DISC '04

Nonblocking Concurrent Data Structures with Condition Synchronization*

William N. Scherer III and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scherer,scott}@cs.rochester.edu

Abstract. We apply the classic theory of linearizability to operations that must wait for some other thread to establish a precondition. We model such an operation as a *request* and a *follow-up*, each with its own linearization point. Linearization of the request marks the point at which a thread's wishes become visible to its peers; linearization of the follow-up marks the point at which the request is fulfilled and the operation takes effect. By placing both linearization points within the purview of object semantics, we can specify not only the effects of operations, but also the order in which pending requests should be fulfilled.
We use the term *dual data structure* to describe a concurrent object implementation that may hold both data and *reservations* (registered requests). By reasoning separately about a request, its successful follow-up, and the period in-between, we obtain meaningful definitions of nonblocking dual data structures. As concrete examples, we present lock-free *dualstacks* and *dualqueues*, and experimentally compare their performance with that of lock-based and nonblocking alternatives.

1 Introduction

Since its introduction nearly fifteen years ago, linearizability has become the standard means of reasoning about the correctness of concurrent objects. Informally, linearizability "provides the illusion that each operation… takes effect instantaneously at some point between its invocation and its response" [3, abstract]. Linearizability is "nonblocking" in the sense that it never requires a call to a total method (one whose precondition is simply **true**) to wait for the execution of any other method. (Certain other correctness criteria, such as serializability [10], may require blocking, e.g. to enforce coherence across a multi-object system.) The fact that it is nonblocking makes linearizability particularly attractive for reasoning about nonblocking *implementations* of concurrent objects, which provide guarantees of various strength regarding the progress of method calls in practice. In a *wait-free* implementation, every contending thread is guaranteed to complete its method call within a bounded number of its own time steps [4]. In a *lock-free* implementation, *some* some contending thread is guaranteed to complete its

* This work was supported in part by NSF grants numbers EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by Sun Microsystems Laboratories.

method call within a bounded number of steps (from any thread's point of view) [4]. In an *obstruction-free* implementation, a thread is guaranteed to complete its method call within a bounded number of steps in the absence of contention, i.e. if no other threads execute competing methods concurrently [2].

These various *progress conditions* all assume that every method is total. As Herlihy puts it [4, p. 128]:

> We restrict our attention to objects whose operations are total because it is unclear how to interpret the wait-free condition for partial operations. For example, the most natural way to define the effects of a partial *deq* in a concurrent system is to have it wait until the queue becomes nonempty, a specification that clearly does not admit a wait-free implementation.

To avoid this problem the designers of nonblocking data structures typically "totalize" their methods by returning an error flag whenever the current state of the object does not admit the method's intended behavior.

But partial methods are important! Many applications need a `dequeue`, `pop`, or `deletemin` operation that waits when its structure is empty; these and countless other examples of *condition synchronization* are fundamental to concurrent programming.

Given a nonblocking data structure with "totalized" methods, the obvious spin-based strategy is to embed each call in a loop, and retry until it succeeds. This strategy has two important drawbacks. First, it introduces unnecessary contention for memory and communication bandwidth, which may significantly degrade performance, even with careful backoff. Second, it provides no fairness guarantees.

Consider a total queue whose `dequeue` method waits until it can return successfully, and a sequence of calls by threads A, B, C, and D:

C enqueues a 1
D enqueues a 2
A calls dequeue
A's call returns the 2
B calls dequeue
B's call returns the 1

This is clearly a "bad" execution history, because it returns results in the wrong (non-FIFO) order; it implies an incorrect implementation. The following is clearly a "good" history:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
A's call returns the 1
B's call returns the 2

But what about the following:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
B's call returns the 1
A's call returns the 2

If the first line is known to have occurred before the second (this may be the case, for example, if waiting threads can be identified by querying the scheduler, examining a thread control block, or reading an object-specific flag), then intuition suggests that while this history returns results in the right order, it returns them to the wrong threads. If we implement our queue by wrapping the nonblocking "totalized" `dequeue` in a loop, then this third, questionable history may certainly occur.

In the following section we show how to apply the theory of linearizability in such a way that object semantics can specify the order in which pending requests will be fulfilled. We then propose that data structures implement those semantics by explicitly representing the set of pending requests. Borrowing terminology from the BBN Butterfly Parallel Processor of the early 1980s [1], we define a *dual* data structure to be one that may hold *reservations* (registered requests) instead of, or in addition to, data. A *nonblocking* dual data structure is one in which (a) every operation either completes or registers a request in a nonblocking fashion, (b) fulfilled requests complete in a nonblocking fashion, and (c) threads that are waiting for their requests to be fulfilled do not interfere with the progress of other threads.

As concrete examples, we introduce two lock-free dual data structures in Section 3: a *dualstack* and a *dualqueue*. The dualstack both returns results and fulfills requests in LIFO order; the dualqueue does both in FIFO order. Both structures are attractive candidates for "bag of tasks" programming on multiprocessor systems. The dualqueue also subsumes scalable queue-based spin locks and semaphores, and can be used in conjunction with a small-scale test-and-set lock to obtain a *limited contention* spin lock that embodies an explicit tradeoff between fairness and locality on distributed shared memory machines. Preliminary performance results for dualstacks and dualqueues appear in section 4. We summarize our findings and suggest directions for future work in Section 5.

2 Definitions

2.1 Linearizable Objects

Following Herlihy and Wing [3], a *history* of an object is a (potentially infinite) sequence of method invocation events $\langle m(args)\ t\rangle$ and response (return) events $\langle r(val)\ t\rangle$, where m is the name of a method, r is a return condition (usually "ok"), and t identifies a thread. An invocation *matches* the next response in the sequence that has the same thread id. Together, an invocation and its matching response are called an *operation*. The invocation and response of operation o may also be denoted $inv(o)$ and $res(o)$, respectively. If event e_1 precedes event e_2 in history H, we write $e_1 <_H e_2$.

A history is *sequential* if every response immediately follows its matching invocation. A non-sequential history is *concurrent*. A *thread subhistory* is the subsequence

of a history consisting of all events for a given thread. Two histories are *equivalent* if all their thread subhistories are identical. We consider only *well-formed* concurrent histories, in which every thread subhistory is sequential, and begins with an invocation.

We assume that the semantics of an object (which we do not consider formally here) uniquely determine a set of *legal* sequential histories. In a queue, for example, items must be inserted and removed in FIFO order. That is, the nth successful `dequeue` in a legal history must return the value inserted by the nth `enqueue`. Moreover at any given point the number of prior `enqueue`s must equal or exceed the number of successful `dequeue`s. To permit `dequeue` calls to occur at any time (i.e., to make `dequeue` a *total* method—one whose precondition is simply **true**), one can allow unsuccessful `dequeue`s [$\langle deq(\,)\ t\rangle\ \langle no(\bot)\ t\rangle$] to appear in the history whenever the number of prior `enqueue`s equals the number of prior successful dequeues.

A (possibly concurrent) history H induces a partial order $\prec_H$ on operations: $o_i \prec_H o_j$ if $res(o_i) <_H inv(o_j)$. H is *linearizable* if (a) it is equivalent to some legal sequential history S, and (b) $\prec_H \subseteq \prec_S$.

Departing slightly from Herlihy and Wing, we introduce the notion of an *augmented history* of an object. A (well-formed) augmented history H' is obtained from a history H by inserting a *linearization point* $\langle m^l(args, val)\ t\rangle$ (also denoted $lin(o)$) somewhere between each response and its previous matching invocation: $inv(o) <_{H'} lin(o) <_{H'} res(o)$.

If H is equivalent to some legal sequential history S and the linearization points of H' appear in the same order as the corresponding operations in S, then H' embodies a linearization of H: the order of the linearization points defines a total order on operations that is consistent with the partial order induced by H. Put another way: $res(o_i) <_H inv(o_j) \Rightarrow lin(o_i) <_{H'} lin(o_j)$. Given this notion of augmented histories, we can define legality without resort to equivalent sequential histories: we say that an augmented history is *legal* if its sequence of linearization points is permitted by the object semantics. Similarly, H is *linearizable* if it can be augmented to produce a legal augmented history H'.

2.2 Implementations

We define an *implementation* of a concurrent object as a pair $(\mathcal{E}, \mathcal{I})$, where

1. $\mathcal{E}$ is a set of valid *executions* of some operational system, e.g. the possible interleavings of machine instructions among threads executing a specified body of C code on some commercial multiprocessor. Each execution takes the form of a series of *steps* (e.g., instructions), each of which is identified with a particular thread and occurs atomically. Implementations of nonblocking concurrent objects on real machines typically rely not only on atomic loads and stores, but on such *universal* atomic primitives [4] as `compare_and_swap` or `load_linked` and `store_conditional`, each of which completes in a single step.
2. $\mathcal{I}$ is an *interpretation* that maps each execution $E \in \mathcal{E}$ to some augmented object history $H' = \mathcal{I}(E)$ whose events (including the linearization points) are identified with steps of E in such a way that if $e_1 <_{H'} e_2$, then $s(e_1) \leq_E s(e_2)$, where $s(e)$ for any event e is the step identified with e.

We say an implementation is *correct* if $\forall E \in \mathcal{E}$, $\mathcal{I}(E)$ is a legal augmented history of the concurrent object.

In practice, of course, steps in an execution have an observable order only if they are executed by the same thread, or if there is a data dependence between them [7]. In particular, while we cannot in general observe that $s(inv(o_i)) <_E s(inv(o_j))$, where o_i and o_j are performed by different threads, we can observe that $s(res(o_i)) <_E s(inv(o_j))$, because o_i's thread may write a value after its response that is read by o_j's thread before its invocation. The power of linearizability lies in its insistence that the semantic order of operations on a concurrent object be consistent with such externally observable orderings.

In addition to correctness, an implementation may have a variety of other properties of interest, including bounds on time (steps), space, or remote memory accesses; the suite of required atomic instructions; and various *progress conditions*. An implementation is *wait-free* if we can bound, for all executions E and invocations $inv(o)$ in $\mathcal{I}(E)$, the number of steps between (the steps identified with) $inv(o)$ and $res(o)$. An implementation is *lock-free* if for all invocations $inv(o_i)$ we can bound the number of steps between (the steps identified with) $inv(o_i)$ and the (not necessarily matching) first subsequent response $res(o_j)$. An implementation is *obstruction-free* if for all threads t and invocations $inv(o)$ performed by t we can bound the number of consecutive steps performed by t (with no intervening steps by any other thread) between (the steps identified with) $inv(o)$ and $res(o)$. Note that the definitions of lock freedom and obstruction freedom permit executions in which an invocation has no matching response (i.e., in which threads may starve).

2.3 Adaptation to Objects with Partial Methods

When an object has partial methods, we divide each such method into a *request* method and a *follow-up* method, each of which has its own invocations and responses. A total queue, for example, would provide `dequeue_request` and `dequeue_followup` methods. By analogy with Lamport's bakery algorithm [6], the request method returns a *ticket*, which is then passed as an argument to the follow-up method. The follow-up, for its part, returns either the desired result or, if the method's precondition has not yet been satisfied, an error indication.

The history of a concurrent object now consists not only of invocation and response events $\langle m(args)\ t\rangle$ and $\langle ok(val)\ t\rangle$ for total methods m, but also request invocation and response events $\langle p_{req}(args)\ t\rangle$ and $\langle ok(tik)\ t\rangle$, and follow-up invocation and response events $\langle p_{fol}(tik)\ t\rangle$ and $\langle r(val)\ t\rangle$ for partial methods p. A request invocation and its matching response are called a *request operation*; a follow-up invocation and its matching response are called a *follow-up operation*. The request invocation and response of operation o may be denoted $inv(o^r)$ and $res(o^r)$; the follow-up invocation and response may be denoted $inv(o^f)$ and $res(o^f)$.

A follow-up with ticket argument k *matches* the previous request that returned k. A follow-up operation is said to be *successful* if its response event is $\langle ok(val)\ t\rangle$; it is said to be *unsuccessful* its its response event is $\langle no(\perp)\ t\rangle$. We consider only well-formed histories, in which every thread subhistory is sequential, and is a prefix of some string

in the regular set $(ru^\star s)^\star$, where r is a request, u is an unsuccessful follow-up that matches the preceding r, and s is a successful follow-up that matches the preceding r.

Because it consists of a sequence of operations (beginning with a request and ending with a successful response), a call to a partial method p has a sequence of linearization points, including an *initial* linearization point $\langle p^i(args)\ t\rangle$ somewhere between the invocation and the response of the request, and a *final* linearization point $\langle p^f(val)\ t\rangle$ somewhere between the invocation and the response of the successful matching follow-up. The initial and final linearization points for p may also be denoted $in(p)$ and $fin(p)$.

We say an augmented history is *legal* if its sequence of linearization points is among those determined by the semantics of the object. This definition allows us to capture partial methods in object semantics. In the previous section we suggested that the semantics of a queue might require that (1) the nth successful `dequeue` returns the value inserted by the nth `enqueue`, and (2) the number of prior `enqueue`s at any given point equals or exceeds the number of prior successful `dequeue`s. We can now instead require that (1′) the nth final linearization point for `dequeue` contains the value from the linearization point of the nth `enqueue`, (2′) the number of prior linearization points for `enqueue` equals or exceeds the number of prior final linearization points for `dequeue`, and (3′) at the linearization point of an unsuccessful `dequeue_followup`, the number of prior linearization points for `enqueue` exactly equals the number of prior final linearization points for `dequeue` (i.e., linearization points for successful `dequeue_followup`s). These rules ensure not only that the queue returns results in FIFO order, but also that pending requests for partial methods (which are now permitted) are fulfilled in FIFO order.

As before, a history H is linearizable if it can be augmented to produce a legal augmented history H', and an implementation $(\mathcal{E}, \mathcal{I})$ is correct if $\forall E \in \mathcal{E}$, $\mathcal{I}(E)$ is a legal augmented history of the concurrent object.

Given the definition of well-formedness above, a thread t that wishes to execute a partial method p must first call p`_request` and then call p`_followup` in a loop until it succeeds. This is very different from calling a traditional "totalized" method until it succeeds: linearization of a distinguished request operation is the hook that allows object semantics to address the order in which pending requests will be fulfilled.

As a practical matter, implementations may wish to provide a p`_demand` method that waits until it can return successfully, and/or a plain p method equivalent to p`_demand(`p`_request)`. The obvious implementation of p`_demand` contains a busy-wait loop, but other implementations are possible. In particular, an implementation may choose to use scheduler-based synchronization to put t to sleep on a semaphore that will be signaled when p's precondition has been met, allowing the processor to be used for other purposes in the interim. We require that it be possible to provide request and follow-up methods, as defined herein, with no more than trivial modifications to any given implementation. The algorithms we present in Section 3 provide only a plain p interface, with internal busy-wait loops.

Progress Conditions When reasoning about progress, we must deal with the fact that a partial method may wait for an arbitrary amount of time (perform an arbitrary number of unsuccessful follow-ups) before its precondition is satisfied. Clearly we wish to require

that requests and follow-ups are nonblocking. But this is not enough: we must also prevent unsuccessful follow-ups from interfering with progress in other threads. We do so by prohibiting such operations from accessing remote memory. On a cache-coherent machine, an access by thread t within operation o is said to be *remote* if it writes to memory that may (in some execution) be read or written by threads other than t more than a constant number of times between $inv(o^r)$ and $res(o^f)$, or if it reads memory that may (in some execution) be written by threads other than t more than a constant number of times between $inv(o^r)$ and $res(o^f)$. On a non-cache-coherent machine, an access by thread t is also remote if it refers to memory that t itself did not allocate.

2.4 Dual Data Structures

We define a *dual data structure* D to be a concurrent object implementation that may hold *reservations* (registered requests) instead of, or in addition to, data. Reservations correspond to requests that cannot be fulfilled until the object satisfies some necessary precondition. A reservation may be removed from D, and a call to its follow-up method may return, when some call by another thread makes the precondition true. D is a *nonblocking* dual data structure if

1. It is a correct implementation of a linearizable concurrent object, as defined above.
2. All operations, including requests and follow-ups, are nonblocking.
3. Unsuccessful follow-ups perform no remote memory accesses.

Nonblocking dual data structures may be further classified as wait-free, lock-free, or obstruction-free, depending on their guarantees with respect to condition (2) above. In the following section we consider concrete lock-free implementations of a dualstack and a dualqueue.

3 Example Data Structures

Space limitations preclude inclusion of pseudocode in the conference proceedings. Both example data structures can be found on-line at www.cs.rochester.edu/u/scott/synchronization/pseudocode/duals.html. Both use a double-width `compare_and_swap` (CAS) instruction (as provided, for example, on the Sparc) to create "counted pointers" that avoid the ABA problem: each vulnerable pointer is paired with a serial number, which is incremented every time the pointer is updated to a non-`NULL` value. We assume that no thread can stall long enough to see a serial number repeat. On a machine with (single-word) `load_linked`/`store_conditional` (LL/SC) instructions, the serial numbers would not be needed.[1]

[1] CAS takes an address, an expected value, and a new value as argument. If the expected value is found at the given address, it is replaced with the new value, atomically; a Boolean return value indicates whether the replacement occurred. The ABA problem [5] can arise in a system in which memory is dynamically allocated, freed, and then reallocated: a thread that performs a load followed by a CAS may succeed when it should not, if the value at the location in question has changed *and then changed back* in-between. LL reads a memory location and makes a note of having done so. SC stores a new value to the location accessed by the most recent LL, provided that no other thread has modified the location in-between. Again, a Boolean return value indicates whether the store occurred.

3.1 The Dualstack

The dualstack is based on the standard lock-free stack of Treiber [13]. So long as the number of calls to `pop` does not exceed the number of calls to `push`, the dualstack behaves the same as its non-dual cousin.

When the stack is empty, or contains only reservations, the `pop` method pushes a reservation, and then spins on the `data_node` field within it. A `push` method always pushes a data node. If the previous top node was a reservation, however, the two adjacent nodes "annihilate each other": any thread that finds a data node and an underlying reservation at the top of the stack attempts to (a) write the address of the former into the `data_node` field of the latter, and then (b) pop both nodes from the stack. At any given time, the stack contains either all reservations, all data, or one datum (at the top) followed by reservations.

Both the head pointer and the `next` pointers in stack nodes are *tagged* to indicate whether the next node in the list is a reservation or a datum and, if the latter, whether there is a reservation beneath it in the stack. We assume that nodes are word-aligned, so that these tags can fit in the low-order bits of a pointer. For presentation purposes the on-line pseudocode assumes that data values are integers, though this could obviously be changed to any type (including a pointer) that will fit, together with a serial number, in the target of a double-width CAS (or in a single word on a machine with LL/SC). To differentiate between the cases where the topmost data node is present to fulfill a request and where the stack contains all data, pushes for the former case set both the data and reservation tags; pushes for the latter set only the data tag.

As mentioned in Section 2.3 our code provides a single `pop` method that subsumes the sequence of operations from a `pop` request through its successful follow-up. The initial linearization point in `pop`, like the linearization point in `push`, is the CAS that modifies the top-of-stack pointer. For `pop`s when the stack is non-empty, this CAS is also the final linearization point. For `pop`s that have to spin, the final linearization point is the CAS (in some other thread) that writes to the `data_node` field of the requester's reservation, terminating its spin.

The code for `push` is lock-free, as is the code from the beginning of `pop` to the initial linearization point, and from the final linearization point (the read that terminates the spin) to the end of `pop`. Moreover the spin in `pop` (which would comprise the body of an unsuccessful follow-up operation, if we provided it as a separate method), is entirely local: it reads only the requester's own reservation node, which the requester allocated itself, and which no other thread will write except to terminate the spin. The dualstack therefore satisfies conditions 2 and 3 of Section 2.4.

Though we do not offer a proof, inspection of the code confirms that the dualstack satisfies the usual LIFO semantics for total methods: if the number of previous linearization points for `push` exceeds the number of previous initial linearization points for `pop`, then a new `pop` operation p will succeed immediately, and will return the value provided by the most recent previous `push` operation h such that the numbers of pushes and pops that linearized between h and p are equal. In a similar fashion, the dualstack satisfies pending requests in LIFO order: if the number of previous initial linearization points for `pop` exceeds the number of previous linearization points for `push`, then a `push` operation h will provide the value to be returned by the most recent previous `pop`

operation p such that the numbers of pushes and pops that linearized between p and h are equal. This is condition 1 from Section 2.4.

The spin in `pop` is terminated by a CAS in some other thread (possibly the fulfilling thread, possibly a helper) that updates the `data_node` field in the reservation. This CAS is the final linearization point of the spinning thread. It is not, however, the final linearization point of the fulfilling thread; that occurs earlier, when the fulfilling thread successfully updates the top-of-stack pointer to point to the fulfilling datum. Once the fulfilling `push` has linearized, no thread will be able to make progress until the spinning `pop` reaches its final linearization point. It is possible, however, for the spinning thread to perform an unbounded number of (local, spinning) steps in a legal execution before this happens: hence the need to separate the linearization points of the fulfilling and fulfilled operations.

It is tempting to consider a simpler implementation in which the fulfilling thread pops a reservation from the stack and then writes the fulfilling datum directly into the reservation. This implementation, however, is incorrect: it leaves the requester vulnerable to a failure or stall in the fulfilling thread subsequent to the pop of the reservation but prior to the write of the datum. Because the reservation would no longer be in the stack, an arbitrary number of additional `pop` operations (performed by other threads, and returning subsequently `push`ed data) could linearize before the requester's successful follow-up.

One possible application of a dualstack is to implement a "bag of tasks" in a locality-conscious parallel execution system. If newly created tasks share data with recently completed tasks, it may make sense for a thread to execute a newly created task, rather than one created long ago, when it next needs work to do. Similarly, if there is insufficient work for all threads, it may make sense for newly created tasks to be executed by threads that became idle recently, rather than threads that have been idle for a long time. In addition to enhancing locality, this could allow power-aware processors to enter a low-power mode when running spinning threads, potentially saving significant energy. The LIFO ordering of a dualstack will implement these policies.

3.2 The Dualqueue

The dualqueue is based on the M&S lock-free queue [9]. So long as the number of calls to `dequeue` does not exceed the number of calls to `push`, it behaves the same as its non-dual cousin. It is initialized with a single "dummy" node; the first real datum (or reservation) is always in the second node, if any. At any given time the second and subsequent nodes will either all be reservations or all be data.

When the queue is empty, or contains only reservations, the `dequeue` method enqueues a reservation, and then spins on the `request` pointer field of the former tail node. The `enqueue` method, for its part, fulfills the request at the head of the queue, if any, rather than enqueue a datum. To do so, the fulfilling thread uses a CAS to update the reservation's `request` field with a pointer to a node (outside the queue) containing the provided data. This simultaneously fulfills the request and breaks the requester's spin. Any thread that finds a fulfilled request at the head of the queue removes and frees it. (NB: acting on the head of the queue requires that we obtain a *consistent snapshot* of the `head`, `tail`, and `next` pointers. Extending the technique of the original M&S

queue, we use a two-stage check to ensure sufficient consistency to prevent untoward race conditions.)

As in the dualstack, queue nodes are tagged as requests by setting a low-order bit in pointers that point to them. We again assume, without loss of generality, that data values are integers, and we provide a single `dequeue` method that subsumes the sequence of operations from a `dequeue` request through its successful follow-up.

The code for `enqueue` is lock-free, as is the code from the beginning of `dequeue` to the initial linearization point, and from the final linearization point (the read that terminates the spin) to the end of `dequeue`. The spin in `dequeue` (which would comprise the body of an unsuccessful follow-up) accesses a node that no other thread will write except to terminate the spin. The dualqueue therefore satisfies conditions 2 and 3 of Section 2.4 on a cache-coherent machine. (On a non-cache-coherent machine we would need to modify the code to provide an extra level of indirection; the spin in `dequeue` reads a node that the requester did not allocate.)

Though we do not offer a proof, inspection of the code confirms that the dualqueue satisfies the usual FIFO semantics for total methods: if the number of previous linearization points for `enqueue` exceeds the number of previous initial linearization points for `dequeue`, then a new, nth `dequeue` operation will return the value provided by the nth `enqueue`. In a similar fashion, the dualqueue satisfies pending requests in FIFO order: if the number of previous initial linearization points for `dequeue` exceeds the number of previous linearization points for `enqueue`, then a new, nth `enqueue` operation will provide a value to the nth `dequeue`. This is condition 1 from Section 2.4.

The spin in `dequeue` is terminated by a CAS in another thread's `enqueue` method; this CAS is the linearization point of the `enqueue` and the final linearization point of the `dequeue`. Note again that a simpler algorithm, in which the `enqueue` method could remove a request from the queue and then fulfill it, would not be correct: the CAS operation used for removal would constitute the final linearization point of the `enqueue`, but the corresponding `dequeue` could continue to spin for an arbitrary amount of time if the thread performing the `enqueue` were to stall.

Dualqueue Applications Dualqueues are versatile. They can obviously be used as a traditional "bag of tasks" or a producer–consumer buffer. They have several other uses as well:

Mutual exclusion. A dualqueue that is initialized to hold a single datum is a previously unknown variety of queue-based mutual exclusion lock. Unlike the widely used MCS lock [8], a dualqueue lock has no spin in the release code: where the MCS lock updates the tail pointer of the queue and then the `next` pointer of the predecessor's node, a dualqueue lock updates the `next` pointer first, and then swings the tail to cover it.

Semaphores. A dualqueue that is initialized with k data nodes constitutes a contention-free spin-based semaphore. It can be used, for example, to allocate k interchangeable resources among a set of competing threads.

Limited contention lock. As noted by Radovic and Hagersten [11], among others, the strict fairness of queue-based locks may not be desirable on a non-uniform memory

access (distributed shared memory) multiprocessor. At the same time, a test-and-set lock, which tends to grant requests to physically nearby threads, can be unacceptably unfair (not to mention slow) when contention among threads is high: threads that are physically distant may starve. An attractive compromise is to allow waiting threads to bypass each other in line to a limited extent. A dualqueue paired with a test-and-set lock provides a straightforward implementation of such a "limited contention" lock. We initialize the dualqueue with k tokens, each of which grants permission to contend for the test-and-set lock. The value of k determines the balance between fairness and locality. The `acquire` operation first dequeues a token from the dualqueue and then contends for the test-and-set lock. The `release` operation enqueues a token in the dualqueue and releases the test-and-set lock. Starvation is still possible, though less likely than with an ordinary test-and-set lock. We can eliminate it entirely, if desired, by reducing k to one on a periodic basis.

4 Experimental Results

In this section we compare the performance of the dualstack and dualqueue to that of Treiber's lock-free stack [13], the M&S lock-free queue [9], and four lock-based alternatives. With Treiber's stack and the M&S queue we embed the calls to `pop` and `dequeue`, respectively, in a loop that repeats until the operations succeed. Two lock-based alternatives, the "locked stack" and the "locked queue" employ similar loops. The remaining two alternatives are lock-based dual data structures. Like the nonblocking dualstack and dualqueue, the "dual locked stack" and "dual locked queue" can contain either data or requests. All updates, however, are protected by a test-and-set lock.

Our experimental platform is a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2Ghz UltraSPARC III processors. Our benchmark creates $n+1$ threads for an n thread test. Thread 0 executes as follows:

```
while time has not expired
   for i = 1 to 3
      insert -1 into data structure
   repeat
      pause for about 50μs
   until data structure is empty
   pause for about 50μs
```

Other threads all run the following:

```
while time has not expired
   remove val from data structure
   if val == -1
      for i = 1 to 32
         insert i into data structure
   pause for about 0.5μs
```

These conventions arrange for a series of "rounds" in which the data structure alternates between being full of requests and being full of data. Three threads, chosen more

or less at random, prime the structure for the next round, and then join their peers in emptying it. We ran each test for two seconds, and report the minimum per-operation run time across five trials. Spot checks of longer runs revealed no anomalies. Choosing the minimum effectively discards the effects of periodic execution by kernel daemons.

Code for the various algorithms was written in C (with embedded assembly for CAS), and was compiled with `gcc` version 3.3.2 and the `-O3` level of optimization. We use the fast local memory allocator from our 2002 *PODC* paper [12].

Stack results appear in Figure 1. For both lock-based and lock-free algorithms, dualism yields a significant performance improvement: at 14 worker threads the dual locked stack is about 9% faster than (takes 93% as much time as) the locked stack that retries failed `pop` calls repeatedly; the nonblocking dualstack is about 20% faster than its non-dual counterpart. In each case the lock-based stack is faster than the corresponding lock-free stack due, we believe, to reduced contention for the top-of-stack pointer.

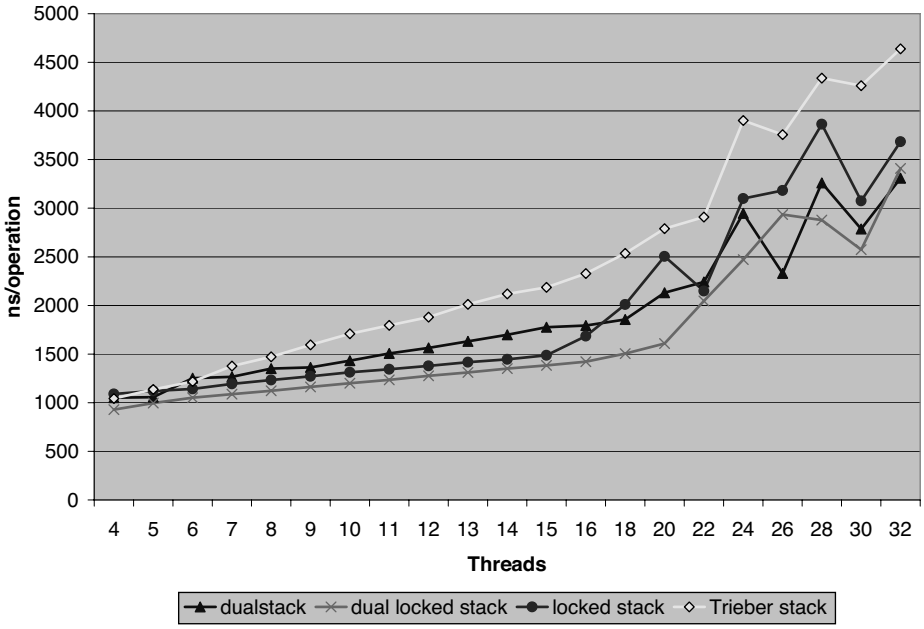

Fig. 1. Benchmark time per operation for stack algorithms.

Queue results appear in Figure 2. Here dualism again yields significant improvements: at 14 worker threads the dual locked queue is about 14% faster than the locked queue that retries failed `dequeue` calls repeatedly; the nonblocking dualqueue is more than 40% faster than its non-dual counterpart. Unlike the stacks, the nonblocking dualqueue outperforms the dual locked queue by a significant margin; we attribute this difference to the potential concurrency between enqueues and dequeues. The M&S queue is slightly faster than the locked queue at low thread counts, slightly slower for

12–15 threads, and significantly faster once the number of threads exceeds the number of processors, and the lock-based algorithm begins to suffer from preemption in critical sections. Performance of the nonblocking dualqueue is almost flat out to 16 threads (the size of the machine), and reasonable well beyond that, despite an extremely high level of contention in our benchmark; we can recommend this algorithm without reservation on any cache-coherent machine.

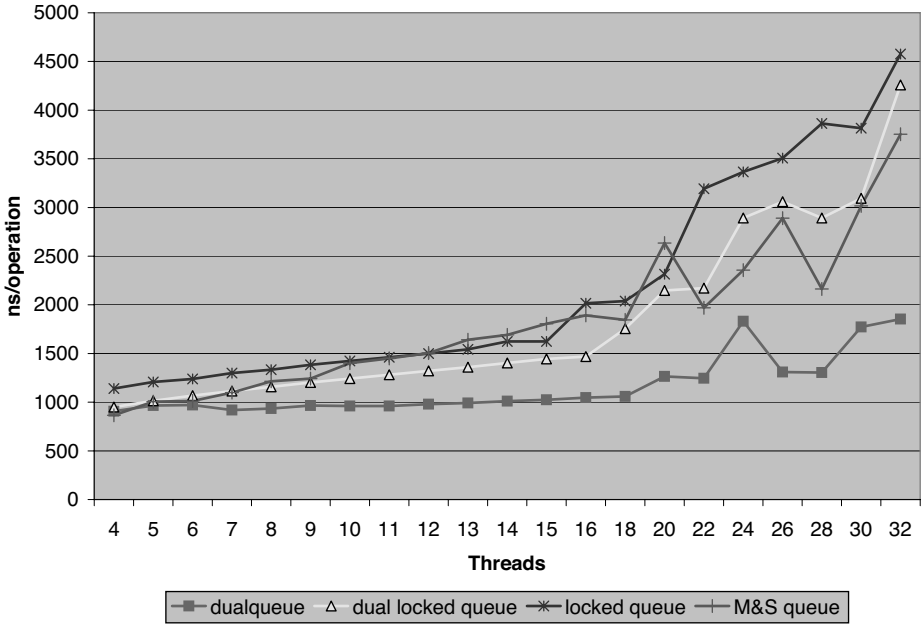

Fig. 2. Benchmark time per operation for queue algorithms.

5 Conclusions

Linearizability is central to the study of concurrent data structures. It has historically been limited by its restriction to methods that are total. We have shown how to encompass partial methods by introducing a *pair* of linearization points, one for the registration of a request and the other for its later fulfillment. By reasoning separately about a request, its successful follow-up, and the period in-between, we obtain meaningful definitions of wait-free, lock-free, and obstruction-free implementations of concurrent objects with condition synchronization.

We have presented concrete lock-free implementations of a *dualstack* and a *dualqueue*. Performance results on a commercial multiprocessor suggest that dualism can yield significant performance gains over naive retry on failure. The dualqueue, in par-

ticular, appears to be an eminently useful algorithm, outperforming the M&S queue in our experiments by almost a factor of two for large thread counts.

Nonblocking dual data structures could undoubtedly be developed for double-ended queues, priority queues, sets, dictionaries, and other abstractions. Each of these may in turn have variants that embody different policies as to which of several pending requests to fulfill when a matching operation makes a precondition true. One could imagine, for example, a stack that grants pending requests in FIFO order, or (conceivably) a queue that grants them in LIFO order. More plausibly, one could imagine an arbitrary system of thread priorities, in which a matching operation fulfills the highest priority pending request.

Further useful structures may be obtained by altering behavior between a request and its subsequent successful follow-up. As noted in Section 2.3, one could deschedule waiting threads, thereby effectively incorporating scheduler-based condition synchronization into nonblocking data structures. For real-time or database systems, one might combine dualism with timeout, allowing a spinning thread to remove its request from the structure if it waits "too long".

Acknowledgments

We are grateful to the anonymous referees for several helpful suggestions, and in particular to referee 4, who suggested that requests and follow-ups be full-fledged operations, thereby significantly simplifying the description of progress conditions between initial and final linearization points.

References

[1] BBN Laboratories. Butterfly Parallel Processor Overview. BBN Report #6148, Version 1, Cambridge, MA, March 1986.

[2] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[3] 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, July 1990.

[4] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[5] *System/370 Principles of Operation*. IBM Corporation, 1983.

[6] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[9] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[10] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[11] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proc of the Ninth International Symposium on High Performance Computer Architecture*, pages 241–252, Anaheim, CA, February 2003.

[12] M. L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 31–40, Monterey, CA, July 2002.

[13] R. K. Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.