathcal{C}$ and show that it runs in $O(np^2)$ time.

Theorem 3.5. *Let $(U, <, \pi, op)$ be a distinct-values trace with n events such that $<$ is the union of p chains. Algorithm $\mathcal{C}$ runs in $O(np^2)$ time and returns a queue sort of $(U, <, \pi, op)$ if and only if one exists.*

Proof. In a simple $O(n)$ time preprocessing of each chain, we can label each event with its chain, its rank within its chain, and the index of its next and previous dequeue and enqueue events on its chain. By Lemma 3.4, we maintain $O(p)$ edges per node. Thus all but step 5(b) can be done in $O(np)$ time. Step 5(b) requires $O(p^2)$ time per node, to examine the $O(p)$ children and $O(p^2)$ grandchildren, for a total time bound of $O(np^2)$.

We next show that if the algorithm returns a σ, then σ is a queue sort. By Lemma 3.3, we have that σ is a topological sort of G and since G contains the chain edges $<$, it preserves $<$. Moreover, step 8 mimics a sequential FIFO queue, such that a dequeue is scheduled only if its value is at the head of the queue. Thus σ is a queue sort.

Finally, we show that if the algorithm returns NO in steps 1, 3, 4, 5(c), 6, or 8, the trace is invalid. Step 1 returns NO only if we discover an event not in an event pair, so clearly the trace is invalid. Steps 3, 4, and 5(c) return NO only if we discover two enqueue events such that one must precede the other in σ and vice versa, so the trace is invalid. Step 6 returns NO only if each unprocessed node has at least one unprocessed child. Thus we have a cycle of unprocessed nodes. It follows from property (1) of Lemma 3.3 that the trace is invalid. In the remainder of this proof, we argue that if we are unable to schedule the next operation in step 8(b), then the trace is invalid.

Since S is empty in step 8(b), we have that all enqueue events have at least one incoming edge (since none are eligible). If Q is empty, then all dequeues have an (implicit) incoming edge from their respective enqueues. Thus we have a cycle of dependencies, and the trace is invalid.

If Q is not empty, then Algorithm $\mathcal{C}$ returns NO because the dequeue h' in step 8(b) is not eligible. Interchain edges are only between enqueue events, so h' must have an incoming chain edge. Let j be the chain containing h'. There are two cases to consider:

1. Suppose there is a dequeue event earlier than h' in chain j that is still in G (i.e., it has not yet been added to σ). Let a' be the latest such dequeue event. Then as a result of step 4 of the algorithm (recall Figure 6), either there is an edge added to G from a to h, or a appears before h on the same chain. By Lemma 3.3, a path from a to h still exists in G and any queue sort must have a before h. Since $v(h)$ is at the head of the queue, h has been added to σ, and hence so has a. Moreover, a is before h, so $v(a)$ is in Q before $v(h)$, contradicting $v(h)$ at the head of the queue.
2. Suppose that only enqueue events are both earlier than h' in chain j and still in G. Let a be the enqueue event that is the current head of chain j. If the current heads of all chains are ineligible events, we have a cycle and the trace is invalid. Because no enqueue events are eligible, there must be a path in the current G from an eligible dequeue event, c', to a. Moreover, by Lemma 3.4, this path is comprised of chain edges from c' to the earliest enqueue event, b, after c' on the same chain, followed by an edge from b to a. See Figure 8. When the edge from b to a was processed in step 5(c), the edge from c to h was added (since h' is the earliest dequeue after a). By Lemma 3.3, a path from c to h still exists in G and any queue sort must have c before h. Since $v(h)$ is at the head of the queue, h has been added to σ, and hence so has c. Moreover, c is before h, so $v(c)$ is in Q before $v(h)$, contradicting $v(h)$ at the head of the queue.

Thus in all cases, if we are unable to schedule the next operation in step 8(b), then the trace is invalid.

Therefore, Algorithm $\mathcal{C}$ returns a queue sort if and only if one exists. □

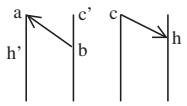

Fig. 8. Case 2 in the proof of Theorem 3.5.

In the course of Algorithm $\mathcal{C}$, we construct a partial order on the enqueue events, to be used in step 8 of the algorithm, using $O(np)$ edges in all. Note that there are partial orders that can be constructed starting with p chains such that, for any p and any $n \geq p^2$, the number of edges is $\Theta(pn)$, and removing any edge alters the transitive closure. For example, consider p chains of n/p events and the partial order in which for all i, j, and k, $1 \leq i \leq p$, $1 \leq j \leq n/p$, $0 \leq k \leq \min(p - i, n/p - j - 1)$, there is an edge from the jth event in the ith chain to the $(j + k + 1)$th event in the $(i + k)$th chain. Since each edge increases the event number by exactly one more than it increases the chain number, no single edge can start and end at the same events as a path of more than one edge, so removing any of the $\Theta(pn)$ edges alters the transitive closure.

3.4. *Testing Queues and Priority Queues*: *Arbitrary Partial Orders*

For comparison, we consider testing queues with an arbitrary partial order. We have the following $O(n^3)$ time algorithm that simply repeatedly adds elements to the relation $<_\sigma$ that are implied by the queue sort property.

Algorithm $\mathcal{D}$

Given: A distinct-values trace $(U, <, \pi, op)$ for a (possibly buggy) FIFO queue.
Question: Does there exist a queue sort σ of the trace?

1. Match up event pairs. If there exists an event that is not in an event pair, return NO.
2. Let $<_\sigma = <$.
3. For each event pair (a, a'), add $a <_\sigma a'$.
4. Let (a, a') and (b, b') be a pair of event pairs. If $a <_\sigma b$, then add $a' <_\sigma b'$. If $a' <_\sigma b'$, then add $a <_\sigma b$.
5. Add all transitive pairs to $<_\sigma$.
6. Repeat steps 4 and 5 until no new pair can be added to $<_\sigma$.
7. If the resultant relation $<_\sigma$ has a cycle, return NO.
8. Return a topological sort σ using the following greedy schedule. At each step of the schedule, an event α in U is *eligible* if it is unscheduled and all events β in U such that $\beta <_\sigma \alpha$ have been scheduled. Starting with an empty schedule and an empty queue, repeat until all events are scheduled: Schedule an eligible enqueue (if any), adding its value to the tail of the queue. If there are no eligible enqueue events, schedule the eligible dequeue for the value that is presently at the head of the queue.

Theorem 3.6. *Let $(U, <, \pi, op)$ be a distinct-values trace with n events. Algorithm $\mathcal{D}$ runs in $O(n^3)$ time and returns a queue sort of $(U, <, \pi, op)$ if and only if one exists.*

Proof. All constraints added in steps 3–5 must be satisfied by all queue sorts of $(U, <, \pi, op)$. Thus if the resultant relation has a cycle, then there is no queue sort.

Assume that we have partially constructed the topological sort and there are no eligible enqueue events and all eligible dequeue events violate the queue sort property. Since $<_\sigma$ is acyclic, there must be an eligible dequeue event, e'. Since $e <_\sigma e'$ by step 3,

where (e, e') is an event pair, and e' is eligible, e has been scheduled. Thus the queue is not empty. Let $v(a)$ be the value at the head of the queue, and let (a, a') be an event pair. Since no enqueue events are eligible, there is an eligible dequeue event, b', such that $b' <_\sigma a'$, and $v(b)$ is in the queue. By step 4, we have that $b <_\sigma a$, where (b, b') is an event pair. Thus $v(b)$ precedes $v(a)$ in the FIFO queue, a contradiction. Therefore, step 8 succeeds in constructing a queue sort.

Algorithm $\mathcal{D}$ can be implemented in $O(n^3)$ time using known techniques for maintaining transitive closure under edge insertions [It], [LPvL]. □

Priority Queues. Algorithm $\mathcal{D}$ can be readily adapted to priority queues: We add elements to the relation $<_\sigma$ (and take the transitive closure) that are implied by the serial semantics of the priority queue. Namely, let (a, a') and (b, b') be a pair of event pairs. If $v(a) > v(b)$ and $b' <_\sigma a'$, then add the pair $b' <_\sigma a$. The proof of correctness and time bounds of this algorithm follow along the lines of Theorem 3.6.

3.5. *Testing Stacks*

Let $(U, <, \pi, op)$ be a distinct-values trace where the parallel data structure is a stack. A *stack sort* of a distinct-values trace $(U, <, \pi, op)$ is a topological sort σ of $(U, <, \pi, op)$ such that for all event pairs (a, a'), $a <_\sigma a'$, and for all other event pairs (b, b'), if $a' <_\sigma b'$, then either $a' <_\sigma b$ or $b <_\sigma a$. We consider arbitrary partial orders $<$.

The techniques used for testing queues do not extend to testing stacks. One can construct a trace which has no stack sort and yet, when all of the implications derived from the serial stack semantics are added to $<$, the resulting relation is acyclic. Intuitively, it appears that the case of stacks is difficult due to the fact that the implication of the serial stack semantics given $a' <_\sigma b'$ is either $a' <_\sigma b$ or $b <_\sigma a$. One might suspect that there is a combinatorial blowup in trying to decide which of the implications holds. However, we have found a polynomial time algorithm for producing a stack sort if one exists.

The idea of the algorithm is to find an event pair (a, a') for which we can make a the first event in the stack sort and solve two subproblems, the first on events that must appear between a and a' in the topological sort and the second on events that can appear after a'. This is a natural divide-and-conquer strategy, because the first event pushes $v(a)$ onto an empty stack and then a value can be pushed onto the stack prior to a' in the stack sort if and only if it is also popped prior to a'. The pop event a' empties the stack, and hence a value can be subsequently pushed onto the stack if and only if it is also popped. For example,

$$a, b, b', c, d, d', c', a', e, f, f', e'$$

is a valid stack sort. Thus, having selected (a, a') as the "dividing pair," we can recurse on the two subproblems.

Algorithm $\mathcal{E}$ below is a recursive algorithm. Prior to invoking it on the input trace, we match up event pairs and return NO if there exists an event not in an event pair. Also, for each event pair (a, a'), add $a < a'$ to $<$. Add all transitive implications to $<$. Return NO if there is a cycle in $<$.

The original partial order $<$ on the events $\{a, a', b, b', c, c', d, d', e, e', f, f'\}$:

$$a < b', c < a' < f', b < d' < c', e < f'$$

The preprocessing phase adds the following to $<$:

$$a < a', b < b', c < c', d < d', e < e', f < f', a < f', c < f', b < c', d < c'$$

The graph G:

The sets for (d, d'), not a dividing pair:

$$L_{(d,d')} = \{(b, b')\}, \quad L^*_{(d,d')} = \{(a, a'), (b, b'), (c, c')\}$$
$$R_{(d,d')} = \{(c, c')\}, \quad R^*_{(d,d')} = \{(a, a'), (b, b'), (c, c')(f, f')\}$$

The sets for (a, a'), a dividing pair:

$$L_{(a,a')} = \{(c, c')\}, \quad L^*_{(a,a')} = \{(b, b'), (c, c'), (d, d')\}$$
$$R_{(a,a')} = \{(f, f')\}, \quad R^*_{(a,a')} = \{(f, f')\}$$
$$\overline{L^*_{(a,a')}} \setminus \{(a, a')\} = \{(e, e'), (f, f')\}$$

A valid stack sort (after recursing):

$$a, b, b', c, d, d', c', a', e, f, f', e'$$

Fig. 9. A stack trace example.

The reader is referred to Figure 9 for a running example. Let G be a directed graph whose nodes are the set of event pairs, with an edge $(b, b') \rightarrow (c, c')$ whenever $b < c$, $b < c'$, $b' < c$, or $b' < c'$.

For each (potential dividing) pair (a, a'), we are interested in computing event pairs that must appear between a and a' in σ, and event pairs that must appear after a' in σ. The pair (a, a') can be selected as a dividing pair if and only if (i) there is no intersection between these two sets, and (ii) no event must precede a in σ. Accordingly, define $L_{(a,a')} = \{(b, b')$ is an event pair such that $b \neq a$ and $b < a'\}$. Such pairs must precede a'. (Note that we need not explicitly check for $b' < a'$ because by the preprocessing phase, if $b' < a'$ then we have $b < a'$. Also, we do not explicitly check for $b < a$ or $b' < a$ because these are checked later when we test for condition (ii).) Define the *left-set* of (a, a') as $L^*_{(a,a')} = \{(c, c')$ is an event pair such that $c \neq a$ and there is a path in G of length zero or more from (c, c') to some $(b, b') \in L_{(a,a')}\}$. Such pairs must all appear to the left of (i.e., precede) a' in σ. Similarly, for an event pair (a, a'), define $R_{(a,a')} = \{(b, b')$ is an event pair $\mid a' < b'\}$. (Again, we need not explicitly check for $a' < b$.) Define the *right-set* of (a, a') as $R^*_{(a,a')} = \{(c, c')$ is an event pair such that $c \neq a$ and there is a path in G of length zero or more from some $(b, b') \in R_{(a,a')}$ to $(c, c')\}$. Such pairs must all appear to the right of a' in σ.

Algorithm $\mathcal{E}$

Given: A distinct-values trace for a (possibly buggy) stack.
Question: Does there exist a stack sort σ of the trace?

1. If there are no event pairs, return the empty sequence.
2. Find an event pair (a, a') satisfying the following: (i) there is no event pair (b, b') with $b < a$, and (ii) $L^*_{(a,a')} \cap R^*_{(a,a')} = \emptyset$.
3. If there is no event pair satisfying 2(i) and 2(ii), output NO and halt.
4. Otherwise, recursively determine a stack sort sequence σ_L for the distinct-values trace restricted to the events in $L^*_{(a,a')}$ and a stack sort sequence σ_R for the distinct-values trace restricted to the events in $\overline{L^*_{(a,a')}} \backslash \{(a, a')\}$.
5. If both recursive calls are successful, return $\sigma = a,\ \sigma_L,\ a',\ \sigma_R$.

Theorem 3.7. *Algorithm $\mathcal{E}$ returns a stack sort of $(U, <, \pi, op)$ if and only if one exists.*

Proof. Due to the preprocessing phase of algorithm $\mathcal{E}$, if the algorithm returns a σ, then each event in the trace is in exactly one matched pair. Moreover, each recursive call by the algorithm is on a trace in which each event is in exactly one matched pair.

We first show that if the algorithm returns a sequence σ, then the trace is valid. Specifically, we must show that σ satisfies the following four properties:

1. Each event is in exactly one event pair.
2. For each event pair (b, b'), $b <_\sigma b'$.
3. For each pair of event pairs (b, b') and (c, c'), if $b' <_\sigma c'$, then $b' <_\sigma c$ or $c <_\sigma b$.
4. For each pair of event pairs (b, b') and (c, c'), if $b < c$, then $b <_\sigma c$, if $b' < c$, then $b' <_\sigma c$, if $b < c'$, then $b <_\sigma c'$, and if $b' < c'$, then $b' <_\sigma c'$.

The proof is by induction, with a trivial basis and the following inductive claim:

Claim 3.8. *At any level of the recursion, if σ_L and σ_R each satisfy the above four properties, then so does $\sigma = a,\ \sigma_L,\ a',\ \sigma_R$.*

To prove this claim, we consider each property in turn. Property 1 holds because the matched pair (a, a') is added to the set of matched pairs in σ_L and σ_R. Property 2 holds because $a <_\sigma a'$ and every other matched pair is either in σ_L or in σ_R. As for property 3, consider a pair of event pairs (b, b') and (c, c') such that $b' <_\sigma c'$ and these matched pairs are neither both in σ_L nor both in σ_R. If $c = a$, we have $c <_\sigma b$, and the property holds. Otherwise, we have $c, c' \in \sigma_R$ and $b, b' \notin \sigma_R$. Thus $b' <_\sigma c$, and the property holds.

To show that property 4 holds, consider two event pairs (b, b') and (c, c'). There are three cases:

1. (b, b') is the dividing pair (a, a'). In this case we note that $a <_\sigma c$ and $a <_\sigma c'$. If $a' < c'$, then $(c, c') \in R^*_{(a,a')}$, and by condition (ii) of step 2, $(c, c') \notin L^*_{(a,a')}$. Hence, $c' \in \sigma_R$ and thus $a' <_\sigma c'$. If $a' < c$, then $a' < c'$ after the preprocessing phase, and it follows that $c \in \sigma_R$, and hence $a' <_\sigma c$.

2. (c, c') is the dividing pair (a, a'). In this case we note that condition (i) of step 2 (applied after the preprocessing phase) ensures that neither $b < a$ nor $b' < a$. If $b < a'$, then $(b, b') \in L_{(a,a')}$, and it follows that $b <_\sigma a'$. If $b' < a'$, then $b < a'$ after the preprocessing phase, and hence $(b, b') \in L_{(a,a')}$, so $b' <_\sigma a'$.
3. Neither (b, b') nor (c, c') are the dividing pair (a, a'). If $b < c$, $b' < c$, $b < c'$, or $b' < c'$, we have an edge $(b, b') \to (c, c')$ in G. If $(c, c') \in \sigma_L$ then either $(c, c') \in L_{(a,a')}$ or there is a path in G from (c, c') to some event pair in $L_{(a,a')}$. In either case, $(b, b') \in L^*_{(a,a')}$ and hence $(b, b') \in \sigma_L$, and the property holds inductively. If $(c, c') \in \sigma_R$, then either $(b, b') \in \sigma_L$ or $(b, b') \in \sigma_R$. In either case the property holds.

This completes the proof of the claim, and hence it follows by induction that if the algorithm returns a sequence σ, then σ is a stack sort of the input trace, and hence the trace is valid.

We now show the converse, that is, if the trace is valid (i.e., it has a stack sort), the algorithm returns a sequence σ. The proof proceeds by a series of observations:

1. If a trace has a stack sort, then each event is in exactly one event pair and there will be no cycles in the augmented $<$ after the preprocessing phase.
2. If a trace has a stack sort, then any subset of the event pairs in the trace also has a stack sort.
3. Consider a stack sort $\sigma = a,\ \sigma_1,\ a',\ \sigma_2$, where (a, a') is an event pair and σ_1 and σ_2 are sequences of events. Then for all event pairs (b, b'), $b \in \sigma_1$ if and only if $b' \in \sigma_1$ (and hence $b \in \sigma_2$ if and only if $b' \in \sigma_2$).

   This follows from the stack sort property (recalled at the beginning of this section, and repeated in property 3 above for the specific case of algorithm $\mathcal{E}$. As with the observations that follow, observation 3 holds for all stack sorts, not just those produced by the algorithm.
4. In any stack sort $\sigma = a,\ \sigma_1,\ a',\ \sigma_2$, where (a, a') is an event pair, if $(b, b') \in L^*_{(a,a')}$, then $b \in \sigma_1$ and $b' \in \sigma_1$.

   To see this, note that if $(b, b') \in L_{(a,a')}$, then because $b < a'$ and by observation 3, both b and b' are in σ_1. Moreover, if $c \in \sigma_1$, $c' \in \sigma_1$, and $(b, b') \to (c, c')$, then at least one of b or b' is in σ_1, and hence by observation 3, both are. The observation follows by an easy inductive argument on the length of the path in G.
5. In any stack sort $\sigma = a,\ \sigma_1,\ a',\ \sigma_2$, where (a, a') is an event pair, if $(b, b') \in R^*_{(a,a')}$, then $b \in \sigma_2$ and $b' \in \sigma_2$.

   To see this, note that if $(b, b') \in R_{(a,a')}$, then because $a' < b'$ and by observation 3, both b and b' are in σ_2. Moreover, if $c \in \sigma_2$, $c' \in \sigma_2$, and $(c, c') \to (b, b')$, then at least one of b or b' is in σ_2, and hence by observation 3, both are. The observation follows by an easy inductive argument on the length of the path in G.
6. If a trace has a stack sort, then there exists an event pair satisfying 2(i) and 2(ii).

   To see this, let (a, a') be an event pair such that a is the first event in a stack sort for the trace. Then clearly (a, a') satisfies 2(i), and by observations 4 and 5, (a, a') also satisfies 2(ii).

7. If a trace has a stack sort, then for any event pair (a, a') satisfying 2(i) and 2(ii), the trace comprising events in $L^*_{(a,a')}$ has a stack sort and the trace comprising events in $\overline{L^*_{(a,a')}} \backslash \{(a, a')\}$ has a stack sort.
   This follows by observations 1 and 2.

Observations 1 and 6 imply that the algorithm will return a sequence σ if for each recursive call, the trace has a stack sort. Thus by observation 7, if the input trace has a stack sort, the algorithm returns a sequence σ.

This completes the proof of the theorem. □

4. Arbitrary Values

In this section we consider testing parallel executions of queues, priority queues, and stacks in which multiple insert and delete events may have the same value, so we do not have an a priori 1–1 correspondence between inserts and deletes. We show that these testing problems are NP-complete, even for interval orders and for chains. Two completely different reductions from 3SAT are used for interval orders and for chains, with only slight variations between a queue, a priority queue, and a stack. We begin with the construction for chains, which is straightforward, before presenting the construction for interval orders, which uses more interesting techniques.

4.1. *Chains*

Theorem 4.1. *Testing a concurrent FIFO queue, priority queue, or stack with arbitrary value types is NP-complete, even if the partial order is the union of chains.*

Proof. The proof is a reduction from 3SAT; the same reduction can be used for queues, priority queues, or stacks.

Given a 3SAT instance, $\mathcal{F}$, with n variables and m clauses, we construct the following trace, $\mathcal{T}$. In the figures below, each column represents one chain in $\mathcal{T}$. First, we have the following $2n+1$ chains. We have a chain consisting of Ins(i) for $i = 1, 2, \ldots, n$, denoted the *variable chain*. For $i = 1, \ldots, n$ we also have two chains headed by Del(i): one with an Ins(c_j) for each clause C_j containing the literal x_i (in increasing order), denoted chain(x_i), and one with an Ins(c_j) for each clause C_j containing the literal $\bar{x}_i$ (also in increasing order), denoted chain($\bar{x}_i$). The figure, for example, depicts a 3SAT instance where the literal x_1 is in clauses $\{c_2, c_3, \ldots\}$, the literal $\bar{x}_1$ is in clauses $\{c_4, c_6, \ldots\}$, the literal x_2 is in clauses $\{c_4, c_5, \ldots\}$, the literal $\bar{x}_2$ is in clauses $\{c_1, c_6, \ldots\}$, and so forth.

Ins(1)	Del(1)	Del(1)	Del(2)	Del(2)	$\cdots$	Del(n)	Del(n)
Ins(2)	Ins(c_2)	Ins(c_4)	Ins(c_4)	Ins(c_1)		Ins(c_3)	Ins(c_2)
$\vdots$	Ins(c_3)	Ins(c_6)	Ins(c_5)	Ins(c_6)		Ins(c_5)	Ins(c_4)
Ins(n)	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$

In addition, we have the following $m + 1$ chains:

$$
\begin{array}{llllll}
\mathrm{Del}(c_1) & \mathrm{Del}(c_1) & \mathrm{Del}(c_2) & \cdots & \mathrm{Del}(c_m) \\
\mathrm{Del}(c_2) & \mathrm{Del}(c_1) & \mathrm{Del}(c_2) & & \mathrm{Del}(c_m) \\
\quad\vdots & & & & \\
\mathrm{Del}(c_m) & & & & \\
\mathrm{Ins}(1) & & & & \\
\mathrm{Ins}(2) & & & & \\
\quad\vdots & & & & \\
\mathrm{Ins}(n) & & & &
\end{array}
$$

We denote the leftmost chain above as the *clause chain*.

Intuitively, the insertions and deletions in this construction force a commitment to either chain(x_i) or chain($\bar{x}_i$) for all i that must satisfy all the clauses. If and only if each clause is indeed satisfied, then all the deletions in the clause chain can be scheduled, and the entire trace has a valid schedule.

We first show that if $\mathcal{F}$ is a positive instance, then there exists a topological sort of $\mathcal{T}$ that is a sequence of pairs of events, where each pair is an insert operation followed by a delete operation with the same value, and hence is a legal schedule for queues, priority queues, or stacks. Consider a satisfying assignment, **T**, for $\mathcal{F}$. The schedule is as follows:

1. For $i = 1, \ldots, n$: first schedule Ins(i) from the variable chain, then if **T** sets x_i to true, schedule the Del(i) that heads chain(x_i) else schedule the Del(i) that heads chain($\bar{x}_i$).
2. At this point, since **T** is a satisfying assignment, we have at least one Ins(c_1) ready for scheduling. For each ready Ins(c_1), schedule it followed by a Del(c_1) (which will be ready), with the first such Del(c_1) being the one that heads the clause chain. Repeat for $c_2, \ldots, c_m$.
3. Finally, schedule the remaining operations by repeating the above steps, with the roles of x_i and $\bar{x}_i$ reversed.

The reader may verify that this is a valid schedule.

Conversely, we now show that for each of queues, priority queues, and stacks, if there exists a legal topological sort of $\mathcal{T}$, then $\mathcal{F}$ is a positive instance. The reduction is the same for each type of data structure, since the only property used is that a value must be inserted before it is deleted.

Let S be a legal topological sort of $\mathcal{T}$. Consider the following truth assignment, **T**, for $\mathcal{F}$. For $i = 1, \ldots, n$, if the Del(i) that heads chain(x_i) precedes the Del(i) that heads chain($\bar{x}_i$), then we denote chain(x_i) as the *satisfied* chain and **T** sets variable x_i to true, otherwise chain($\bar{x}_i$) is the *satisfied* chain and **T** sets variable x_i to false.

Let S' be the prefix of S up to and including the Del(c_m) operation in the clause chain. Note that S' contains at most one Ins(i) operation, for $i = 1, \ldots, n$. Thus S' contains at most one Del(i) operation, for $i = 1, \ldots, n$, and hence for each x_i, only its satisfied chain can have operations in S'.

We claim that **T** is a satisfying assignment for $\mathcal{F}$. Suppose there were an unsatisfied clause C_j. Since the Del(c_j) operation from the clause chain is in S', at least one Ins(c_j) must also be in S'. However, by the construction and the definition of **T**, there are no Ins(c_j) operations in satisfied chains, and hence no such operations in S', a contradiction. □

4.2. *Interval Orders*

Theorem 4.2. *Testing a concurrent FIFO queue or priority queue with arbitrary value types is NP-complete, even if the partial order is an interval order.*

Proof. The proof is a reduction from 3SAT; the same reduction can be used for FIFO queues or priority queues. Given a 3SAT instance, $\mathcal{F}$, with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$, we construct a trace, $\mathcal{T}$, as follows.

To simplify the description, we have multiple intervals in $\mathcal{T}$ with common start times or end times; these ties can be broken arbitrarily to ensure unique interval endpoints.

There are $n + m + 1$ values that are enqueued or dequeued, one for each variable v_i, one for each clause c_j, and a special value α. For notational simplicity, we refer to these values by their symbolic names v_i, c_j, and α; there is an implicit 1–1 mapping of symbolic names to the integers $0, 1, \ldots, n + m$. The exact mapping is irrelevant for testing queues. For testing priority queues (with deletemax), the mapping satisfies $c_1 > \cdots > c_m > v_1 > \cdots > v_n > \alpha$ (i.e., $\alpha = 0$, $v_i = n-i+1$, and $c_j = n+m-j+1$). The proof focuses on testing (FIFO) queues, and later shows that the same construction holds for testing priority queues.

Recall from Section 2 that a pair of events (e, e') is a *corresponding pair* in a queue sort if e' is the dequeue that removed the instance of the value enqueued by e. Also recall that in a queue sort, corresponding pairs cannot nest, e.g., if (e, e') and (f, f') are each corresponding pairs, then f, e, e', f' is not a valid ordering of these operations in a queue sort.

In the construction, there are two operations involving α, an enqueue and a dequeue, with short matching intervals. As the only operations with value α, these are necessarily a corresponding pair in any queue sort. This pair ensures that there is no other corresponding pair (e, e') such that e precedes the enqueue of α and e' succeeds the dequeue of α in a queue sort. We say that such a pair *crosses* the *cut-off line* defined by α.

The construction uses two gadgets, depicted in Figure 10. In this figure and the ones that follow, each operation is represented by a labeled vertical box. The label is either Iv for an operation that enqueues (inserts) the value v, or Dv for an operation that dequeues (deletes) the value v. The vertical extent of the box depicts its interval of time; time progresses from top to bottom in the figure.

For each variable v_i, we have the six interval gadget on the left, consisting of three operations that enqueue the value v_i (labeled Iv_i) and three that dequeue v_i (labeled Dv_i). Also shown are the enqueue of α (labeled $I\alpha$), the dequeue of α (labeled $D\alpha$), and the cut-off line defined by α (the dashed line labeled α). Figure 11 depicts two possible schedules for this gadget (the numbers from 1 to 8). It is easily verified that any schedule for these intervals must either pair the two short intervals below the cut-off line (as depicted on the left) or the two short intervals above the cut-off line (as depicted

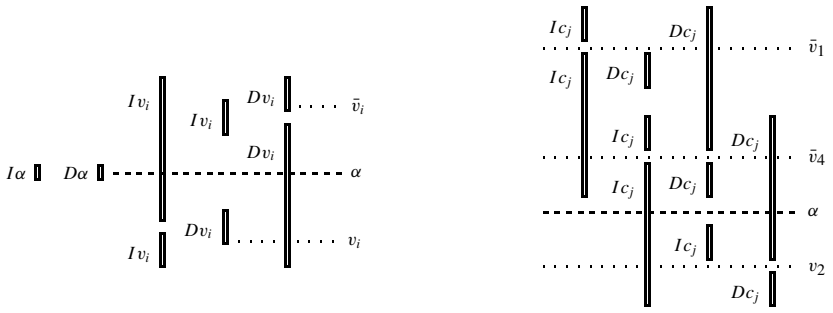

Fig. 10. On the left, the *variable* gadget for v_i. On the right, the *clause* gadget for $C_j = v_1 \vee \bar{v}_2 \vee v_4$.

on the right). (The schedule may pair both, but it turns out this creates less flexibility in scheduling the remaining intervals in the construction, so we may assume without loss of generality that any schedule pairs exactly one.) Pairing the intervals below the line corresponds to setting v_i to true, and creates the cut-off line labeled v_i on the left. No corresponding pair may cross this line if v_i is true. Pairing the intervals above the line corresponds to setting v_i to false, and creates the cut-off line labeled $\bar{v}_i$ on the right. No corresponding pair may cross this line if v_i is false. These cut-off lines are called the *literal lines* for v_i and $\bar{v}_i$. Thus for each variable we have two *potential* literal lines (shown with dotted lines in Figure 10), one of which becomes a *realized* literal line depending on the schedule.

For each clause C_j, we have the ten-interval gadget on the right in Figure 10. The gadget consists of five operations that enqueue the value c_j (labeled Ic_j) and five that dequeue c_j (labeled Dc_j). The gaps between the intervals are positioned around the three potential literal lines for the *complements* of the literals in the clause. In the figure, $C_j = v_1 \vee \bar{v}_2 \vee v_4$, so the gaps are positioned around the potential literal lines for $\bar{v}_1$, v_2, and $\bar{v}_4$. The rationale behind this gadget can be seen from the following claim.

Claim 4.3. *A clause gadget can be scheduled if and only if at least one of its three potential literal lines is not realized.*

To see this, consider the clause gadget in Figure 10 and suppose all three potential literal lines are realized. Then the short Ic_j above the top literal line (in the figure, the $\bar{v}_1$ line) can only be paired with the Dc_j that begins above the line, thus the Ic_j below it is the

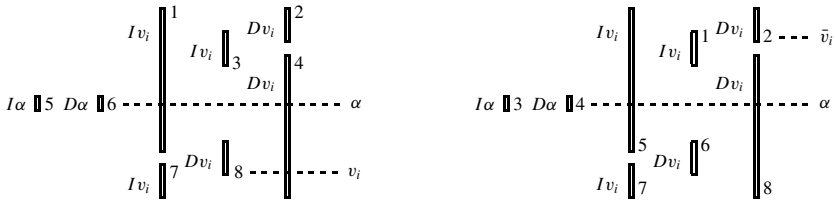

Fig. 11. On the left, v_i is set to true. On the right, v_i is set to false.

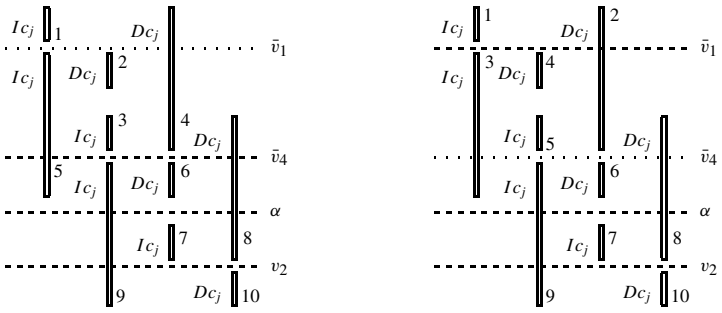

Fig. 12. The clause gadget can be scheduled if and only if a legal schedule can pair an insert with a delete across one of the *literal* lines. On the left, a legal schedule when the $\bar{v}_1$ line can be crossed. On the right, a legal schedule when the $\bar{v}_4$ line can be crossed.

only operation remaining that can be paired with the short Dc_j next to it, the short Ic_j between the top and middle literal lines (just above the $\bar{v}_4$ line) can only be paired with the Dc_j that crosses the middle line, the Ic_j that crosses the bottom literal line (v_2) is the only operation remaining that can be paired with the short Dc_j just below the middle line, which leaves the short Ic_j above the bottom line to be paired with the short Dc_j on the other side of the bottom line. This last pairing is not permitted, so there is no schedule. Conversely, if any one of the literal lines are not realized, there is a schedule. For example, the schedule just described would be a queue sort if the bottom line is not realized. Schedules for the other two cases are depicted in Figure 12 (the numbers from 1 to 10).

Thus the clause can be scheduled if and only if at least one of the *complement* literal lines is *not* realized, i.e., if and only if at least one of the literals is set and hence the clause is satisfied.

An example of the full construction is shown in Figure 13. Note that in each of the variable schedules above, corresponding pairs are each scheduled arbitrarily close to the same time. Likewise, in each of the clause schedules above, all corresponding pairs but the one that crosses the unrealized literal line are each scheduled arbitrarily close to the same time. Thus for all these pairs, we schedule the dequeue immediately after the enqueue (and the queue has at most one value). As for the pairs that cross an unrealized literal line, we schedule the set of pairs crossing a given line such that the enqueues are scheduled first followed by the corresponding dequeues in the same order, to preserve the FIFO property. It can be readily verified that scheduling in this way according to a satisfying truth assignment for $\mathcal{F}$ results in a queue sort for the constructed trace $\mathcal{T}$. Conversely, if there is no satisfying truth assignment for $\mathcal{F}$, then any scheduling of the variable gadgets will result in some clause gadget that cannot be scheduled.

We now argue that the same construction suffices for priority queues, given the mapping of symbolic values to actual values discussed above, i.e., given that $c_1 > \cdots > c_m > v_1 > \cdots > v_n > \alpha$.

Recall from Section 2 that in a priority-queue (with deletemax) sort, corresponding pairs cannot nest with the smaller value in the middle, e.g., if (e, e') and (f, f') are each corresponding pairs with the former pair having a smaller value than the latter pair,

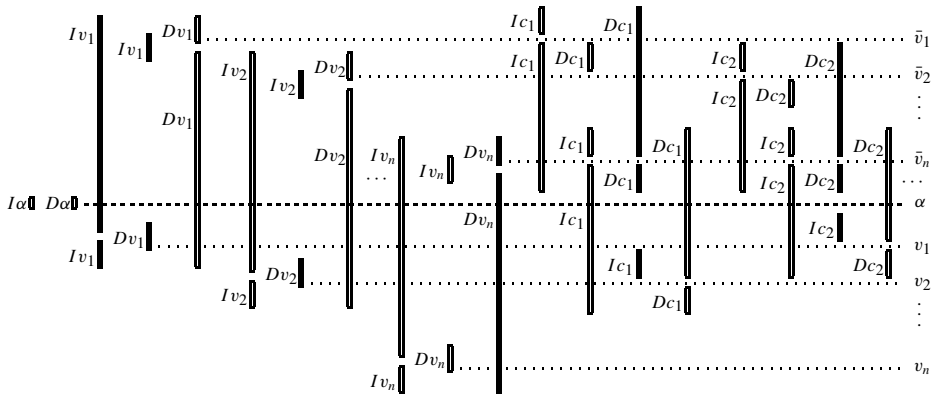

Fig. 13. Transforming an instance of 3SAT to an instance of the testing parallel executions of queues (and priority queues) problem for interval orders. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. The full construction has $6n + 10m + 2$ operations. Here, $C_1 = v_1 \vee \bar{v}_2 \vee v_n$ and $C_2 = \bar{v}_1 \vee v_2 \vee v_n$.

then f, e, e', f' is not a valid ordering of these operations in a priority-queue sort. Thus, it still holds that no other corresponding pair can cross the cut-off line defined by the smallest value α, and hence one literal line must be realized for each variable. Moreover, no clause corresponding pair can cross a realized literal line because the v_i values are smaller than the c_j values. Thus, an individual clause can be scheduled if and only if the setting of the v_i satisfies the clause.

When scheduling pairs that cross an unrealized literal line, we schedule the set of pairs crossing a given line such that the enqueues are scheduled first followed by the corresponding dequeues in order of decreasing value, to preserve the deletemax property. It can be readily verified that scheduling in this way according to a satisfying truth assignment for $\mathcal{F}$ results in a priority-queue sort for the constructed trace $\mathcal{T}$. Conversely, if there is no satisfying truth assignment for $\mathcal{F}$, then any scheduling of the variable gadgets will result in some clause gadget that cannot be scheduled.

This completes the proof of Theorem 4.2. □

Theorem 4.4. *Testing a concurrent stack with arbitrary value types is NP-complete, even if the partial order is an interval order.*

Proof. We use a similar construction for stacks as for queues. The primary difference is that a corresponding pair scheduled one after another does not produce a cut-off line (the push is immediately popped, regardless of what has happened before or after), so we need a different gadget to create cut-off lines. Figure 14 depicts the new variable gadget for stacks.

This gadget produces a slightly weaker type of literal line, which nevertheless suffices for our clause gadget (the clause gadget is the same as for queues). Consider the schedule on the left of Figure 14 and its literal line for v_i. This line cannot be crossed by a corresponding pair (push(c_j), pop(c_j)) such that the pop precedes the Dv_i labeled 6, because then we would have push(c_j), push(v_i), pop(c_j), pop(v_i), which is not a valid

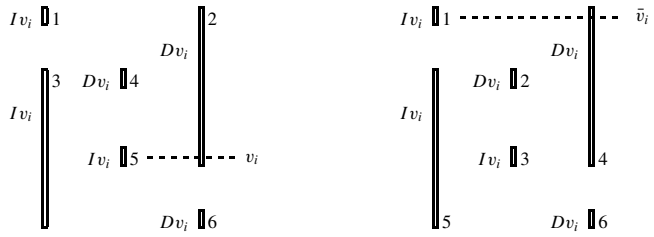

Fig. 14. The variable gadget for v_i for testing stacks. On the left, v_i is set to true. On the right, v_i is set to false.

stack sort. In particular, if the interval for a push(c_j) ends before the interval for the push(v_i) begins, the interval for the push(v_i) ends before the interval for the pop(c_j) begins, and the interval for the pop(c_j) ends before the interval for the pop(v_i) begins (i.e., push(c_j) $<$ push(v_i) $<$ pop(c_j) $<$ pop(v_i)), then the push(c_j) cannot be paired with the pop(c_j) in a stack sort if v_i is set to true.

An example of the full construction is shown in Figure 15. Note that the appropriate intervals surrounding a literal line have the property discussed in the previous paragraph, so that a realized literal line does indeed prevent crossing by corresponding pairs. Thus, an individual clause can be scheduled if and only if the setting of the v_i satisfies the clause.

When scheduling pairs that cross an unrealized literal line for a variable v_i, we schedule the set of pairs crossing a given line such that the pushes are scheduled first, followed by the push and pop of v_i, followed by the corresponding pops in reverse order

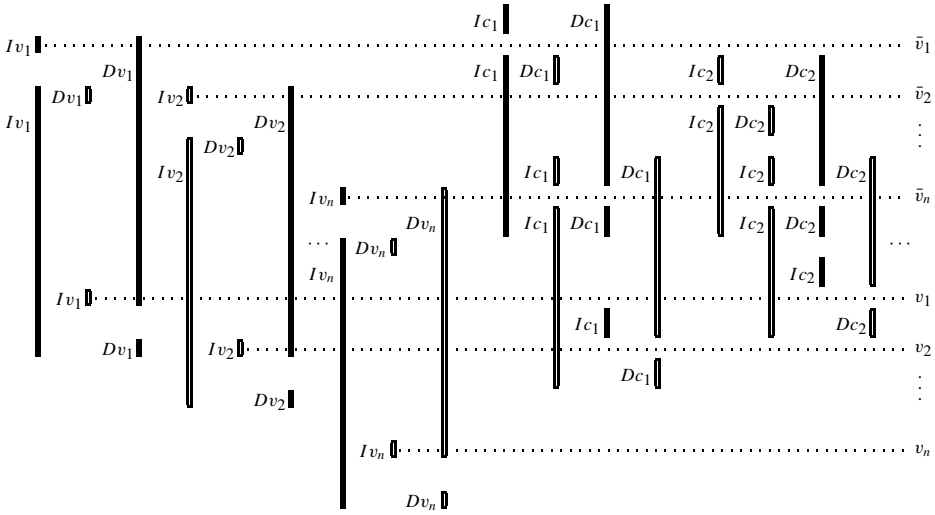

Fig. 15. Transforming an instance of 3SAT to an instance of the testing parallel executions of stacks problem for interval orders. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. The full construction has $6n + 10m$ operations. Here, $C_1 = v_1 \vee \bar{v}_2 \vee v_n$ and $C_2 = \bar{v}_1 \vee v_2 \vee v_n$.

of the pushes, to preserve the stack property. It can be readily verified that scheduling in this way according to a satisfying truth assignment for a 3SAT instance $\mathcal{F}$ results in a stack sort for the constructed trace $\mathcal{T}$. Conversely, if there is no satisfying truth assignment for $\mathcal{F}$, then any scheduling of the variable gadgets will result in some clause gadget that cannot be scheduled. □

5. Counters

This section addresses testing parallel executions of shared counters under various scenarios. A *shared counter* is defined to be an integer variable that can be increased or decreased subject to the constraint that the value should never drop below zero. This definition encompasses such objects as semaphores, as well as data structures such as stacks or queues when we care only about the *number* of items stored (and we allow a single event to insert or delete multiple items). A *shared-counter trace* is a trace $(U, <, \pi, op)$ such that $op\colon U \to \mathcal{Q}$, the set of rational numbers. A shared-counter trace is *valid* if there is a topological sort $\sigma = \alpha_1, \ldots, \alpha_n$ of $(U, <, \pi, op)$ such that for $i = 1, \ldots, n$ we have $\sum_{j=1}^{i} op(\alpha_j) \geq 0$.

In this section we first describe a straightforward linear time algorithm for testing parallel executions of linearizable counters. Then we discuss how an $O(n \log n)$ time algorithm for testing parallel executions of counters, where the partial order is either the union of chains or a series-parallel order, can be obtained as a corollary of a known result on sequencing to minimize maximum cumulative profit. On the other hand, we observe that it is an immediate consequence of previous work that testing parallel executions of counters with an arbitrary partial order is NP-complete. Moreover, this hardness result can be extended to distinct-values traces. We also show that for two natural generalizations of shared counters, "fetch&add counters" and "counters with reads," the testing parallel executions problem is NP-complete even for interval orders or a collection of chains.

5.1. *Algorithms for Counters*

Sequencing to maximize minimum cumulative profit is the following problem: Given a shared-counter trace $(U, <, \pi, op)$, find a topological sort σ, such that the minimum sum of op over prefixes of σ is maximum over all topological sorts of $(U, <, \pi, op)$. An algorithm for this problem is clearly sufficient for testing parallel executions of shared counters, where the non-negativity constraint only requires a topological sort with no negative prefix sum.

This problem has another pleasing interpretation, namely, minimizing memory usage in a serial computation. Consider a computation composed of a set of steps, with a partial order encoding timing constraints between the steps. A prototypical example is expression tree evaluation. We can label each step with its effect on space utilization, namely, we assign to a step a positive number if the step requires additional memory be allocated to the computation and we assign it a negative number if the step results in memory deallocation. Finding a schedule of the computation that minimizes the total memory used is just sequencing to minimize maximum cumulative cost, which is equivalent (by changing the sign of the assignments) to maximizing minimum cumulative

profit. Note that this is an offline optimization, so it cannot be used if either the computation or the partial order is revealed online. The case of expression tree evaluation was considered by Biswas and Kannan [BK2]. Note that by Theorem 4.7 of [BGM], an offline serial schedule that minimizes memory usage can always be the basis for an offline parallel schedule with good time and space bounds.

First consider the case in which the partial order of the shared-counter trace $(U, <, \pi, op)$ is an interval order.

Theorem 5.1. *There is an $O(n)$ time algorithm for determining a maximum minimum cumulative profit sort σ of a shared-counter trace $(U, <, \pi, op)$ with n events where $<$ is an interval order.*

Proof. Consider the following algorithm (Algorithm $\mathcal{F}$): Sort the timestamps, and then construct the topological sort by scanning the timestamps in increasing order. When we encounter $[_e$ for an event e that increases the counter, then we append the event to the topological sort. When we encounter $]_{e'}$ for an event e' that decreases the counter, we append the event to the topological sort.

To see that Algorithm $\mathcal{F}$ sequences to maximize minimum cumulative profit, let S be a schedule produced by the algorithm. Let S' be a prefix of S with minimum cumulative profit, and let δ be this minimum value. We will show that any other schedule T must have a prefix with cumulative profit at most δ. Specifically, let T' be the shortest prefix of T that includes all the events in S' that decrease the counter. Note that the last event in T' is an event e' in S' that decreases the counter. By construction of S, for all events e in $S - S'$ that increase the counter, we have $]_{e'} < [_e$. Thus none of these events can be scheduled before e', and hence they cannot be in T'. So T' has a superset of the events in S' that decrease the counter and a subset of the events in S' that increase the counter, so T' has cumulative profit at most δ. Therefore, Algorithm $\mathcal{F}$ outputs a schedule that maximizes the minimum cumulative profit. □

Thus testing parallel executions of linearizable counters can be done in linear time.

Next we consider counters where the partial order is a series-parallel order. An efficient algorithm is known for sequencing to maximize minimum cumulative profit in the case of series-parallel constraints.

Theorem 5.2 [AWK], [MS]. *Given a shared-counter trace $(U, <, \pi, op)$ with n events where $<$ is a series-parallel partial order, a topological sort of $<$ with maximum minimum cumulative profit can be found in $O(n \log n)$ time.*

5.2. *NP-Completeness Results for Counters*

If we consider counters with arbitrary partial orders, the NP-completeness result in [AW] for sequencing to maximize minimum cumulative profit implies:

Theorem 5.3 [AW]. *Testing parallel executions of a shared counter (with an arbitrary partial order) is NP-complete.*

Corollary 5.4. *Testing parallel executions of a shared counter with distinct values (with an arbitrary partial order) is NP-complete.*

Proof. We give a polynomial time reduction from the problem P of testing parallel executions of a shared counter with arbitrary values. Given a shared-counter trace $\mathcal{T} = (U, <, \pi, op)$ with n operations, an instance of problem P, we construct a shared-counter trace $\mathcal{T}_d = (U, <, \pi, op^*)$ with distinct values as follows. We arbitrarily number the operations in $\mathcal{T}$ from 1 to n; let $id(\alpha)$ denote the number assigned to an operation α. For each event $\alpha \in U$, we replace op with $op^* = \langle op(\alpha), id(\alpha)\rangle$,[5] so that all values are distinct. The counter is initially $\langle 0, 0\rangle$, and $op^*(\alpha)$ applied to a counter $\langle x, y\rangle$ results in $\langle (x + op(\alpha)), (y + id(\alpha))\rangle$. It is easy to see that $\mathcal{T}$ has a topological sort $\alpha_1, \ldots, \alpha_n$ such that $\forall i \sum_{j=1}^{i} op(\alpha_j) \geq 0$ if and only if $\mathcal{T}_d$ has a topological sort $\alpha_1, \ldots, \alpha_n$ such that $\forall i \sum_{j=1}^{i} op^*(\alpha_j) \geq \langle 0, 0\rangle$. □

Note that this construction does not work with data structures such as queues and stacks for which the delete event must correspond to an insert event with the same value.

5.2.1. *Fetch&Add Counter.* We first consider testing a concurrent counter data structure, X, whose trace contains a single type of operation:

- *fetch&add*(X, v, u), an operation returning the current value v of the counter X and updating the value to $v + u$, for some rational number u.

The value of the counter is assumed to be initially zero. A topological sort *fetch&add* (X, v_1, u_1), *fetch&add*$(X, v_2, u_2), \ldots,$ *fetch&add*(X, v_n, u_n) is legal if $v_1 = 0$ and for $i = 2, 3, \ldots, n$, $v_i = v_{i-1} + u_{i-1}$. This data structure can be viewed as a generalization of a shared counter, in which each event, in updating the value, also returns the value of the counter. In contrast to the shared counter, testing a fetch&add counter is NP-complete not only for arbitrary partial orders, but also for the special cases of chains (and hence series-parallel graphs) and interval graphs.

Theorem 5.5 [GK]. *Testing parallel executions of a fetch&add counter, where the partial order is either an interval order or the union of chains, is NP-complete.*

Proof. We note that within a trace, a *fetch&add*(X, v, u) event is equivalent to a *read&write*$(X, v, v+u)$ event, i.e., an occurrence of an operation that returns the current value v of X and writes the new value $v + u$ to X. This latter operation was studied in [GK] in the context of testing a collection of read/write registers (memory cells) under sequential consistency (denoted the VSC problem) and linearizability (denoted the VL problem). For chains, testing a fetch&add counter is equivalent to a special case of the *VSC problem with read&write only* in which there is but a single memory location; this latter problem was shown to be NP-complete [GK]. Similarly, for interval orders, testing a fetch&add counter is equivalent to a special case of the *VL problem with read&write*

[5] In various proofs throughout this section, we represent values as vectors of multiple components. These can be converted to rational numbers that preserve componentwise addition using standard techniques.

only in which there is but a single memory location; this latter problem was also shown to be NP-complete [GK]. □

5.2.2. *Counter with Reads*. The second generalization considered in this section is a concurrent counter data structure in which each event *either* updates the counter *or* returns the value of the counter. For this *counter with reads* data structure, a topological sort of a trace is legal if for each read operation, $r = read(X, v)$, v is the sum of the values of all update operations (on X) preceding r in the topological sort. We show that testing a counter with reads is NP-complete even for chains (and hence series-parallel graphs) or for interval graphs.

We first consider chains. To show that the counter with reads problem is NP-complete, we give a polynomial time reduction from the fetch&add counter problem. This general reduction replaces each fetch&add event with a read event and an update event. The main difficulty to overcome is that the read event and the update event may end up far apart in the topological sort, so that an illegal fetch&add trace is a legal counter with reads trace. Moreover, whereas a *fetch&add*(v, u) operation can only add u to a counter whose value is v, a separate *update*(u) event can be applied regardless of the counter value; this may permit the counter to obtain certain values not possible with the fetch&add. In replacing one event with two, the reduction potentially alters the properties of the partial order. However, for chains (as well as forests and series-parallel graphs), the given property of the partial order is unchanged.

Lemma 5.6. *There is a polynomial time reduction from testing a concurrent fetch&add counter problem to testing a concurrent counter with reads problem.*

Proof. We first give a high-level description of the proof, before proceeding with the formal proof. To avoid the difficulties mentioned above, we construct a counter with reads trace, $\mathcal{T}_r$, that encodes a value appearing in fetch&add operations into a characteristic vector representation. Although a topological sort σ of $\mathcal{T}_r$ may separate a read event from its corresponding update event, and obtain problematic values, we then show inductively how to shuffle σ to bring corresponding pairs together without violating the partial order. This shuffling is done by finding Euler paths in multigraphs defined by maximal subsequences of consecutive update operations in σ.

The formal proof is as follows. Given a common-values trace, $\mathcal{T}_f$, for a fetch&add counter, we construct the following common-values trace, $\mathcal{T}_r$, for a counter with reads. Associated with each operation, *fetch&add*(v, u), in $\mathcal{T}_f$ are two values, v and $v + u$. Let V_f be the set of all such values in $\mathcal{T}_f$. A value in the constructed trace $\mathcal{T}_r$ will be represented as a vector, χ, with one entry, $\chi[v]$, in the vector for each value v in V_f. We replace each operation, *fetch&add*(v_i, u_i), in $\mathcal{T}_f$ with two operations, *read*(χ) and *update*(U), where $\chi[v]$ is 1 if $v = v_i$ and is 0 otherwise, and $U[v]$ is -1 if $v = v_i$, is 1 if $v = v_i + u_i$, and is 0 otherwise. In the serial semantics for a counter with reads, a *read*(χ) operation is legal if and only if the current counter value is χ. The counter is initially the zero vector and an *update*(U) operation applied to a value χ updates the counter to $\chi + U$, where the plus here indicates elementwise addition. Viewing the partial order in $\mathcal{T}_f$ as a graph, the partial order in $\mathcal{T}_r$ is obtained by splitting each fetch&add node in two, with an edge directed from the corresponding read to the corresponding update.

We claim that $\mathcal{T}_f$ is a positive instance if and only if $\mathcal{T}_r$ is a positive instance. The "only if" direction is straightforward and left to the reader. Conversely, suppose that $\mathcal{T}_r$ is a positive instance, and let σ be a legal topological sort. Let σ_i be the ith maximal subsequence of consecutive update operations in σ. Then $\sigma = r(v_1)^+, \sigma_1, r(v_2)^+, \sigma_2, \ldots, r(v_m)^+, \sigma_m$, where each $r(v_i)^+$ represents one or more *read*(χ) operations with $\chi[v] = 1$ if $v = v_i$ and 0 otherwise. We construct the legal topological sort, σ', for $\mathcal{T}_f$ inductively for $i = 1, \ldots, m$, starting with the empty sequence, as follows. Assume the construction has been completed through step $i - 1$ and consider step i. The relevant subsequence of σ is $r(v_i), \sigma_i, r(v_{i+1})$. Construct a multigraph, G_i, with a node for each $v \in V_f$, and an edge directed from v_j to v_k for each *update*(U) operation in σ_i such that $U[v_j] = -1$ and $U[v_k] = 1$. Since σ_i is legal, its operations take the counter vector from "v_i" to "v_{i+1}", and in order to cancel out all other nonzero entries in the counter vector, the multigraph G_i must have an Euler path, P, from v_i to v_{i+1}. The order in P may differ from the corresponding order in σ_i, however, we claim that the order in P is consistent with the partial order given for $\mathcal{T}_f$. To see this, suppose that there are two updates, x and y, in σ_i such that the corresponding fetch&adds are ordered, x before y, in $\mathcal{T}_f$. Then by construction, x is ordered before the fetch associated with y which is ordered before y in $\mathcal{T}_r$, contradicting x and y both being in σ_i. We append to σ', in the order of P, the corresponding sequence of fetch&add operations. It follows by induction that σ' is a legal topological sort for $\mathcal{T}_f$. □

The following theorem follows immediately from Lemma 5.6.

Theorem 5.7. *Testing parallel executions of a counter with reads, where the partial order is the union of chains, is NP-complete.*

For interval orders, we are not able to use Lemma 5.6 since the partial order it constructs is not guaranteed to be an interval order. Instead, we adapt a reduction used in [GK] to prove that verifying linearizability with read&write only is NP-complete. As in the proof of Lemma 5.6, we represent values as vectors in our adaptation.

Theorem 5.8. *Testing parallel executions of a counter with reads, where the partial order is an interval order, is NP-complete.*

Proof. Our reduction is from the Satisfiability problem (SAT). Consider an instance, $\mathcal{F}$, of SAT with n variables, $v_1, v_2, \ldots, v_n$, and m clauses, $C_1, C_2, \ldots, C_m$. Without loss of generality, assume that each variable and its negation appear in at least one clause, but not the same clause, and that there are no repeated variables in a clause. We construct a trace $\mathcal{T}_r$, an instance of the counter with reads problem for interval orders with at most $5nm + 10n + m + 1$ operations, corresponding to $\mathcal{F}$. To simplify the description, we have multiple intervals in $\mathcal{T}_r$ with common start times or end times; these ties can be broken arbitrarily to ensure unique interval endpoints.

A value in $\mathcal{T}_r$ will be represented as a vector, χ, with one entry, $\chi[v]$, in the vector for $v \in \{0, 1, \bar{1}, \hat{1}, 2, \bar{2}, \hat{2}, \ldots, n, \bar{n}, \hat{n}, c_1, c_2, \ldots, c_m\}$. The counter is initially the vector

χ_0, where $\chi_0[v] = 1$ if $v = 0$ and 0 otherwise. In what follows we use the notation *update*$(x\colon\ y, t_1, t_2)$ to indicate an operation with interval $[t_1, t_2]$ that updated the counter by decrementing its entry for x, incrementing its entry for y, and leaving its remaining entries unchanged. We use the notation *read*(y, t_1, t_2) to indicate an operation with interval $[t_1, t_2]$ that read the current counter value χ such that $\chi[y] = 1$ and for $v \neq y$, $\chi[v] = 0$.

Figure 16 depicts an example construction. For each clause C_j, $j = 1, \ldots, m$, we have a *read*$(c_j, 3, 5n + 4)$ operation, denoted the *clause read* for C_j. For $i = 1, \ldots, n$, let m_i ($m_{\bar{\imath}}$) be the number of clauses containing the literal v_i ($\bar{v}_i$, respectively). By assumption, $m_i > 0$, $m_{\bar{\imath}} > 0$, and $m_i + m_{\bar{\imath}} \leq m$. For $i = 1, \ldots, n + 1$, let $\Delta_i = (7m + 4)(i - 1) + 5n + 4$. For $i = 1, \ldots, n$, let $\Delta_{\bar{\imath}} = \Delta_i + 7m_i + 2$.

For each variable v_i, assignment to v_i is simulated using six operations:

- *update*$(0\colon\ i, 5i, 5i + 1)$,
- *update*$(0\colon\ \bar{\imath}, 5i, 5i + 1)$,
- *update*$(\hat{\imath}\colon\ 0, 5i, 5i + 1)$,
- *update*$(\hat{\imath}\colon\ 0, 5i + 3, 5i + 4)$,
- *update*$(i\colon\ \hat{\imath}, 5i, \Delta_i - 1)$,
- *update*$(\bar{\imath}\colon\ \hat{\imath}, 5i, \Delta_{\bar{\imath}} - 1)$.

There is a set of operations for each literal v_i, denoted the *group* of operations for i. For $i = 1, \ldots, n$, if $C_{i_1}, C_{i_2}, \ldots, C_{i_{m_i}}$ are the clauses containing the literal v_i, we have the following $5m_i + 2$ intervals. First, for C_{i_1}, we have:

- *read*$(i, 5i + 2, \Delta_i + 4)$,
- *update*$(i\colon\ c_{i_1}, 5i + 2, \Delta_i + 5)$,
- *update*$(0\colon\ c_{i_1}, \Delta_i + 1, \Delta_i + 2)$,
- *read*$(c_{i_1}, \Delta_i + 5, \Delta_i + 6)$,
- *update*$(c_{i_1}\colon\ i, \Delta_i + 3, \Delta_i + 7)$.

Then for C_{i_k}, $k = 2, \ldots, m_i$, we have:

- *update*$(i\colon\ c_{i_{k-1}}, \Delta_i + 7(k - 1), \Delta_i + 7(k - 1) + 4)$,
- *update*$(c_{i_{k-1}}\colon\ c_{i_k}, 5i + 2, \Delta_i + 7(k - 1) + 5)$,
- *update*$(c_{i_{k-1}}\colon\ c_{i_k}, \Delta_i + 7(k - 1) + 1, \Delta_i + 7(k - 1) + 2)$,
- *read*$(c_{i_k}, \Delta_i + 7(k - 1) + 5, \Delta_i + 7(k - 1) + 6)$,
- *update*$(c_{i_k}\colon\ i, \Delta_i + 7(k - 1) + 3, \Delta_i + 7k)$.

Finally, we have *update*$(c_{i_{m_i}}\colon\ \hat{\imath}, 5i + 2, \Delta_{\bar{\imath}} - 1)$ and *update*$(\hat{\imath}\colon\ 0, \Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}})$.

Likewise, there is a set of operations for each literal $\bar{v}_i$, denoted the *group* of operations for $\bar{\imath}$. For $i = 1, \ldots, n$, if $C_{\bar{i}_1}, C_{\bar{i}_2}, \ldots, C_{\bar{i}_{m_{\bar{\imath}}}}$ are the clauses containing the literal $\bar{v}_i$, we have the $5m_{\bar{\imath}} + 2$ intervals obtained from the previous definition by replacing $\Delta_{\bar{\imath}}$ with Δ_{i+1}, and leaving "$5i + 2$" unchanged but otherwise replacing i with $\bar{\imath}$ throughout.

This completes the construction.

Intuitively, the construction works as follows. Consider Figure 16, the changes of counter value as each operation is scheduled, and the corresponding truth assignment. Let the *cut-off point* denote the time stamp corresponding to the third dashed line from the top in the figure, where the clause reads end (i.e., time stamp $5n + 4$). Each variable is assigned in turn before the cut-off point. Consider the intervals that start above the

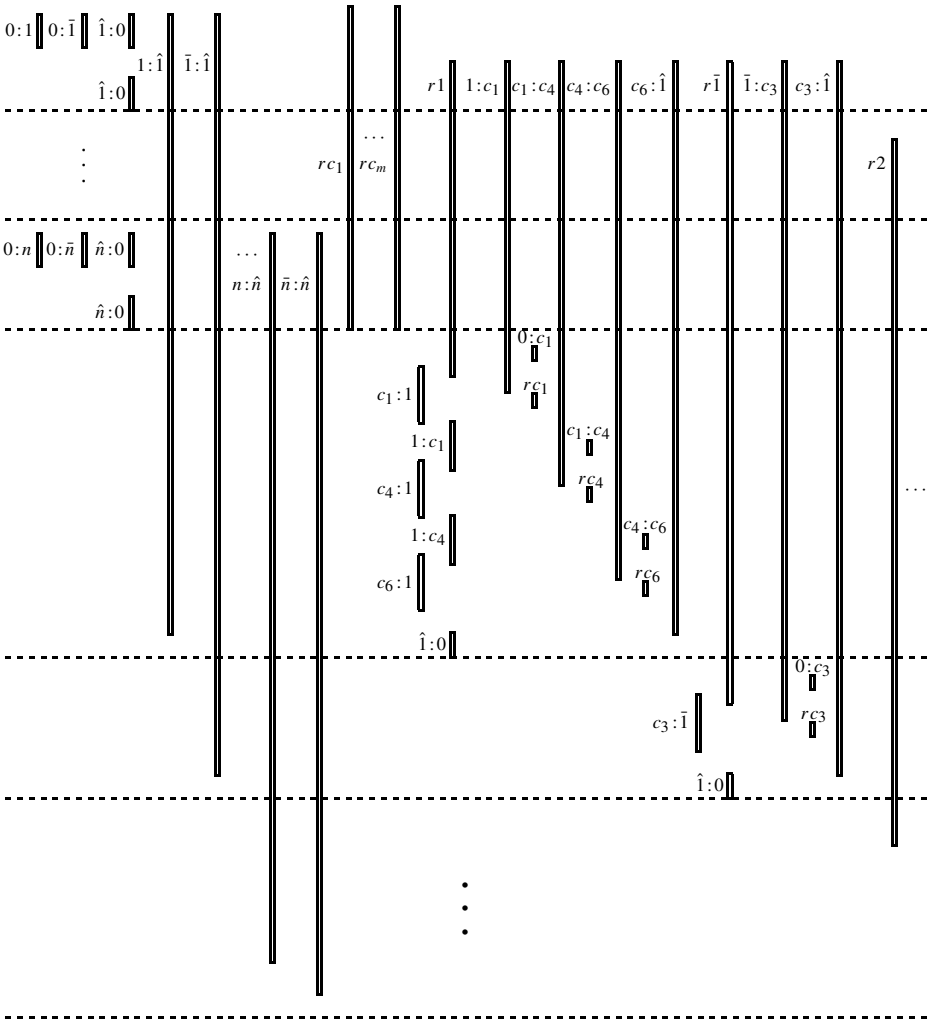

Fig. 16. (This figure is adapted from Figure 10 of [GK] © 1997.) Transforming an instance of SAT to an instance of the counter with reads problem for interval orders. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. The full construction has at most $5nm + 10n + m + 1$ operations. Here, the literal v_1 appears in exactly clauses C_1, C_4, and C_6, the literal $\bar{v}_1$ appears in clause C_3 only, and so forth. Vertical boxes depict the intervals of time for the respective *update* or *read* operations to the counter; time progresses from top to bottom in the figure. Update operations in the construction are labeled with two values: $x : y$. Read operations are labeled ry, where y is the value read.

top dashed line in the figure, corresponding to the setting of v_1. If v_1 is to be set to true, the initial counter transitions are 0, $\bar{1}$, $\hat{1}$, 0, 1. At this point, the $r1$ can be scheduled, followed by the operations in the group for i. As the counter transitions 1, c_1, c_4, c_6, $\hat{1}$, 0, the clause reads (rc_1, rc_4, rc_6) can be scheduled for the three clauses satisfied by setting v_1 to true. The key to the construction is to ensure that in a legal schedule, when a clause C_j has not been satisfied, the counter does not transition to the value c_j before the cut-off point. In Figure 16 the operations shown immediately below the cut-off point ensure that if v_1 is set to false, then all operations in the group for i are scheduled below the cut-off point (and hence too late for any clause read).

In more detail, consider a satisfying assignment that sets v_1 to true, and refer again to the figure. A legal schedule begins: $0 : \bar{1}$, then $\bar{1} : \hat{1}$, then the first $\hat{1} : 0$, then $0 : 1$, then $r1$, then $1 : c_1$ (immediately to the right of $r1$ in the figure), then the clause read rc_1, then $c_1 : c_4$, then the clause read rc_4 (not shown), then $c_4 : c_6$, then the clause read rc_6 (not shown), then $c_6 : \hat{1}$, and then the second $\hat{1} : 0$. The schedule continues with operations for $v_2, v_3, \ldots, v_n$, until the second $\hat{n} : 0$ is scheduled. Then the remaining (clean-up) operations for v_1 are scheduled, as follows: $0 : c_1$ (in the figure, just below the cut-off point), then rc_1, then $c_1 : 1$, then the $1 : c_1$ to its right, then the $c_1 : c_4$ to its right, then the rc_4 below it, then $c_4 : 1$, then $1 : c_4$, then the $c_4 : c_6$ to its right, then the rc_6 below it, then $c_6 : 1$, then $1 : \hat{1}$, and then the $\hat{1} : 0$ just above the fourth dashed line. Then $0 : c_3$, then $c_3 : \bar{1}$, then $r\bar{1}$, then $\bar{1} : c_3$, then rc_3, then $c_3 : \hat{1}$, and then the $\hat{1} : 0$ just above the fifth dashed line. The schedule continues with clean-up operations for $v_2, v_3, \ldots, v_n$, until all operations have been scheduled.

Claim 5.9. *Let $\mathcal{F}$ be an instance of a SAT problem, and let $\mathcal{T}_r$ be the trace constructed as described above. Then $\mathcal{T}_r$ is a valid counter with reads trace if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable, and let $\mathbf{T}$ be a satisfying truth assignment for $\mathcal{F}$. We construct the following topological sort for $\mathcal{T}_r$:

1. Repeat the following for $i = 1, 2, \ldots, n$:
   If v_i is set to true by $\mathbf{T}$, schedule $0 : \bar{\imath}$, then $\bar{\imath} : \hat{\imath}$, then the $\hat{\imath} : 0$ with interval $[5i, 5i + 1]$, then $0 : i$, and then ri. Schedule $i : c_{i_1}$, followed by, if it has not already been scheduled, the clause read rc_j, where $j = i_1$. Then repeat for $k = 2, \ldots, m_i$: schedule the $c_{i_{k-1}} : c_{i_k}$ with interval $[5i + 2, \Delta_i + 7(k - 1) + 5]$ followed by, if it has not already been scheduled, the clause read rc_j, where $j = i_k$. Finally, schedule $c_{i_{m_i}} : \hat{\imath}$, then the $\hat{\imath} : 0$ with interval $[5i + 3, 5i + 4]$.
   The case where v_i is set to false by $\mathbf{T}$ is symmetric, and left to the reader.
2. Since $\mathbf{T}$ is a satisfying assignment for $\mathcal{F}$, all clause reads have been scheduled by this point. Note as well that all the above operations can be scheduled prior to time Δ_1.
   Repeat the following for $i = 1, 2, \ldots, n$:
   If v_i is set to true by $\mathbf{T}$, consider the $4m_i$ unscheduled operations in group i, together with the unscheduled $i : \hat{\imath}$, and finally the $5m_{\bar{\imath}} + 2$ (unscheduled) operations in group $\bar{\imath}$:
   (a) Schedule $0 : c_{i_1}$, then rc_{i_1}, then $c_{i_1} : i$.

(b) Then repeat for $k = 2, \ldots, m_i$: Schedule $i : c_{i_{k-1}}$, then the unscheduled $c_{i_{k-1}} : c_{i_k}$, then rc_{i_k}, and then $c_{i_k} : i$.
(c) Schedule the unscheduled $i : \hat{\imath}$, then schedule the $\hat{\imath} : 0$ with interval $[\Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}}]$, to complete group i.
(d) Next schedule $0 : c_{\bar{\imath}_1}$, then $c_{\bar{\imath}_1} : \bar{\imath}$, then $r\bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_1}$, and then $rc_{\bar{\imath}_1}$.
(e) Then repeat for $k = 2, \ldots, m_{\bar{\imath}}$: Schedule the $c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}$ with interval $[\Delta_{\bar{\imath}} + 7(k-1)+1, \Delta_{\bar{\imath}}+7(k-1)+2]$, then $c_{\bar{\imath}_k} : \bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_{k-1}}$, then the unscheduled $c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}$, and then $rc_{\bar{\imath}_k}$.
(f) Finally, to complete group $\bar{\imath}$, schedule $c_{\bar{\imath}_{m_{\bar{\imath}}}} : \hat{\imath}$, and then the $\hat{\imath} : 0$ with interval $[\Delta_{i+1} - 1, \Delta_{i+1}]$.

The case where v_i is set to false by $\mathbf{T}$ is symmetric, and left to the reader.

The reader may verify that this is a legal schedule.

We next prove the converse, i.e., if $\mathcal{T}_r$ is a valid trace, then $\mathcal{F}$ is satisfiable. If $\mathcal{S}$ is a legal schedule for $\mathcal{T}_r$, then let $\mathbf{T}$ be the truth assignment defined as follows: for each v_i, if $update(0 : \bar{\imath}, 5i, 5i + 1)$ precedes $update(0 : i, 5i, 5i + 1)$ in $\mathcal{S}$, then $\mathbf{T}(v_i) = \mathtt{T}$, else $\mathbf{T}(v_i) = \mathtt{F}$.

We observe the following, for $i = 1, \ldots, n$: Both $update(0 : i, 5i, 5i + 1)$ and $update(0 : \bar{\imath}, 5i, 5i + 1)$ precede both ri and $r\bar{\imath}$ in $\mathcal{S}$. Thus if $\mathbf{T}(v_i) = \mathtt{F}$, then while $r\bar{\imath}$ may be scheduled in $\mathcal{S}$ prior to time $5i + 4$, ri cannot be. The ri operation must be scheduled after the only other operation that increments the ith vector entry prior to the end of its interval, namely, $c_{i_1} : i$. Hence $0 : c_{i_1}$ precedes $c_{i_1} : i$ precedes ri precedes rc_{i_1}. However, this implies that for $k = 2, \ldots, m_i$, $c_{i_{k-1}} : c_{i_k}$ precedes $c_{i_k} : i$ precedes $i : c_{i_{k-1}}$ precedes rc_{i_k}. The case where $\mathbf{T}(v_i) = \mathtt{T}$ is symmetric.

Suppose $\mathbf{T}$ does not satisfy a clause C_j. Consider the literals in C_j sorted by their index, and let x be the number of literals in C_j. For $k = 1, \ldots, x$, define $j_k \in \{1, \bar{1}, 2, \bar{2}, \ldots, n, \bar{n}\}$ such that $j_k = i$ (or $\bar{\imath}$) if and only if v_i ($\bar{v}_i$, respectively) is the kth literal in C_j. We claim that all operations that increment the c_jth vector entry are scheduled in $\mathcal{S}$ after Δ_1. The proof is by induction on decreasing k. Consider the last rc_j in $\mathcal{S}$, in group j_x. Due to the argument given in the preceding paragraph, there is only one operation that increments the c_jth entry that could have been scheduled so as to be read by the rc_j: the sole such operation in group j_x whose interval is not strictly after Δ_{j_x}. Assume inductively that all operations that increment the c_jth entry from groups j_{k+1} to j_x are scheduled after $\Delta_{j_{k+1}}$. Thus due to the argument given in the preceding paragraph, there is only one such operation that could have been scheduled so as to be read by the rc_j: the sole operation that increments the c_jth entry in group j_k whose interval is not strictly after Δ_{j_k}. The claim follows by induction.

Since there is no operation that increments the c_jth vector entry in $\mathcal{S}$ until after $\Delta_1 = 5n + 4$, but the clause read $read(c_j, 3, 5n + 4)$ must be scheduled no later than Δ_1, $\mathcal{S}$ is not a legal schedule, a contradiction.

Hence $\mathbf{T}$ is a satisfying assignment for $\mathcal{F}$, completing the proof of Claim 5.9. □

Since the above transformation can be done in polynomial time, Theorem 5.8 follows. □

6. Extensions

This paper provides the first systematic study of algorithms and hardness results for testing the correctness of parallel data structure executions by postmortem analysis of traces. We present a series of results for testing parallel executions of queues, priority queues, stacks, and various types of counters for various value and partial order types. We have focused on black-box testing procedures, with low overhead, in which each data object is tested individually. In the remainder of this section we briefly discuss some variations on the testing framework considered thus far.

First, the algorithms presented in this paper can be extended to report more than just a YES/NO answer. For example, when the trace is invalid, the algorithms can be readily extended to report an offending event pair, or perhaps all uncovered offending pairs.

Second, one can consider testing of partial traces, in order to provide earlier feedback. For example, the algorithms for linearizable data objects can be readily extended to test partial traces up through a given timestamp.

Third, one could consider parallel algorithms for testing parallel executions. Each of the fast algorithms for linearizable data structures are amenable to efficient parallel implementation. The time complexities depend on the model of parallel computation used. For example, on a QRQW PRAM, Algorithms $\mathcal{A}$ and $\mathcal{F}$ can each be implemented in $O(n)$ work and $O(\log n)$ time with high probability,[6] using standard techniques.

Fourth, invasive procedures can be used that either modify the values inserted into the data structure (e.g., [GK]) or modify the data structure implementation itself (e.g., [SM1], [SM2], [BS], and [GK]) to assist in testing. There is a clear benefit to doing this in our case: the NP-hardness results for arbitrary values can be overcome by tagging each value inserted into the data structure with a unique index; our polynomial time algorithms for distinct-values traces can then be used.

Fifth, there may be scenarios where traces contain events on multiple objects, and we seek a single topological sort that preserves the semantics of each object. The NP-completeness results in this paper for individual objects clearly imply NP-completeness results for collective objects. On the positive side, each linear/$O(n \log n)$/polynomial time algorithm for a single linearizable data object implies a linear/$O(n \log n)$/polynomial time algorithm, respectively, for traces intermixing any number of linearizable data objects, since linearizability is a *local* property [HW]. Beyond these results, most of the problems regarding testing collective objects remain to be studied.

Finally, one might rely more on proofs of correctness. For example, one could prove that a shared memory algorithm, maybe even an implementation, is correct based on the *assumption* that the shared memory is sequentially consistent or linearizable. However, this just moves the problem to a different level. Existing proofs that a memory system always preserves sequential consistency or linearizability assume fault-free hardware (e.g., [GMG], [PD], and [PSCH]), and even testing whether the memory system preserved sequential consistency or linearizability for a single program run is NP-complete [GK].

---

[6] That is, a randomized algorithm achieves the stated work and time bounds with probability $\geq 1 - n^{-c}$, for any constant c.

References

[AGMT] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proc*. 11*th ACM Symp*. *on Principles of Distributed Computing*, pages 47–58, August 1992.

[AHS] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[AW] H. M. Abdel-Wahab. Scheduling with Applications to Register Allocation and Deadlock Problems. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1976.

[AWK] H. M. Abdel-Wahab and T. Kameda. Scheduling to minimize maximum cumulative cost subject to series-parallel precedence constraints. *Operations Research*, 26:141–158, 1978.

[BA] P. Banerjee and J. A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *IEEE Transactions on Computers*, C-35(4):296–306, 1986.

[BB] V. Balasubramanian and P. Banerjee. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. *IEEE Transactions on Computers*, C-39(4):436–459, 1990.

[BC] J. L. Bruno and E. C. Coffman, Jr. Optimal Fault-Tolerant Computing on Two Parallel Processors. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, October 1994.

[BEG$^+$] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[BGM] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc*. 7*th ACM Symp*. *on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[BK1] M. Blum and S. Kannan. Designing programs that check their work. In *Proc*. 21*st ACM Symp*. *on Theory of Computing*, pages 86–97, May 1989.

[BK2] S. K. Biswas and S. Kannan. Minimizing space usage in computations. In *Proc*. 15*th Symp*. *on Foundations of Software Technology and Theoretical Computer Science*, pages 377–390, December 1995.

[BLR] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc*. 22*nd ACM Symp*. *on Theory of Computing*, pages 73–83, May 1990.

[BRS$^+$] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Mair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, C-39(9):1132–1245, 1990.

[BS] J. Bright and G. Sullivan. Checking mergeable priority queues. In *Proc*. 24*th IEEE Fault-Tolerant Computing Symp*., pages 144–153, June 1994.

[DRT] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[FM] U. Finkler and K. Mehlhorn. Checking priority queues. In *Proc*. 10*th ACM–SIAM Symp*. *on Discrete Algorithms*, pages S901–S902, January 1999.

[GBP] P. B. Gibbons, J. L. Bruno, and S. Phillips. Post-mortem black-box correctness tests for basic parallel data structures. In *Proc*. 11*th ACM Symp*. *on Parallel Algorithms and Architectures*, pages 44–53, June 1999.

[GBT] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc*. 16*th ACM Symp*. *on Theory of Computing*, pages 135–143, 1984.

[GK] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997. Preliminary versions appeared in *Proc*. 1992 *IEEE Symp*. *on Parallel and Distributed Processing* and *Proc*. 1994 *ACM Symp*. *on Parallel Algorithms and Architectures*.

[GMG] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc*. 3*rd ACM Symp*. *on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[HA] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[HW] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[It] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[La] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LPvL] H. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Lecture Notes in Computer Science, Vol. 314. Springer-Verlag, Berlin, 1988.

[Ly] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[MS] C. L. Monma and J. B. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4(3):215–224, 1979.

[Pa] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD, 1986.

[PD] S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 288–296, June 1996.

[PSCH] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proc. 10th ACM Symp. on Parallel Algorithms and Architectures*, pages 67–76, June–July 1998.

[Ra] P. Ramanan. Testing the optimality of alphabetic trees. *Theoretical Computer Science*, 93:279–302, 1992.

[Ru] R. Rubinfeld. Designing checkers for programs that run in parallel. *Algorithmica*, 15(4):287–301, 1996.

[SM1] G. F. Sullivan and G. M. Masson. Using certification trails to achieve software fault tolerance. In *Proc. 20th IEEE Fault-Tolerant Computing Symp.*, pages 423–431, 1990.

[SM2] G. F. Sullivan and G. M. Masson. Certification trails for data structures. In *Proc. 21st IEEE Fault-Tolerant Computing Symp.*, pages 240–247, 1991.

[ST] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 54–63, July 1995.

[SUZ] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 33–41, June 1996.

[SZ] N. Shavit and A. Zemach. Combining funnels. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, pages 61–70, June–July 1998.

[WG] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17:164–182, 1993.

Online publication June 21, 2002.