MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

A Method for Implementing Lock-Free Shared Data Structures

G. Barnes

MPI–I–94–120 April 1994

Im Stadtwald
66123 Saarbrücken
Germany

A Method for Implementing Lock-Free Shared Data Structures

G. Barnes

MPI–I–94–120 April 1994

A Method for Implementing Lock-Free Shared Data Structures

(Extended Abstract)

Greg Barnes

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
barnes@mpi-sb.mpg.de

February 7, 1994

Abstract

We are interested in implementing data structures on shared memory multiprocessors. A natural model for these machines is an asynchronous parallel machine, in which the processors are subject to arbitrary delays. On such machines, it is desirable for algorithms to be *lock-free*, that is, they must allow concurrent access to data without using mutual exclusion. Efficient lock-free implementations are known for some specific data structures, but these algorithms do not generalize well to other structures. For most data structures, the only previously known lock-free algorithm is due to Herlihy [12]. Herlihy presents a simple methodology to create a lock-free implementation of a general data structure, but his approach can be very expensive.

We present a technique that provides the semantics of exclusive access to data without using mutual exclusion. Using this technique, we devise the *caching method*, a general method of implementing lock-free data structures that is provably better than Herlihy's methodology for many well-known data structures. The cost of one operation using the caching method is proportional to $T \log T$, where T is the sequential cost of the operation. Under Herlihy's methodology, the cost is proportional to $T + C$, where C is the time needed to make a logical copy of the data structure. For many data structures, such as arrays and *well connected* pointer-based structures (e.g., a doubly linked list), the best known value for C is proportional to the size of the structure, making the copying time much larger than the sequential cost of an operation. The new method can also allow *concurrent updates* to the data structure; Herlihy's methodology cannot. A correct lock-free implementation can be derived from a correct sequential implementation in a straightforward manner using this method. The method is also flexible; a programmer can change many of the details of the default implementation to optimize for a particular pattern of data structure use.

1 Introduction

We are interested in designing efficient data structures and algorithms for shared memory multiprocessors. Processors on these machines may execute instructions at a varying rate (due to cache behavior, for example), and are subject to long delays (e.g. when swapped out by the scheduler, or after a page fault). Programs are executed by a collection of *threads*, which are time-shared among the processors. There may be more threads than processors, so the user can view a program as running on an arbitrarily large collection of processors subject to arbitrary delays. A natural model to capture this behavior is the asynchronous parallel machine, where the processors can suffer delays of any length at any time. On such a model, concurrent access using mutual exclusion is undesirable; a thread that holds the exclusive access to some data can be delayed indefinitely, forcing other active threads to wait uselessly. This paper presents a *lock-free* technique (that is, a technique that does not use mutual exclusion) that provides the semantics of exclusive access to data. Immediately, this allows us to convert many existing concurrent algorithms based on mutual exclusion into lock-free algorithms, and use existing mutual exclusion strategies in lock-free algorithms. Using this technique and some common mutual exclusion strategies, we devise the *caching method*, a general method for implementing lock-free data structures.

Efficient lock-free algorithms are known for some specific data structures, but such algorithms are not easy to design or reason about, and they do not generalize well to other data structures. Herlihy [12] presents a simple methodology to create a lock-free version of a *general* data structure. Any sequential data structure implementation that follows certain conventions can automatically be transformed into a lock-free version using this methodology. Unfortunately, the methodology can be very expensive, often requiring the entire data structure to be copied for each operation. The methodology also does not allow *concurrent updates* to the structure. Only one thread that is trying to update the structure is actually doing useful work at any particular time, so it can never achieve better throughput than the original sequential implementation.

Ideally, a method to create lock-free data structures should work on all data structures, it should be simple to use and reason about, and it should be as efficient as possible, including allowing concurrent updates. Realistically, any general method is unlikely to work well for all data structures. Even if it can be shown that a method is theoretically efficient, in practice a shared memory machine can behave quite differently from the worst case performance of a theoretical model. In addition to the three characteristics above, then, a general method should be flexible, so that clever and ambitious implementors can improve its performance for a given data structure or a given pattern of machine behavior, while still maintaining guarantees on its correctness and worst-case performance.

The caching method is close to this goal. It works for all data structures and achieves a nearly optimal asymptotic work bound, proportional to $T \log T$ per operation, where T is the sequential cost of the operation. For many data structures, this is a large improvement over Herlihy's methodology. A correct lock-free implementation can be derived from a correct sequential implementation in a straightforward manner using this method. For some data structures, this straightforward lock-free implementation will allow concurrent updates of the data structure. For other data structures, the implementor can often create a lock-free implementation that allows concurrent updates with only a little extra work. Finally, the method is flexible, so the implementor is free to change many details of the straightforward implementation to optimize for a specific pattern of data structure use.

1.1 Background and Previous Work

Lock-free data structures help us to design *non-blocking* and *wait-free* parallel algorithms. An asynchronous algorithm is *non-blocking* if it always guarantees at least one thread will complete an operation in a finite number of steps. An algorithm is *wait-free* if it guarantees *all* threads will complete their work in a finite number of steps. The caching method transforms a correct sequential implementation into a non-blocking parallel implementation. In Section 6 we discuss ways to make an implementation wait-free.

Early work on lock-free objects focused on proving the power of various synchronization primitives. Herlihy [13] unifies much of this work by showing the existence of *universal* primitives, such as Compare&Swap, which can be used to implement any wait-free object. Using Load_Linked and Store_Conditional, a universal pair of primitives similar to Compare&Swap, Herlihy [12] describes a methodology for converting synchronous implementations of data structure algorithms to non-blocking and wait-free implementations. Alemany and Felten [1] present techniques for improving the performance of Herlihy's protocol in practice. Herlihy and Moss [14] introduce transactional memory, an architectural approach to supporting lock-free data structures. Efficient lock-free implementations of some specific data structures are known. Lamport [19] and Herlihy and Wing [15] give non-blocking algorithms for queues. Lanin and Shasha give a non-blocking set manipulation algorithm [20]. Anderson and Woll [4] design non-blocking algorithms for Union-Find sets, and Anderson analyzes non-blocking algorithms for the related problem of list compression [3]. Massalin uses a lock-free implementation of stacks, queues, and linked lists in his Synthesis operating system kernel [22].

Many different versions of the asynchronous parallel random access machine, or APRAM, have been proposed (including [8, 9, 10, 24]), most with differing notions of runtime. We measure the performance of our algorithm using *work*, the same measure used in a series of papers on fault-tolerant PRAMs [16, 17, 21]. The work done by an algorithm is the

total number of steps taken by all threads. Work is a generalization of the time-processor product of the PRAM. For a given pattern of delays, the minimum work algorithm will yield the minimum time algorithm on an APRAM, so work is a measure of the efficiency of an asynchronous algorithm. We measure the worst case performance of algorithms against a strong adversary, previously used by Anderson [2], Anderson and Woll [4], and Buss and Ragde [7]. The adversary chooses the operations performed on the data structure, and the order in which threads execute instructions.

Both Herlihy's methodology and our method use the Load_Linked and Store_Conditional synchronization primitives. Load_Linked acts like a load instruction. Store_Conditional is similar to a store instruction, but it succeeds only if no other thread has written the variable since the thread read the variable using Load_Linked. Store_Conditional returns a boolean value indicating whether the write succeeded or failed. Load_Linked and Store_Conditional can be efficiently implemented given a cache-coherent architecture, and are supported in the MIPS-II architecture [23]. Given these primitives, it is not difficult to construct a lock-free implementation of a one-word data object, such as a counter. Larger data objects can also be atomically updated by using Load_Linked and Store_Conditional on a pointer to the object. This strategy is the basis for Herlihy's methodology.

1.2 Herlihy's Methodology and the Copying Algorithm

The main difficulty with concurrent updates to a data structure is that multiple threads may want to change the same portion of the data structure at the same time. Most implementations of lock-free data structures solve this problem either by proving that, for a specific set of operations, two threads working on the same portion of the data structure do not substantially interfere with each other, or by preventing such interaction. Herlihy's methodology uses the latter strategy. In his basic methodology, all threads change the data structure by changing a pointer to the structure. To perform an operation, a thread uses Load_Linked to read the pointer to the data structure, and makes a private copy of the structure. It then changes its private copy, and tries to replace the old pointer with a pointer to this private copy, using Store_Conditional to test whether the pointer has been changed in the interim. If it has changed, the replacement fails and the thread must start over.

Since copying the entire object can be time-consuming, Herlihy suggests that the programmer specify ways to reduce the amount of copying for large objects. For example, if we wish to change the first element in a singly linked list, only the first element needs to be copied. The new element can use the old element's *next* pointer to logically copy the rest of the list without actually doing the work. It is not clear that this strategy can be effective for all data structures. For random access data structures, such as arrays, copying

only a portion of the data structure destroys the random access property. For pointer-based data structures, an algorithm should make a copy of all elements in the structure that are changed by an operation. But if an element e is copied, all elements that point to e change and must be copied as well, which means that all elements that point to elements that point to e must be copied, and so on. So, for example, if the kth element in a linked list is to be changed, the first k elements should be copied.

Using these observations, we define the *copying algorithm* for data structures, the best general algorithm we are aware of that can be derived from Herlihy's methodology. For a random access data structure, the copying algorithm copies the entire structure. For pointer-based data structures, the algorithm copies the elements that are changed, and, recursively, all elements that point to elements that are copied. The copying algorithm performs work proportional to $T + C$ per operation, where T is, as before, the sequential cost of the operation, and C is the amount of copying work needed for the operation. This is much more than the cost of the caching method for many data structures, such as arrays, or pointer based structures that are *well connected* (that is, structures where, for any element, e, there are many elements, e_i, such that there is a path of pointers from e_i to e). For these structures, the copying algorithm must copy a large portion of the structure on each operation.

The remainder of the paper is organized as follows. We begin in Section 2 with a short explanation of the *cooperative technique*, our technique for lock-free exclusive access. Section 3 outlines the caching method. In Section 4 we discuss the proofs of correctness of the technique and of the caching method. In Section 5 we present some performance bounds for the caching method, and sketch a proof of the bounds. Section 6 concludes with some notes and suggestions for future work.

2 The Cooperative Technique

The cooperative technique uses a different approach to handle thread interference, previously used by Barnes [6]. We observe that multiple threads can simultaneously work on the same data structure if all threads write down exactly what they are doing. If a thread t_1 wishes to change some part of the data structure, it first checks whether another thread t_2 was working there first. If so, t_1 reads t_2's information, and cooperates to complete t_2's work.

This idea can be used to guarantee the same semantics as standard mutual exclusion primitives, such as locks. Let an *opdesc* be a variable that describes an operation that a thread wishes to perform on some sets of shared data. Suppose we have a set of locks that provide exclusive access to disjoint sets of shared data, $D_1 \ldots D_N$. We can replace each set

D_i and its lock with a pointer to a record. The record holds the shared data, D_i, along with an *opptr* field which, if not empty, contains the address of an *opdesc*. We say a thread *claims* a cell D_i (that is, acquires the equivalent of exclusive access to D_i) by installing its *opdesc* in the *opptr* field of D_i's record. To claim D_i, a thread first reads the pointer to D_i's record using Load_Linked, and then reads the record. If the *opptr* field is not empty, it cooperates to complete the associated operation, and then begins again. (Note that the thread only needs to cooperate until the operation releases its claim to D_i.) If the *opptr* field is empty, the thread creates a new record whose *opptr* field points to its own *opdesc*, and tries to replace the old pointer with a pointer to this new record, using Store_Conditional. If the Store_Conditional fails, it begins again. Otherwise it has the equivalent of exclusive access to D_i, since it can be assured that no other thread will interfere with its work on D_i until it releases its claim on D_i.

As an immediate consequence, we can convert any algorithm that provide exclusive access to data using mutual exclusion into a lock-free algorithm. The main question is how to allow threads to cooperate. One elementary scheme is to write the operation as a sort of program, and have the threads interpret the program, using a *state* record to hold a "program counter" and some auxiliary "memory". A pseudocode version of this scheme is given in Figure 1.

In this scheme, a thread repeatedly reads the *state*, tries to execute the next instruction, and then tries to update the *state*. If an update of the *state* fails, the thread has been delayed, but this does not matter, since the thread rereads the *state* in the next step. If an update of shared data fails, the thread reads the shared data, and checks if the data matches the values it was trying to write. If so, it assumes another thread succeeded in updating the shared data, and tries to update the *state* as if its own update had succeeded. If not, it assumes no other thread succeeded in updating the shared data, and tries to update the *state* accordingly. In general, the process in Figure 1 can be very slow, because the threads are acting as interpreters, and the size of the *state* can be large. In practice, we do not expect the process to be so slow, since critical sections are usually designed to be short and simple.

3 The Caching Method

The caching method uses the cooperative technique along with standard ideas from the study of concurrent algorithms to generate a lock-free caching algorithm for any data structure. Assume the data structure is divided into small *cells*, corresponding to the disjoint sets of shared data above. The basic strategy of a thread is to claim all the cells it wishes to change. Once it has successfully claimed all cells, the thread has effectively completed

```
procedure Cooperate (stateptr: pointer to state);
  Load_Linked(stateptr), and determine which instruction should be executed next.
  while the operation is not complete do begin
    write_status := SUCCESS;
    if the next instruction reads or writes shared data then
      Use Load_Linked to read or Store_Conditional to write the pointer to the data.
      if the instruction was a write, and the Store_Conditional failed then
        Load_Linked the pointer to the data.
        if the data does not match the values we were trying to write then
          write_status := FAILURE;
    nextptr := pointer to an updated version of state (based, if appropriate, on write_status)
    Store_Conditional(stateptr, nextptr);
    Load_Linked(stateptr), and determine which instruction should be executed next.
  end;
end Cooperate.
```

Figure 1: An elementary cooperation scheme

its operation, since no other thread can change these cells until this thread's operation is finished. Just as we must guard against deadlock when using mutual exclusion, we must guard against *livelock* when using this strategy. For example, if two threads, t_1 and t_2, both want to change cells c_1 and c_2, t_1 could claim c_1, t_2 could claim c_2, and then both would have to complete the other's operation before completing their own. We avoid livelock using a standard deadlock avoidance technique: assign each cell a unique key, and require that a thread claim its cells in increasing order based on their keys. Of course, to claim cells in increasing order based on their keys, a thread must know all the cells that it needs to claim before it claims its first cell. For many data structure operations, the thread does not know exactly which cells will be changed when the operation begins. We solve this problem by having the threads first carry out their operation on a private cached version of the cells.

In the caching method, a thread t performs a lock-free operation in four stages.

1. Perform the operation as usual, but on a cached version of the structure. Read or write cells in t's private cache only. If the cache doesn't contain a cell, use Load_Linked to read the cell from the structure, and make a private copy.

2. Validate the operation. For each cell in t's cache, Load_Linked the corresponding entry

in the data structure and make sure the cell has not changed since it was initially copied. If any cell has changed, abort and start over at the first stage.

3. For each cell in t's cache, in ascending order of their keys, try to claim the corresponding cell in the data structure. If any cell has changed since the cell was read by t, Load_Linked the cell, and then abort the operation: release any claims already made, and start over at the first stage.

4. Change the cells and release t's claims.

Every time a thread performs a Load_Linked on the pointer to a cell's record, it must follow the cooperative technique: check whether the *opptr* field of the corresponding record is empty, and if not, help the appropriate thread complete its operation.

A complete description of the details of the caching method is deferred to the full paper. We present some of the more important points below.

Some of the details of the caching method presented above are not necessary for correctness, but help prove better performance bounds. In particular, the second stage (validation) allows us to prove better bounds by ensuring that a thread only claims a cell if it saw a consistent version of the data (see the proof of Theorem 5.2, below). It can be omitted and the method will still be correct. Also, note that whenever a thread's operation is aborted (in the second or third stage), it first performs a Load_Linked operation. Because the threads follow the cooperative technique, this helps maintain an important invariant—no thread is aborted more often than the number of operations successfully completed. This invariant is also used to prove upper bounds on the amount of work per operation.

The division of the data structure into cells can be arbitrary, as long as the cells are of constant size, and structured in some logical way (e.g. as continuous locations in memory, or in a connected pointer structure). This insures that each read or write takes only constant time. Most data structures have a natural partition. For example, each entry in a linked list can be a cell. Similarly, the keys assigned to each cell can be arbitrary. For example, the key could be the address of the pointer to the cell. In Section 6, we mention a variant of the caching method where it is more useful if the ordering of the keys corresponds to the underlying structure of the data.

The cache used in each operation must hold three values for each cell: the `address` of the pointer to the cell, the `old` pointer the thread, t, originally read in this address, and a `new` pointer to t's version of the cell. Let s be the number of cells in the cache. Using a balanced tree sorted on the keys of the cells, each cache read or write takes $O(\log s)$ steps, and the tree can be converted in $O(s)$ steps into a linked list of (`address`, `old`, `new`) records sorted on the keys of the cells. This linked list makes it simple for threads to

cooperate to complete the operation. After the first cell is claimed, the thread needs only to execute a series of Store_Conditional operations. We can use the elementary scheme in Figure 1, where the state is merely some pointers into the linked list and a status variable to indicate whether the operation is claiming cells, releasing claims, etc. If the operation is aborted, any cooperating threads use Load_Linked to read the cell that caused the abort (and cooperate on the indicated operation, if any), and then *return to their own work*. The possible cooperation resulting from reading the cell that caused the abort helps maintain the invariant mentioned above: no thread is aborted more often than the number of operations successfully completed. Having the threads return to their own work keeps the cooperation process simple, since threads never cooperate on the first or second stage of an operation.

4 Correctness

The following invariants are used to prove the cooperative technique and the caching method correct.

1. When using the cooperative technique, if the *opptr* field in the record for a set of data is not empty, the pointer to that record is changed only by a thread cooperating to complete the associated operation.

2. The elementary scheme of Figure 1 (p7) allows multiple threads to properly execute the indicated operation.

3. Let G_τ be a directed graph associated with a particular time, τ, during the execution of an algorithm generated using the caching method. G_τ has one node per thread, and an edge from the node representing thread t_1 to the node representing thread t_2 if and only if t_1 is cooperating to complete t_2's operation at time τ. Then for all τ, G_τ is acyclic.

4. In the caching method, no thread uses the results of an incomplete operation to perform its operation.

Invariant 3 is true because the cells are claimed in ascending order based on their keys. This invariant can be used to show that no livelock occurs, and therefore some thread is always making progress, i.e. the implementation is non-blocking.

The standard notion of correctness for asynchronous parallel algorithms is to assume that the atomic instructions of the threads are interleaved in some linear order. The algorithm is correct if it behaves properly for all such interleavings [15, 18]. Let Q be the set of operations

performed on a data structure. For a structure's implementation to behave correctly, there must exist some ordering, Π_Q, of the operations in Q such that the results of operations that extract information from the data structure correspond to the results obtained by a uniprocessor algorithm given the sequence of operations, Π_Q. Invariant 4, which follows from the first two invariants, allows us to prove an even stronger guarantee for the caching method.

5 Performance

Consider the *caching algorithm* for a particular data structure, derived using the caching method. For simplicity, assume the data structure is always the same size. For an operation *op*, let T_{op} be the number of steps needed to execute *op* in the sequential implementation of the data structure, let C_{op} be the number of steps used by the copying algorithm to make a logical copy of the data structure when executing *op*, and let s_{op} be the number of distinct cells read or written by *op*. Let T_{ave} and s_{ave} be the average values of T_{op} and s_{op}, and let T_{max}, s_{max}, and C_{max} be the maximum values of T_{op} s_{op}, and C_{op}, over all operations. Note that $s_{op} \leq T_{op}$, and therefore $s_{ave} \leq T_{ave}$ and $s_{max} \leq T_{max}$. Let p be the number of threads executing in the parallel implementation. Recall that we are using a strong adversary that picks the operations on the structure and the interleaving of instructions.

Proposition 5.1 *The copying algorithm performs $O(Kp(C_{max} + T_{max}))$ work to complete K operations on the data structure. The adversary can force the copying algorithm to perform $\Omega(Kp(C_{max} + T_{max}))$ work to complete K operations.*

Both bounds are based on the observation that if p threads perform their work and then simultaneously try to change the pointer to the data structure, exactly one will succeed.

The bounds for the caching algorithm depend on one further parameter. Suppose there is only one thread, and let the *first cell* of an operation *op* be the first cell the thread would claim if it were to execute *op* next. Let α_D be the number of distinct first cells for the data structure D over all operations, and let α be the maximum value of α_D for all possible data structures, D, of the given size. So, for example, in an implementation of a queue, a dequeue operation might always claim the head first, while an enqueue claims the tail or, if the queue is empty, the head. For the queue, then, α is two, since for any queue, at most two of the cells can be the first cell claimed by the next operation.

Theorem 5.2 *The caching algorithm performs $O(Kp(\alpha \cdot s_{max} + T_{max} \log s_{max}))$ work to complete K operations. The adversary can force the caching algorithm to perform $\Omega(Kp T_{ave} \log s_{ave})$ work to complete K operations.*

Proof:[Sketch] The lower bound is achieved when p threads perform the same operation, complete their first stages simultaneously, and then simultaneously try to claim the same cell.

The upper bound is based on amortizing the cost of operations. Divide an operation into a series of *attempts*, where the last attempt succeeds but the previous attempts are all aborted. By the invariant mentioned in Section 3, for each aborted attempt of a thread t_1, there is at least one successful attempt by another thread. Most of the work done by t_1 during an aborted attempt is charged to this successful attempt. The only work not charged is work t_1 performed on another thread t_2's aborted attempt. Note that t_2 must have successfully claimed a cell during the attempt in question. Intuitively, though, for each successful attempt, there can be at most $\alpha - 1$ other threads that claim a cell (this is true because the validation stage guarantees that these threads saw a consistent version of the data). If we charge the work spent on the third stage of these aborted attempts to the successful attempt, we get the bound in the theorem. □

6 Notes and Future Work

As noted before, the bounds for the caching algorithm are much better than the bounds for the copying algorithm, regardless of the distribution of operations, for many common data structures, including array-based structures and well connected pointer-based structures such as doubly-linked lists. For many other data structures, the caching algorithm will have better bounds under certain common distributions of operations. For example, given a queue implemented as a linked list, the copying algorithm will always have to copy the entire structure either on an enqueue or a dequeue. If we assume there are as many dequeues as enqueues, the copying algorithm must copy the entire structure on half of the operations, while the caching algorithm need only perform constant work per operation.

Still, the bounds for the caching algorithm are not as good as one might hope. The p term in the lower bound says that the adversary can always force the algorithm to run as slowly as a sequential version. This is not surprising given such a strong adversary. For most applications, the adversary is too strong. In particular, the assumption that multiple threads will perform the same operation simultaneously is not always believable. One open problem is to analyze the performance of the caching algorithm for specific data structures using more realistic adversaries. We would like to devise techniques to analyze weaker adversaries that correspond to common patterns of machine behavior or data structure use. For example, Anderson and Woll [4] consider the case where there is always a large pool of operations to perform, and a thread chooses the next operation at random from the pool. If the adversary has no knowledge of the random bits, such an assumption may allow us to

prove better upper bounds.

The $\langle \alpha \cdot s_{max} \rangle$ and $\langle \log s_{max} \rangle$ terms in the upper bound are also troublesome. These terms are constant for some data structures, but not all. We can artificially make α constant for any data structure, and thus change the upper bound to $O(KpT_{max} \log s_{max})$, by forcing all threads to claim a sentinel variable before they claim any other cells. With some extra overhead, the sentinel can be coupled with an idea similar to Herlihy's [12, Section 4.3] to make any implementation wait-free. Instead of grabbing the sentinel using Store_Conditional, a protocol that prevents starvation is used to decide who will claim the sentinel (and hence, perform their operation next).

Note, however, that the α parameter is a crude measure of the parallelism of the data structure, the number of concurrent updates that can be performed at once. Lowering the value of α by implementing a sentinel has the undesirable side effect of disallowing concurrent updates. We would like a general method that yields lock-free data structures which achieve good performance and allow concurrent updates (even for data structures with a low value of α), and which can be modified to be wait-free if desired. We have devised a modification of the caching method that seems promising for some structures.

Suppose we can divide an operation into suboperations that claim only a few cells at once. Furthermore, suppose a suboperation always claims the cell that will have the lowest key in the next suboperation. For example, consider a heap where a parent cell always has a lower key than the cells of its children. Note that $\alpha = 1$ in the standard heap implementation with delete_min and insert operations, because every operation must claim the root node. Divide a delete_min operation into a swap suboperation between the root and the highest numbered leaf, followed by a series of swap suboperations down the tree. Then the delete_min operation meets the desired criteria, since each swap needs to claim either the root and a leaf, or a node and its two children, and one of these nodes will be the cell with the lowest key in the next swap suboperation. So, if a thread is executing a delete_min operation, it can release its claim on the root once the first few swaps are completed, and another thread can begin another delete_min operation. If similar guarantees can be made for all operations, we may be able to devise an algorithm that allows concurrent updates *and* eliminates the $\langle \alpha \cdot s_{max} \rangle$ and $\langle \log s_{max} \rangle$ terms in the upper bound of Theorem 5.2. Such a scheme can sometimes also be coupled with a sentinel to give a wait-free implementation. Details of this method are deferred to the full paper. A complete version of the heap implementation appears in [6].

One can use the caching method exactly as given to transform the sequential implementation of a data structure into a non-blocking concurrent implementation, but many of the details of the method, such as the thread cooperation scheme, can be changed without affecting the correctness of an implementation. One open problem is to devise variants of

the caching method that take advantage of particular patterns of data structure use. For example, if one believes threads will often cooperate, one may want a thread cooperation scheme with a better division of the work, instead of having every thread work on the same task. On the other hand, if one believes cooperation will be rare, one would want to optimize the procedure so that the original thread works quickly, at the expense of the rare extra threads.

Finally, the cooperative technique suggests that other ideas from the study of mutual exclusion can be used in lock-free algorithms. For example, the caching method avoids livelock by using the well-known deadlock avoidance scheme of ordering the resources. There are other ways to avoid deadlock, and ways to detect and break deadlock. Can these methods be used to create different lock-free algorithms with good performance?

References

[1] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, Aug. 1992.

[2] R. J. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 95–102, Crete, Greece, June 1990.

[3] R. J. Anderson. Primitives for asynchronous list compression. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 199–208, San Diego, CA, June 1992.

[4] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. Technical Report 91-04-05, University of Washington, 1991. See also [5].

[5] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 370–380, New Orleans, LA, May 1991.

[6] G. Barnes. Wait-free algorithms for heaps. University of Washington, Preprint, 1992.

[7] J. F. Buss and P. Ragde. Certified write-all on a strongly asynchronous PRAM. Preliminary Report, 1990.

[8] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, Santa Fe, NM, June 1989.

[9] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 85–94, Crete, Greece, July 1990.

[10] P. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. In *Proceedings of the Second Annual ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 197–206, Mar. 1990.

[12] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Lab, Oct. 1991. See also [11].

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

[14] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL 92/07, DEC Cambridge Research Lab, Dec. 1992.

[15] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, Jan. 1987.

[16] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 211–222, 1989.

[17] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 138–148, Baltimore, MD, May 1990.

[18] L. Lamport. 'Sometime' is sometimes 'not never'. Technical report, S.R.I. International, Menlo Park, CA, Jan. 1979.

[19] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.

[20] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Database Systems*, pages 211–220, Mar. 1988.

[21] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *31st Annual Symposium on Foundations of Computer Science*, pages 590–599, St. Louis, MO, Oct. 1990. IEEE.

[22] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[23] MIPS Computer Company. *The MIPS RISC architecture.*

[24] N. Nishimura. Asynchronous shared memory parallel computation. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 76–84, Crete, Greece, July 1990.