1

Concurrent Data Structures

Mark Moir and Nir Shavit
Sun Microsystems Laboratories

The proliferation of commercial shared-memory multiprocessor machines has brought about significant changes in the art of concurrent programming. Given current trends towards low-cost chip multithreading (CMT), such machines are bound to become ever more widespread.

Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. The efficiency of these data structures is crucial to performance, yet designing effective data structures for multiprocessor machines is an art currently mastered by few. By most accounts, concurrent data structures are far more difficult to design than sequential ones because threads executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome. This requires designers to modify the way they think about computation, to understand new design methodologies, and to adopt a new collection of programming tools. Furthermore, new challenges arise in designing *scalable* concurrent data structures that continue to perform well as machines that execute more and more concurrent threads become available. This chapter provides an overview of the challenges involved in designing concurrent data structures, and a summary of relevant work for some important data structure classes. Our summary is by no means comprehensive; instead, we have chosen popular data structures that illustrate key design issues, and hope that we have provided sufficient background and intuition to allow the interested reader to approach the literature we do not survey.

1.1 Designing Concurrent Data Structures

Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts.

0-8493-8597-0/01/$0.00+$1.50

```
oldval = X;
X = oldval + 1;
return oldval;
```

```
acquire(Lock);
oldval = X;
X = oldval + 1;
release(Lock);
return oldval;
```

FIGURE 1.1: Code fragments for sequential and lock-based `fetch-and-inc` operations.

The primary source of this additional difficulty is concurrency: Because threads are executed concurrently on different processors, and are subject to operating system scheduling decisions, page faults, interrupts, etc., we must think of the computation as completely asynchronous, so that the steps of different threads can be interleaved arbitrarily. This significantly complicates the task of designing correct concurrent data structures.

Designing concurrent data structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. On today's machines, the layout of processors and memory, the layout of data in memory, the communication load on the various elements of the multiprocessor architecture all influence performance. Furthermore, the issues of correctness and performance are closely tied to each other: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.

The following example illustrates various features of multiprocessors that affect concurrent data structure design. Suppose we wish to implement a *shared counter* data structure that supports a `fetch-and-inc` operation that adds one to the counter and returns the value of the counter immediately before the increment. A trivial sequential implementation of the `fetch-and-inc` operation contains code like that shown on the left in Figure 1.1:[1]

If we allow concurrent invocations of the `fetch-and-inc` operation by multiple threads, this implementation does not behave correctly. To see why, observe that most compilers will translate this source code into machine instructions that load `X` into a register, then add one to that register, then store that register back to `X`. Suppose that the counter is initially 0, and two `fetch-and-inc` operations execute on different processors concurrently. Then there is a risk that both operations read 0 from `X`, and therefore both store back 1 and return 0. This is clearly incorrect: one of the operations should return 1.

The incorrect behavior described above results from a "bad" interleaving of the steps of the two `fetch-and-inc` operations. A natural and common way to prevent such interleavings is to use a mutual exclusion lock (also known as a *mutex* or a *lock*). A lock is a construct that, at any point in time, is unowned or is owned by a single thread. If a thread t_1 wishes to acquire ownership of a lock that is already owned by another thread t_2, then t_1 must wait until t_2 releases ownership of the lock.

We can obtain a correct sequential implementation of the `fetch-and-inc` operation by using a lock as shown on the right in Figure 1.1. With this arrangement, we prevent the bad interleavings by preventing *all* interleavings. While it is easy to achieve a correct shared counter this way, this simplicity comes at a price: Locking introduces a host of problems related to both performance and software engineering.

[1] Throughout our examples, we ignore the fact that, in reality, integers are represented by a fixed number of bits, and will therefore eventually "wrap around" to zero.

1.1.1 Performance

The *speedup* of an application when run on P processors is the ratio of its execution time on a single processor to its execution time on P processors. It is a measure of how effectively the application is utilizing the machine it is running on. Ideally, we want *linear speedup*: we would like to achieve a speedup of P when using P processors. Data structures whose speedup grows with P are called *scalable*. In designing scalable data structures we must take care: naive approaches to synchronization can severely undermine scalability.

Returning to the lock-based counter, observe that the lock introduces a *sequential bottleneck*: at any point in time, at most one `fetch-and-inc` operation is doing useful work, i.e. incrementing the variable `X`. Such sequential bottlenecks can have a surprising effect on the speedup one can achieve. The effect of the sequentially executed parts of the code on performance is illustrated by a simple formula based on Amdahl's law [110]. Let b be the fraction of the program that is subject to a sequential bottleneck. If the program takes 1 time unit when executed on a single processor, then on a P-way multiprocessor the sequential part takes b time units, and the concurrent part takes $(1-b)/P$ time units in the best case, so the speedup S is at most $1/(b+(1-b)/P)$. This implies that if just 10% of our application is subject to a sequential bottleneck, the best possible speedup we can achieve on a 10-way machine is about 5.3: we are running the application at half of the machine's capacity. Reducing the number and length of sequentially executed code sections is thus crucial to performance. In the context of locking, this means reducing the number of locks acquired, and reducing *lock granularity*, a measure of the number of instructions executed while holding a lock.

A second problem with our simple counter implementation is that it suffers from *memory contention*: an overhead in traffic in the underlying hardware as a result of multiple threads concurrently attempting to access the same locations in memory. Contention can be appreciated only by understanding some aspects of common shared-memory multiprocessor architectures. If the lock protecting our counter is implemented in a single memory location, as many simple locks are, then in order to acquire the lock, a thread must repeatedly attempt to modify that location. On a *cache-coherent* multiprocessor[2] for example, exclusive ownership of the cache line containing the lock must be repeatedly transferred from one processor to another; this results in long waiting times for each attempt to modify the location, and is further exacerbated by the additional memory traffic associated with unsuccessful attempts to acquire the lock. In Section 1.1.7, we discuss lock implementations that are designed to avoid such problems for various types of shared memory architectures.

A third problem with our lock-based implementation is that, if the thread that currently holds the lock is delayed, then all other threads attempting to access the counter are also delayed. This phenomenon is called *blocking*, and is particularly problematic in *multiprogrammed* systems, in which there are multiple threads per processor and the operating system can preempt a thread while it holds the lock. For many data structures, this difficulty can be overcome by devising *nonblocking* algorithms in which the delay of a thread does not cause the delay of others. By definition, these algorithms cannot use locks.

Below we continue with our shared counter example, discussing blocking and nonblocking techniques separately; we introduce more issues related to performance as they arise.

[2] A *cache-coherent* multiprocessor is one in which processors have local caches that are updated by hardware in order to keep them consistent with the latest values stored.

1.1.2 Blocking Techniques

In many data structures, the undesirable effects of memory contention and sequential bottlenecks discussed above can be reduced by using a *fine-grained* locking scheme. In fine-grained locking, we use multiple locks of small granularity to protect different parts of the data structure. The goal is to allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure. This approach can also help to avoid excessive contention for individual memory locations. For some data structures, this happens naturally; for example, in a hash table, operations concerning values that hash to different buckets naturally access different parts of the data structure.

For other structures, such as the lock-based shared counter, it is less clear how contention and sequential bottlenecks can be reduced because—abstractly—all operations modify the same part of the data structure. One approach to dealing with contention is to spread operations out in time, so that each operation accesses the counter in a separate time interval from the others. One widely used technique for doing so is called *backoff* [3]. However, even with reduced contention, our lock-based shared counter still lacks parallelism, and is therefore not scalable. Fortunately, more sophisticated techniques can improve scalability.

One approach, known as a *combining tree* [36, 37, 51, 138], can help implement a scalable counter. This approach employs a binary tree with one leaf per thread. The root of the tree contains the actual counter value, and the other internal nodes of the tree are used to coordinate access to the root. The key idea is that threads climb the tree from the leaves towards the root, attempting to "combine" with other concurrent operations. Every time the operations of two threads are combined in an internal node, one of those threads—call it the loser—simply waits at that node until a return value is delivered to it. The other thread—the winner—proceeds towards the root carrying the sum of all the operations that have combined in the subtree rooted at that node; a winner thread that reaches the root of the tree adds its sum to the counter in a single addition, thereby effecting the increments of all of the combined operations. It then descends the tree distributing a return value to each waiting loser thread with which it previously combined. These return values are distributed so that the effect is as if all of the increment operations were executed one after the other at the moment the root counter was modified.

The technique losers employ while waiting for winners in the combining tree is crucial to its performance. A loser operation waits by repeatedly reading a memory location in a tree node; this is called *spinning*. An important consequence in a cache-coherent multiprocessor is that this location will reside in the local cache of the processor executing the loser operation until the winner operation reports the result. This means that the waiting loser does not generate any unnecessary memory traffic that may slow down the winner's performance. This kind of waiting is called *local spinning*, and has been shown to be crucial for scalable performance [97].

In so-called *non-uniform memory access* (NUMA) architectures, processors can access their local portions of shared memory faster than they can access the shared memory portions of other processors. In such architectures, *data layout*—the way nodes of the combining tree are placed in memory—can have a significant impact on performance. Performance can be improved by locating the leaves of the tree near the processors running the threads that own them. (We assume here that threads are statically bound to processors.)

Data layout issues also affect the design of concurrent data structures for cache-coherent multiprocessors. Recall that one of the goals of the combining tree is to reduce contention for individual memory locations in order to improve performance. However, because cache-coherent multiprocessors manage memory in cache-line-sized chunks, if two threads are accessing different memory locations that happen to fall into the same cache line, performance

suffers just as if they had been accessing the same memory location. This phenomenon is known as *false sharing*, and is a common source of perplexing performance problems.

By reducing contention for individual memory locations, reducing memory traffic by using local spinning, and allowing operations to proceed in parallel, counters implemented using combining trees scale with the number of concurrent threads much better than the single lock version does [51]. If all threads manage to repeatedly combine, then a tree of width P will allow P threads to return P values after every $O(\log P)$ operations required to ascend and descend the tree, offering a speedup of $O(P/\log P)$ (but see Section 1.1.4).

Despite the advantages of the combining tree approach, it also has several disadvantages. It requires a known bound P on the number of threads that access the counter, and it requires $O(P)$ space. While it provides better throughout under *heavy loads*, that is, when accessed by many concurrent threads, its best-case performance under low loads is poor: It must still traverse $O(\log P)$ nodes in the tree, whereas a `fetch-and-inc` operation of the single-lock-based counter completes in constant time. Moreover, if a thread fails to combine because it arrived at a node immediately after a winner left it on its way up the tree, then it must wait until the winner returns before it can continue its own ascent up the tree. The coordination among ascending winners, losers, and ascending late threads, if handled incorrectly, may lead to *deadlocks*: threads may block each other in a cyclic fashion such that none evers make progress. Avoiding deadlocks significantly complicates the task of designing correct and efficient implementations of blocking concurrent data structures.

In summary, blocking data structures can provide powerful and efficient implementations if a good balance can be struck between using enough blocking to maintain correctness, while minimizing blocking in order to allow concurrent operations to proceed in parallel.

1.1.3 Nonblocking Techniques

As discussed earlier, nonblocking implementations aim to overcome the various problems associated with the use of locks. To formalize this idea, various nonblocking progress conditions—such as wait-freedom [49, 83], lock-freedom [49], and obstruction-freedom [54]—have been considered in the literature. A *wait-free* operation is guaranteed to complete after a finite number of its own steps, regardless of the timing behavior of other operations. A *lock-free* operation guarantees that after a finite number of its own steps, *some* operation completes. An *obstruction-free* operation is guaranteed to complete within a finite number of its own steps after it stops encountering interference from other operations.

Clearly, wait-freedom is a stronger condition than lock-freedom, and lock-freedom in turn is stronger than obstruction-freedom. However, all of these conditions are strong enough to preclude the use of blocking constructs such as locks.[3] While stronger progress conditions seem desirable, implementations that make weaker guarantees are generally simpler, more efficient in the common case, and easier to design and to verify as correct. In practice, we can compensate for the weaker progress conditions by employing backoff [3] or more sophisticated contention management techniques [55].

Let us return to our shared counter. It follows easily from results of Fischer et al. [32] (extended to shared memory by Herlihy [49] and Loui and Abu-Amara [93]) that such a shared counter cannot be implemented in a lock-free (or wait-free) manner using only

[3] We use the term "nonblocking" broadly to include all progress conditions requiring that the failure or indefinite delay of a thread cannot prevent other threads from making progress, rather than as a synonym for "lock-free", as some authors prefer.

```
bool CAS(L, E, N) {
  atomically {
    if (*L == E) {
      *L = N;
      return true;
    } else
      return false;
  }
}
```

FIGURE 1.2: The semantics of the CAS operation.

`load` and `store` instructions to access memory. These results show that, in any proposed implementation, a bad interleaving of operations can cause incorrect behaviour. These bad interleavings are possible because the `load` and `store` are separate operations. This problem can be overcome by using a hardware operation that atomically combines a load and a store. Indeed, all modern multiprocessors provide such synchronization instructions, the most common of which are *compare-and-swap* (CAS) [62, 64, 137] and *load-linked/store-conditional* (LL/SC) [63, 70, 132]. The semantics of the CAS instruction is shown in Figure 1.2. For purposes of illustration, we assume an `atomically` keyword which requires the code block it labels to be executed *atomically*, that is, so that that no thread can observe a state in which the code block has been partially executed. The CAS operation atomically loads from a memory location, compares the value read to an expected value, and stores a new value to the location if the comparison succeeds. Herlihy [49] showed that instructions such as CAS and LL/SC are *universal*: there exists a wait-free implementation for *any* concurrent data structure in a system that supports such instructions.

A simple lock-free counter can be implemented using CAS. The idea is to perform the `fetch-and-inc` by loading the counter's value and to then use CAS to atomically change the counter value to a value greater by one than the value read. The CAS instruction fails to increment the counter only if it changes between the load and the CAS. In this case, the operation can retry, as the failing CAS had no effect. Because the CAS fails only as a result of another `fetch-and-inc` operation succeeding, the implementation is lock-free. However, it is not wait-free because a single `fetch-and-inc` operation can repeatedly fail its CAS.

The above example illustrates an *optimistic* approach to synchronization: the `fetch-and-inc` operation completes quickly in the hopefully common case in which it does not encounter interference from a concurrent operation, but must employ more expensive techniques under contention (e.g., backoff).

While the lock-free counter described above is simple, it has many of the same disadvantages that the original counter based on a single lock has: a sequential bottleneck and high contention for a single location. It is natural to attempt to apply similar techniques as those described above in order to improve the performance of this simple implementation. However, it is usually more difficult to incorporate such improvements into nonblocking implementations of concurrent data structures than blocking ones. Roughly, the reason for this is that a thread can use a lock to prevent other threads from "interfering" while it performs some sequence of actions. Without locks, we have to design our implementations to be correct despite the actions of concurrent operations; in current architectures, this often leads to the use of complicated and expensive techniques that undermine the improvements we are trying to incorporate. As discussed further in Section 1.1.7, transactional mechanisms make it much easier to design and modify efficient implementations of complex concurrent data structures. However, hardware support for such mechanisms does not yet exist.

1.1.4 Complexity Measures

A wide body of research is directed at analyzing the asymptotic complexity of concurrent data structures and algorithms in idealized models such as *parallel random access machines* [35, 122, 135]. However, there is less work on modelling such data structures in a realistic multiprocessor setting. There are many reasons for this, most of which have to do with the interplay of the architectural features of the hardware and the asynchronous execution of threads. Consider the combining tree example. Though we argued a speedup of $O(P/\log P)$ by counting instructions, this is not reflected in empirical studies [52, 129]. Real-world behavior is dominated by other features discussed above, such as cost of contention, cache behavior, cost of universal synchronization operations (e.g. CAS), arrival rates of requests, effects of backoff delays, layout of the data structure in memory, and so on. These factors are hard to quantify in a single precise model spanning all current architectures. Complexity measures that capture some of these aspects have been proposed by Dwork et al. [28] and by Anderson and Yang [7]. While these measures provide useful insight into algorithm design, they cannot accurately capture the effects of all of the factors listed above. As a result, concurrent data structure designers compare their designs empirically by running them using micro-benchmarks on real machines and simulated architectures [9, 52, 97, 103]. In the remainder of this chapter, we generally qualify data structures based on their empirically observed behavior and use simple complexity arguments only to aid intuition.

1.1.5 Correctness

It is easy to see that the behavior of the simple lock-based counter is "the same" as that of the sequential implementation. However, it is significantly more difficult to see this is also true for the combining tree. In general, concurrent data structure implementations are often subtle, and incorrect implementations are not uncommon. Therefore, it is important to be able to state and prove rigorously that a particular design correctly implements the required concurrent data structure. We cannot hope to achieve this without a precise way of specifying what "correct" means.

Data structure specifications are generally easier for sequential data structures. For example, we can specify the semantics of a sequential data structure by choosing a set of states, and providing a *transition function* that takes as arguments a state, an operation name and arguments to the operation, and returns a new state and a return value for the operation. Together with a designated initial state, the transition function specifies all acceptable sequences of operations on the data structure. The sequential semantics of the counter is specified as follows: The set of states for the counter is the set of integers, and the initial state is 0. The transition function for the `fetch-and-inc` operation adds one to the old state to obtain the new state, and the return value is the old state of the counter.

Operations on a sequential data structure are executed one-at-a-time in order, and we simply require that the resulting sequence of operations respects the sequential semantics specified as discussed above. However, with concurrent data structures, operations are not necessarily totally ordered. Correctness conditions for concurrent data structures generally require that *some* total order of the operations exists that respects the sequential semantics. Different conditions are distinguished by their different requirements on this total ordering.

A common condition is Lamport's *sequential consistency* [81], which requires that the total order preserves the order of operations executed by each thread. Sequential consistency has a drawback from the software engineering perspective: a data structure implemented using sequentially consistent components may not be sequentially consistent itself.

A natural and widely used correctness condition that overcomes this problem is Herlihy

and Wing's *linearizability* [58], a variation on the *serializability* [16] condition used for database transactions. Linearizability requires two properties: (1) that the data structure be sequentially consistent, and (2) that the total ordering which makes it sequentially consistent respect the *real-time ordering* among the operations in the execution. Respecting the real-time ordering means that if an operation O_1 finishes execution before another operation O_2 begins (so the operations are not concurrent with each other), then O_1 must be ordered before O_2. Another way of thinking of this condition is that it requires us to be able to identify a distinct point within each operation's execution interval, called its *linearization point*, such that if we order the operations according to the order of their linearization points, the resulting order obeys the desired sequential semantics.

It is easy to see that the counter implementation based on a single lock is linearizable. The state of the counter is always stored in the variable `X`. We define the linearization point of each `fetch-and-inc` operation as the point at which it stores its incremented value to `X`. The linearizability argument for the CAS-based lock-free implementation is similarly simple, except that we use the semantics of the CAS instruction, rather than reasoning about the lock, to conclude that the counter is incremented by one each time it is modified.

For the combining tree, it is significantly more difficult to see that the implementation is linearizable because the state of the counter is incremented by more than one at a time, and because the increment for one `fetch-and-inc` operation may in fact be performed by another operation with which it has combined. We define the linearization points of `fetch-and-inc` operations on the combining-tree-based counter as follows. When a winner thread reaches the root of the tree and adds its accumulated value to the counter, we linearize each of the operations with which it has combined in sequence immediately after that point. The operations are linearized in the order of the return values that are subsequently distributed to those operations. While a detailed linearizability proof is beyond the scope of our presentation, it should be clear from this discussion that rigorous correctness proofs for even simple concurrent data structures can be quite challenging.

The intuitive appeal and modularity of linearizability makes it a popular correctness condition, and most of the concurrent data structure implementations we discuss in the remainder of this chapter are linearizable. However, in some cases, performance and scalability can be improved by satisfying a weaker correctness condition. For example, the *quiescent consistency* condition [10] drops the requirement that the total ordering of operations respects the real-time order, but requires that every operation executed after a quiescent state—one in which no operations are in progress—must be ordered after every operation executed before that quiescent state. Whether an implementation satisfying such a weak condition is useful is application-dependent. In contrast, a linearizable implementation is always usable, because designers can view it as atomic.

1.1.6 Verification Techniques

In general, to achieve a rigorous correctness proof for a concurrent data structure implementation, we need mathematical machinery for specifying correctness requirements, accurately modelling a concurrent data structure implementation, and ensuring that a proof that the implementation is correct is complete and accurate. Most linearizability arguments in the literature treat at least some of this machinery informally, and as a result, it is easy to miss cases, make incorrect inferences, etc. Rigorous proofs inevitably contain an inordinate amount of mundane details about trivial properties, making them tedious to write and to read. Therefore, computer-assisted methods for verifying implementations are required. One approach is to use a *theorem prover* to prove a series of assertions which together imply that an implementation is correct. Another approach is to use *model checking* tools,

```
void acquire(Lock *lock) {
  int delay = MIN_DELAY;
  while (true) {
    if (CAS(lock,UNOWNED,OWNED))
      return;
    sleep(random() % delay);
    if (delay < MAX_DELAY)
      delay = 2 * delay;
  }
}

void release(Lock *lock) {
  *lock = UNOWNED;
}
```

FIGURE 1.3: Exponential backoff lock.

which exhaustively check all possible executions of an implementation to ensure that each one meets specified correctness conditions. The theorem proving approach usually requires significant human insight, while model checking is limited by the number of states of an implementation that can be explored.

1.1.7 Tools of the Trade

Below we discuss three key types of tools one can use in designing concurrent data structures: locks, barriers, and transactional synchronization mechanisms. Locks and barriers are traditional low-level synchronization mechanisms that are used to restrict certain interleavings, making it easier to reason about implementations based on them. Transactional mechanisms seek to hide the tricky details of concurrency from programmers, allowing them to think in a more traditional sequential way.

Locks

As discussed earlier, locks are used to guarantee mutually exclusive access to (parts of) a data structure, in order to avoid "bad" interleavings. A key issue in designing a lock is what action to take when trying to acquire a lock already held by another thread. On uniprocessors, the only sensible option is to yield the processor to another thread. However, in multiprocessors, it may make sense to repeatedly attempt to acquire the lock, because the lock may soon be released by a thread executing on another processor. Locks based on this technique are called *spinlocks*. The choice between blocking and spinning is often a difficult one because it is hard to predict how long the lock will be held. When locks are supported directly by operating systems or threads packages, information such as whether the lock-holder is currently running can be used in making this decision.

A simple spinlock repeatedly uses a synchronization primitive such as compare-and-swap to atomically change the lock from unowned to owned. As mentioned earlier, if locks are not designed carefully, such spinning can cause heavy contention for the lock, which can have a severe impact on performance. A common way to reduce such contention is to use *exponential backoff* [3]. In this approach, which is illustrated in Figure 1.3, a thread that is unsuccessful in acquiring the lock waits some time before retrying; repeated failures cause longer waiting times, with the idea that threads will "spread themselves out" in time, resulting in lower contention and less memory traffic due to failed attempts.

Exponential backoff has the disadvantage that the lock can be unowned, but all threads attempting to acquire it have backed off too far, so none of them is making progress. One way to overcome this is to have threads form a queue and have each thread that releases the

lock pass ownership of the lock to the next queued thread. Locks based on this approach are called *queuelocks*. Anderson [8] and Graunke and Thakkar [38] introduce array-based queuelocks, and these implementations are improved upon by the list-based MCS queue locks of Mellor-Crummey and Scott [97] and the CLH queue locks of Craig and Landin and Hagersten [25, 94].

Threads using CLH locks form a virtual linked list of nodes, each containing a `done` flag; a thread enters the critical section only after the `done` flag of the node preceding its own node in the list is raised. The lock object has a pointer to the node at the tail of the list, the last one to join it. To acquire the lock, a thread creates a node, sets its `done` flag to false indicate that it has not yet released the critical section, and uses a synchronization primitive such as CAS to place its own node at the tail of the list while determining the node of its predecessor. It then spins on the `done` flag of the predecessor node. Note that each thread spins on a different memory location. Thus, in a cache-based architecture, when a thread sets its `done` flag to inform the next thread in the queue that it can enter the critical section, the `done` flags on which all other threads are spinning are not modified, so those threads continue to spin on a local cache line, and do not produce additional memory traffic. This significantly reduces contention and improves scalability in such systems. However, if this algorithm is used in a non-coherent NUMA machine, threads will likely have to spin on remote memory locations, again causing excessive memory traffic. The MCS queuelock [97] overcomes this problem by having each thread spin on a `done` flag in its *own* node. This way, nodes can be allocated in local memory, eliminating the problem.

There are several variations on standard locks that are of interest to the data structure designer in some circumstances. The queuelock algorithms discussed above have more advanced *abortable* versions that allow threads to give up on waiting to acquire the lock, for example, if they are delayed beyond some limit in a real-time application [123, 125], or if they need to recover from deadlock. The algorithms of [123] provide an abort that is nonblocking. Finally, [103] presents *preemption-safe* locks, which attempt to reduce the negative performance effects of preemption by ensuring that a thread that is in the queue but preempted does not prevent the lock from being granted to another running thread.

In many data structures it is desirable to have locks that allow concurrent readers. Such *reader-writer* locks allow threads that only read data in the critical section (but do not modify it) to access the critical section exclusively from the writers but concurrently with each other. Various algorithms have been suggested for this problem. The reader-writer queuelock algorithms of Mellor-Crummey and Scott [98] are based on MCS queuelocks and use read counters and a special pointer to writer nodes. In [76] a version of these algorithms is presented in which readers remove themselves from the lock's queue. This is done by keeping a doubly-linked list of queued nodes, each having its own simple "mini-lock." Readers remove themselves from the queuelock list by acquiring mini-locks of their neighboring nodes and redirecting the pointers of the doubly-linked list. The above-mentioned real-time queuelock algorithms of [123] provide a similar ability without locking nodes.

The reader-writer lock approach can be extended to arbitrarily many operation types through a construct called *group mutual exclusion* or *room synchronization*. The idea is that operations are partitioned into groups, such that operations in the same group can execute concurrently with each other, but operations in different groups must not. An interesting application of this approach separates push and pop operations on a stack into different groups, allowing significant simplifications to the implementations of those operations because they do not have to deal with concurrent operations of different types [18]. Group mutual exclusion was introduced by Joung [69]. Implementations based on mutual exclusion locks or `fetch-and-inc` counters appear in [18, 71].

More complete and detailed surveys of the literature on locks can be found in [6, 117].

Barriers

A barrier is a mechanism that collectively halts threads at a given point in their code, and allows them to proceed only when all threads have arrived at that point. The use of barriers arises whenever access to a data structure or application is layered into phases whose execution should not overlap in time. For example, repeated iterations of a numerical algorithm that converges by iterating a computation on the same data structure or the marking and sweeping phases of a parallel garbage collector.

One simple way to implement a barrier is to use a counter that is initialized to the total number of threads: Each thread decrements the counter upon reaching the barrier, and then spins, waiting for the counter to become zero before proceeding. Even if we use the techniques discussed earlier to implement a scalable counter, this approach still has the problem that waiting threads produce contention. For this reason, specialized barrier implementations have been developed that arrange for each thread to spin on a different location [21, 48, 124]. Alternatively, a barrier can be implemented using a *diffusing computation* tree in the style of Dijkstra and Scholten [27]. In this approach, each thread is the owner of one node in a binary tree. Each thread awaits the arrival of its children, then notifies its parent that it has arrived. Once all threads have arrived, the root of the tree releases all threads by disseminating the release information down the tree.

Transactional Synchronization Mechanisms

A key use of locks in designing concurrent data structures is to allow threads to modify multiple memory locations atomically, so that no thread can observe partial results of an update to the locations. Transactional synchronization mechanisms are tools that allow the programmer to treat sections of code that access multiple memory locations as a single atomic step. This substantially simplifies the design of correct concurrent data structures because it relieves the programmer of the burden of deciding which locks should be held for which memory accesses and of preventing deadlock.

As an example of the benefits of transactional synchronization, consider Figure 1.4, which shows a concurrent queue implementation achieved by requiring operations of a simple sequential implementation to be executed atomically. Such atomicity could be ensured either by using a global mutual exclusion lock, or via a transactional mechanism. However, the lock-based approach prevents concurrent `enqueue` and `dequeue` operations from executing in parallel. In contrast, a good transactional mechanism will allow them to do so in all but the empty state because when the queue is not empty, concurrent `enqueue` and `dequeue` operations do not access any memory locations in common.

The use of transactional mechanisms for implementing concurrent data structures is inspired by the widespread use of transactions in database systems. However, the problem of supporting transactions over shared memory locations is different from supporting transactions over databases elements stored on disk. Thus, more lightweight support for transactions is possible in this setting.

Kung and Robinson's *optimistic concurrency control* (OCC) [80] is one example of a transactional mechanism for concurrent data structures. OCC uses a global lock, which is held only for a short time at the end of a transaction. Nonetheless, the lock is a sequential bottleneck, which has a negative impact on scalability. Ideally, transactions should be supported without the use of locks, and transactions that access disjoint sets of memory locations should not synchronize with each other at all.

Transactional support for multiprocessor synchronization was originally suggested by Herlihy and Moss, who also proposed a hardware-based *transactional memory* mechanism for supporting it [56]. Recent extensions to this idea include *lock elision* [114, 115], in which the

```
typedef struct qnode_s { qnode_s *next; valuetype value; } qnode_t;

shared variables:
  // initially null
  qnode_t *head, *tail;

void enqueue(qnode_t *n) {
  atomically {
    if (tail == null)
      tail = head = n;
    else {
      tail->next = n;
      tail = n;
    }
  }
}

qnode_t * dequeue() {
  atomically {
    if (head == null)
      return null;
    else {
      n = head;
      head = head->next;
      if (head == null)
        tail = null;
      return n;
    }
  }
}
```

FIGURE 1.4: A Concurrent Shared FIFO Queue.

hardware automatically translates critical sections into transactions, with the benefit that two critical sections that do not in fact conflict with each other can be executed in parallel. For example, lock elision could allow concurrent `enqueue` and `dequeue` operations of the above queue implementation to execute in parallel, even if the atomicity is implemented using locks. To date, hardware support for transactional memory has not been built.

Various forms of *software transactional memory* have been proposed by Shavit and Touitou [128], Harris et al. [44], Herlihy et al. [55], and Harris and Fraser [43].

Transactional mechanisms can easily be used to implement most concurrent data structures, and when efficient and robust transactional mechanisms become widespread, this will likely be the preferred method. In the following sections, we mention implementations based on transactional mechanisms only when no direct implementation is known.

1.2 Shared Counters and Fetch-and-ϕ Structures

Counters have been widely studied as part of a broader class of *fetch-and-ϕ* coordination structures, which support operations that fetch the current value of a location and apply some function from an allowable set ϕ to its contents. As discussed earlier, simple lock-based implementations of fetch-and-ϕ structures such as counters suffer from contention and sequential bottlenecks. Below we describe some approaches to overcoming this problem.

Combining

The combining tree technique was originally invented by Gottlieb et al. [37] to be used in the hardware switches of processor-to-memory networks. In Section 1.1.2 we discussed a software version of this technique, first described by Goodman et al. [36] and Yew et al. [138], for implementing a fetch-and-add counter. (The algorithm in [138] has a slight bug; see [52].) This technique can also be used to implement fetch-and-ϕ operations for a variety of sets of combinable operations, including arithmetic and boolean operations, and synchronization operations such as load, store, swap, test-and-set, etc. [77].

As explained earlier, scalability is achieved by sizing the tree such that the there is one leaf node per thread. Under maximal load, the throughput of such a tree is proportional

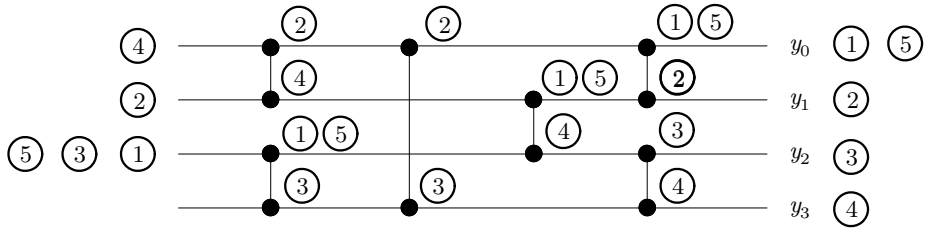

FIGURE 1.5: A Bitonic Counting Network of Width Four.

to $O(P/\log P)$ operations per time unit, offering a significant speedup. Though it is possible to construct trees with fan-out greater than two in order to reduce tree depth, that would sacrifice the simplicity of the nodes and, as shown by Shavit and Zemach [131], will most likely result in reduced performance. Moreover, Herlihy et al. [52] have shown that combining trees are extremely sensitive to changes in the arrival rate of requests: as the load decreases, threads must still pay the price of traversing the tree while attempting to combine, but the likelihood of combining is reduced because of the reduced load.

Shavit and Zemach overcome the drawbacks of the static combining tree structures by introducing combining funnels [131]. A *combining funnel* is a linearizable fetch-and-ϕ structure that allows combining trees to form dynamically, adapting its overall size based on load patterns. It is composed of a (typically small) number of *combining layers*. Each such layer is implemented as a *collision array* in memory. Threads pass through the funnel layer by layer, from the first (widest) to the last (narrowest). These layers are used by threads to locate each other and combine their operations. As threads pass through a layer, they read a thread ID from a randomly chosen array element, and write their own in its place. They then attempt to combine with the thread whose ID they read. A successful combination allows threads to exchange information, allowing some to continue to the next layer, and others to await their return with the resulting value. Combining funnels can also support the *elimination* technique (described in Section 1.3) to allow two operations to complete without accessing the central data structure in some cases.

Counting Networks

Combining structures provide scalable and linearizable fetch-and-ϕ operations. However, they are blocking. An alternative approach to parallelizing a counter that overcomes this problem is to have multiple counters instead of a single one, and to use a counting network to coordinate access to the separate counters so as to avoid problems such as duplicated or omitted values. *Counting networks*, introduced by Aspnes et al. [10], are a class of data structures for implementing, in a highly concurrent and nonblocking fashion, a restricted class of fetch-and-ϕ operations, the most important of which is `fetch-and-inc`.

Counting networks, like sorting networks [24], are acyclic networks constructed from simple building blocks called balancers. In its simplest form, a *balancer* is a computing element with two input wires and two output wires. Tokens arrive on the balancer's input wires at arbitrary times, and are output on its output wires in a balanced way. Given a stream of input tokens, a balancer alternates sending one token to the top output wire, and one to the bottom, effectively balancing the number of tokens between the two wires.

We can wire balancers together to form a network. The *width* of a network is its number of output wires (wires that are not connected to an input of any balancer). Let $y_0, .., y_{w-1}$ respectively represent the number of tokens output on each of the output wires of a network

of width w. A *counting network* is an acyclic network of balancers whose outputs satisfy the following *step property*:

> In any quiescent state, $0 \leq y_i - y_j \leq 1$ for any $i < j$.

Figure 1.5 shows a sequence of tokens traversing a counting network of width four based on Batcher's Bitonic sorting network structure [13]. The horizontal lines are wires and the vertical lines are balancers, each connected to two input and output wires at the dotted points. Tokens (numbered 1 through 5) traverse the balancers starting on arbitrary input wires and accumulate on specific output wires meeting the desired step-property. Aspnes et al. [10] have shown that every counting network has a layout isomorphic to a sorting network, but not every sorting network layout is isomorphic to a counting network.

On a shared memory multiprocessor, balancers are records, and wires are pointers from one record to another. Threads performing increment operations traverse the data structure from some input wire (either preassigned or chosen at random) to some output wire, each time shepherding a new token through the network.

The counting network distributes input tokens to output wires while maintaining the step property stated above. Counting is done by adding a local counter to each output wire, so that tokens coming out of output wire i are assigned numbers $i, i + w, \ldots, i + (y_i - 1)w$. Because threads are distributed across the counting network, there is little contention on the balancers, and the even distribution on the output wires lowers the load on the shared counters. However, as shown by Shavit and Zemach [129], the dynamic patterns through the networks increase cache miss rates and make optimized layout almost impossible.

There is a significant body of literature on counting networks, much of which is surveyed by Herlihy and Busch [22]. An empirical comparison among various counting techniques can be found in [52]. Aharonson and Attiya [4] and Felten et al. [31] study counting networks with arbitrary fan-in and fan-out. Shavit and Touitou [127] show how to perform decrements on counting network counters by introducing the notion of "anti-tokens" and elimination. Busch and Mavronicolas [23] provide a combinatorial classification of the various properties of counting networks. Randomized counting networks are introduced by Aiello et al. [5] and fault-tolerant networks are presented by Riedel and Bruck [118].

The classical counting network structures in the literature are lock-free but not linearizable, they are only quiescently consistent. Herlihy et al. [57] show the tradeoffs involved in making counting networks linearizable.

Klugerman and Plaxton present an optimal $\log w$-depth counting network [73]. However, this construction is not practical, and all practical counting network implementations have $\log^2 w$ depth. Shavit and Zemach introduce *diffracting trees* [129], improved counting networks made of balancers with one input and two output wires laid out as a binary tree. The simple balancers of the counting network are replaced by more sophisticated *diffracting balancers* that can withstand high loads by using a randomized collision array approach, yielding lower depth counting networks with significantly improved throughput. An adaptive diffracting tree that adapts its size to load is presented in [26].

1.3 Stacks and Queues

Stacks and queues are among the simplest sequential data structures. Numerous issues arise in designing concurrent versions of these data structures, clearly illustrating the challenges involved in designing data structures for shared-memory multiprocessors.

Stacks

A concurrent *stack* is a data structure linearizable to a sequential stack that provides `push` and `pop` operations with the usual *LIFO* semantics. Various alternatives exist for the behavior of these data structures in full or empty states, including returning a special value indicating the condition, raising an exception, or blocking.

Michael and Scott present several linearizable lock-based concurrent stack implementations: they are based on sequential linked lists with a top pointer and a global lock that controls access to the stack [103]. They typically scale poorly because even if one reduces contention on the lock, the top of the stack is a sequential bottleneck. Combining funnels [131] have been used to implement a linearizable stack that provides parallelism under high load. As with all combining structures, it is blocking, and it has a high overhead which makes it unsuitable for low loads.

Treiber [134] was the first to propose a lock-free concurrent stack implementation. He represented the stack as a singly-linked list with a top pointer and used CAS to modify the value of the top pointer atomically. Michael and Scott [103] compare the performance of Treiber's stack to an optimized nonblocking algorithm based on Herlihy's methodology [50], and several lock-based stacks (such as an MCS lock [97]) in low load situations. They concluded that Treiber's algorithm yields the best overall performance, and that this performance gap increases as the degree of multiprogramming grows. However, because the top pointer is a sequential bottleneck, even with an added backoff mechanism to reduce contention, the Treiber stack offers little scalability as concurrency increases [47].

Hendler et al. [47] observe that any stack implementation can be made more scalable using the *elimination* technique of Shavit and Touitou [127]. Elimination allows pairs of operations with reverse semantics—like pushes and pops on a stack—to complete without any central coordination, and therefore substantially aids scalability. The idea is that if a `pop` operation can find a concurrent `push` operation to "partner" with, then the `pop` operation can take the `push` operation's value, and both operations can return immediately. The net effect of each pair is the same as if the `push` operation was followed immediately by the `pop` operation, in other words, they eliminate each other's effect on the state of the stack. Elimination can be achieved by adding a collision array from which each operation chooses a location at random, and then attempts to coordinate with another operation that concurrently chose the same location [127]. The number of eliminations grows with concurrency, resulting in a high degree of parallelism. This approach, especially if the collision array is used as an adaptive backoff mechanism on the shared stack, introduces a high degree of parallelism with little contention [47], and delivers a scalable lock-free linearizable stack.

There is a subtle point in the Treiber stack used in the implementations above that is typical of many CAS-based algorithms. Suppose several concurrent threads all attempt a `pop` operation that removes the first element, located in some node "A," from the list by using a CAS to redirect the head pointer to point to a previously-second node "B." The problem is that it is possible for the list to change completely just before a particular `pop` operation attempts its CAS, so that by the time it does attempt it, the list has the node "A" as the first node as before, but the rest of the list including "B" is in a completely different order. This CAS of the head pointer from "A" to "B" may now succeed, but "B" might be anywhere in the list and the stack will behave incorrectly. This is an instance of the "ABA" problem [111], which plagues many CAS-based algorithms. To avoid this problem, Treiber augments the head pointer with a version number that is incremented every time the head pointer is changed. Thus, in the above scenario, the changes to the stack would cause the

CAS to fail, thereby eliminating the ABA problem.[4]

Queues

A concurrent *queue* is a data structure linearizable to a sequential queue that provides `enqueue` and `dequeue` operations with the usual *FIFO* semantics.

Michael and Scott [103] present a simple lock-based queue implementation that improves on the naive single-lock approach by having separate locks for the head and tail pointers of a linked-list-based queue. This allows an `enqueue` operation to execute in parallel with a `dequeue` operation (provided we avoid false sharing by placing the head and tail locks in separate cache lines). This algorithm is quite simple, with one simple trick: a "dummy" node is always in the queue, which allows the implementation to avoid acquiring both the head and tail locks in the case that the queue is empty, and therefore it avoids deadlock.

It is a matter of folklore that one can implement an array-based lock-free queue for a single enqueuer thread and a single dequeuer thread using only load and store operations [82]. A linked-list-based version of this algorithm appears in [46]. Herlihy and Wing [58] present a lock-free array-based queue that works if one assumes an unbounded size array. A survey in [103] describes numerous flawed attempts at devising general (multiple enqueuers, multiple dequeuers) nonblocking queue implementations. It also discusses some correct implementations that involve much more overhead than the ones discussed below.

Michael and Scott [103] present a linearizable CAS-based lock-free queue with parallel access to both ends. The structure of their algorithm is very simple and is similar to the two-lock algorithm mentioned above: it maintains head and tail pointers, and always keeps a dummy node in the list. To avoid using a lock, the `enqueue` operation adds a new node to the end of the list using CAS, and then uses CAS to update the tail pointer to reflect the addition. If the `enqueue` is delayed between these two steps, another `enqueue` operation can observe the tail pointer "lagging" behind the end of the list. A simple *helping technique* [50] is used to recover from this case, ensuring that the tail pointer is always behind the end of the list by at most one element.

While this implementation is simple and efficient enough to be used in practice, it does have a disadvantage. Operations can access nodes already removed from the list, and therefore the nodes cannot be freed. Instead, they are put into a *freelist*—a list of nodes stored for reuse by future `enqueue` operations—implemented using Treiber's stack. This use of a freelist has the disadvantage that the space consumed by the nodes in the freelist cannot be freed for arbitrary reuse. Herlihy et al. [53] and Michael [101] have presented nonblocking memory management techniques that overcome this disadvantage.

It is interesting to note that the elimination technique is not applicable to queues: we cannot simply pass a value from an `enqueue` operation to a concurrent `dequeue` operation, because this would not respect the FIFO order with respect to other values in the queue.

Deques

A concurrent double-ended queue (*deque*) is a linearizable concurrent data structure that generalizes concurrent stacks and queues by allowing pushes and pops at both ends [74].

[4] Note that the version number technique does not technically eliminate the ABA problem because the version number can wrap around; see [106] for a discussion of the consequences of this point in practice, and also a "bounded tag" algorithm that eliminates the problem entirely, at some cost in space and time.

As with queues, implementations that allow operations on both ends to proceed in parallel without interfering with each other are desirable.

Lock-based deques can be implemented easily using the same two-lock approach used for queues. Given the relatively simple lock-free implementations for stacks and queues, it is somewhat surprising that there is no known lock-free deque implementation that allows concurrent operations on both ends. Martin et al. [96] provide a summary of concurrent deque implementations, showing that, even using nonconventional two-word synchronization primitives such as *double-compare-and-swap* (DCAS) [107], it is difficult to design a lock-free deque. The only known nonblocking deque implementation for current architectures that supports noninterfering operations at opposite ends of the deque is an obstruction-free CAS-based implementation due to Herlihy et al. [54].

1.4 Pools

Much of the difficulty in implementing efficient concurrent stacks and queues arises from the ordering requirements on when an element that has been inserted can be removed. A concurrent *pool* [95] is a data structure that supports `insert` and `delete` operations, and allows a `delete` operation to remove *any* element that has been inserted and not subsequently deleted. This weaker requirement offers opportunities for improving scalability.

A high-performance pool can be built using any quiescently consistent counter implementation [10, 129]. Elements are placed in an array, and a `fetch-and-inc` operation is used to determine in which location an `insert` operation stores its value, and similarly from which location a `delete` operation takes its value. Each array element contains a full/empty bit or equivalent mechanism to indicate if the element to be removed has already been placed in the location. Using such a scheme, any one of the combining tree, combining funnel, counting network, or diffracting tree approaches described above can be used to create a high throughput shared pool by parallelizing the main bottlenecks: the shared counters. Alternatively, a "stack like" pool can be implemented by using a counter that allows increments and decrements, and again using one of the above techniques to parallelize it.

Finally, the elimination technique discussed earlier is applicable to pools constructed using combining funnels, counting networks, or diffracting trees: if `insert` and `delete` operations meet in the tree, the `delete` can take the value being inserted by the `insert` operation, and both can leave without continuing to traverse the structure. This technique provides high performance under high load.

The drawback of all these implementations is that they perform rather poorly under low load. Moreover, when used for work-load distribution [9, 19, 119], they do not allow us to exploit locality information, as pools designed specifically for work-load distribution do.

Workload distribution (or *load balancing*) algorithms involve a collection of pools of units of work to be done; each pool is local to a given processor. Threads create work items and place them in local pools, employing a load balancing algorithm to ensure that the number of items in the pools is balanced. This avoids the possibility that some processors are idle while others still have work in their local pools. There are two general classes of algorithms of this type: *work sharing* [46, 119] and *work stealing* [9, 19]. In a work sharing scheme, each processor attempts to continuously offload work from its pool to other pools. In work stealing, a thread that has no work items in its local pool steals work from other pools. Both classes of algorithms typically use randomization to select the pool with which to balance or the target pool for stealing.

The classical work stealing algorithm is due to Arora et al. [9]. It is based on a lock-free construction of a *deque* that allows operations by only one thread (the thread to which

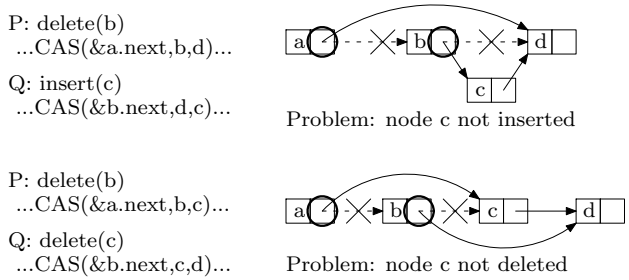

FIGURE 1.6: CAS-based list manipulation is hard. In both examples, P is deleting b from the list (the examples slightly abuse CAS notation). In the upper example, Q is trying to insert c into the list, and in the lower example, Q is trying to delete c from the list. Circled locations indicate the target addresses of the CAS operations; crossed out pointers are the values before the CAS succeeds.

the pool is local) at one end of the deque, allowing only pop operations at the other end, and allowing concurrent pop operations at that end to "abort" if they interfere. A deque with these restrictions is suitable for work stealing, and the restrictions allow a simple implementation in which the local thread can insert and delete using simple low-cost load and store operations, resorting to a more expensive CAS operation only when it competes with the remote deleters for the last remaining item in the queue.

It has been shown that in some cases it is desirable to steal more than one item at a time [15, 104]. A nonblocking multiple-item work-stealing algorithm due to Hendler and Shavit appears in [45]. It has also been shown that in some cases it desirable to use affinity information of work items in deciding which items to steal. A locality-guided work stealing algorithm due to Acar et al. appears in [1].

1.5 Linked Lists

Consider implementations of concurrent search structures supporting `insert`, `delete`, and `search` operations. If these operations deal only with a key value, then the resulting data structure is a *set*; if a data value is assosciated with each key, we have a *dictionary* [24]. These are closely related data structures, and a concurrent set implementation can often be adapted to implement a dictionary. In the next three sections, we concentrate on implementing sets using different structures: linked lists, hash tables, and trees.

Suppose we use a linked list to implement a set. Apart from globally locking the linked list to prevent concurrent manipulation, the most popular approach to concurrent lock-based linked lists is *hand-over-hand locking* (sometimes called *lock coupling*) [14, 90]. In this approach, each node has an associated lock. A thread traversing the linked list releases a node's lock only after acquiring the lock of the next node in the list, thus preventing overtaking which may cause unnoticed removal of a node. This approach reduces lock granularity but significantly limits concurrency because insertions and deletions at disjoint list locations may delay each other.

One way to overcome this problem is to design lock-free linked lists. The difficulty in implementing a lock-free ordered linked list is ensuring that during an insertion or deletion, the adjacent nodes are still valid, i.e., they are still in the list and are still adjacent. As Figure 1.6 shows, designing such lock-free linked lists is not a straightforward matter.

The first CAS-based lock-free linked list is due to Valois [136], who uses a special auxiliary

node in front of every regular node to prevent the undesired phenomena depicted in Figure 1.6. Valois's algorithm is correct when combined with a memory management solution due to Michael and Scott [102], but this solution is not practical. Harris [42] presents a lock-free list that uses a special "deleted" bit that is accessed atomically with node pointers in order to signify that a node has been deleted; this scheme is applicable only in garbage collected environments. Michael [100] overcomes this disadvantage by modifying Harris's algorithm to make it compatible with memory reclamation methods [53, 101].

1.6 Hash Tables

A typical extensible hash table is a resizable array of buckets, each holding an expected constant number of elements, and thus requiring on average a constant time for `insert`, `delete` and `search` operations [24]. The principal cost of *resizing*—the redistribution of items between old and new buckets—is amortized over all table operations, thus keeping the cost of operations constant on average. Here resizing means extending the table, as it has been shown that as a practical matter, hash tables need only increase in size [59].

Michael [100] shows that a concurrent non-extensible hash table can be achieved by placing a read-write lock on every bucket in the table. However, to guarantee good performance as the number of elements grows, hash tables must be extensible [30].

In the eighties, Ellis [29] and others [59, 78] extended the work of Fagin et al. [30] by designing an extensible concurrent hash table for distributed databases based on two-level locking schemes. A recent extensible hash algorithm by Lea [89] is known to be highly efficient in non-multiprogrammed environments [126]. It is based on a version of Litwin's sequential linear hashing algorithm [92]. It uses a locking scheme that involves a small number of high-level locks rather than a lock per bucket, and allows concurrent searches while resizing the table, but not concurrent inserts or deletes. Resizing is performed as a global restructuring of all buckets when the table size needs to be doubled.

Lock-based extensible hash-table algorithms suffer from all of the typical drawbacks of blocking synchronization, as discussed earlier. These problems become more acute because of the elaborate "global" process of redistributing the elements in all the hash table's buckets among newly added buckets. Lock-free extensible hash tables are thus a matter of both practical and theoretical interest.

As described in Section 1.5, Michael [100] builds on the work of Harris [42] to provide an effective CAS-based lock-free linked list implementation. He then uses this as the basis for a lock-free hash structure that performs well in multiprogrammed environments: a fixed-sized array of hash buckets, each implemented as a lock-free list. However, there is a difficulty in making a lock-free array of lists extensible since it is not obvious how to redistribute items in a lock-free manner when the bucket array grows. Moving an item between two bucket lists seemingly requires two CAS operations to be performed together atomically, which is not possible on current architectures.

Greenwald [39] shows how to implement an extensible hash table using his *two-handed emulation* technique. However, this technique employs a DCAS synchronization operation, which is not available on current architectures, and introduces excessive amounts of work during global resizing operations.

Shalev and Shavit [126] introduce a lock-free extensible hash table which works on current architectures. Their key idea is to keep the items in a single lock-free linked list instead of a list per bucket. To allow operations fast access to the appropriate part of the list, the Shalev-Shavit algorithm maintains a resizable array of "hints" (pointers into the list); operations use the hints to find a point in the list that is close to the relevant position, and then follow list

pointers to find the position. To ensure a constant number of steps per operation on average, finer grained hints must be added as the number of elements in the list grows. To allow these hints to be installed simply and efficiently, the list is maintained in a special *recursive split ordering*. This technique allows new hints to be installed incrementally, thereby eliminating the need for complicated mechanisms for atomically moving items between buckets, or reordering the list.

1.7 Search Trees

A concurrent implementation of any search tree can be achieved by protecting it using a single exclusive lock. Concurrency can be improved somewhat by using a reader-writer lock to allow all read-only (`search`) operations to execute concurrently with each other while holding the lock in *shared mode*, while update (`insert` or `delete`) operations exclude all other operations by acquiring the lock in *exclusive mode*. If update operations are rare, this may be acceptable, but with even a moderate number of updates, the exclusive lock for update operations creates a sequential bottleneck that degrades performance substantially. By using fine-grained locking strategies—for example by using one lock per node, rather than a single lock for the entire tree—we can improve concurrency further.

Kung and Lehman [79] present a concurrent binary search tree implementation in which update operations hold only a constant number of node locks at a time, and these locks only exclude other update operations: search operations are never blocked. However, this implementation makes no attempt to keep the search tree balanced. In the remainder of this section, we focus on balanced search trees, which are considerably more challenging.

As a first step towards more fine-grained synchronization in balanced search tree implementations, we can observe that it is sufficient for an operation to hold an exclusive lock on the subtree in which it causes any modifications. This way, update operations that modify disjoint subtrees can execute in parallel. We briefly describe some techniques in this spirit in the context of B^+-trees. Recall that in B^+-trees, all keys and data are stored in leaf nodes; internal nodes only maintain routing information to direct operations towards the appropriate leaf nodes. Furthermore, an insertion into a leaf may require the leaf to be split, which may in turn require a new entry to be added to the leaf's parent, which itself may need to be split to accommodate the new entry. Thus, an insertion can potentially result in modifying all nodes along the path from the root to the leaf. However, such behavior is rare, so it does not make sense to exclusively lock the whole path just in case this occurs.

As a first step to avoiding such conservative locking strategies, we can observe that if an `insert` operation passes an internal B^+-tree node that is not full, then the modifications it makes to the tree cannot propagate past that node. In this case, we say that the node is *safe* with respect to the `insert` operation. If an update operation encounters a safe node while descending the tree acquiring exclusive locks on each node it traverses, it can safely release the locks on all ancestors of that node, thereby improving concurrency by allowing other operations to traverse those nodes [99, 121]. Because `search` operations do not modify the tree, they can descend the tree using lock coupling: as soon as a lock has been acquired on a child node, the lock on its parent can be released. Thus, `search` operations hold at most two locks (in shared mode) at any point in time, and are therefore less likely to prevent progress by other operations.

This approach still requires each update operation to acquire an exclusive lock on the root node, and to hold the lock while reading a child node, potentially from disk, so the root is still a bottleneck. We can improve on this approach by observing that most update operations will not need to split or merge the leaf node they access, and will therefore

eventually release the exclusive locks on all of the nodes traversed on the way to the leaf. This observation suggests an "optimistic" approach in which we descend the tree acquiring the locks in shared mode, acquiring only the leaf node exclusively [14]. If the leaf does not need to be split or merged, the update operation can complete immediately; in the rare cases in which changes do need to propagate up the tree, we can release all of the locks and then retry with the more pessimistic approach described above. Alternatively, we can use reader-writer locks that allow locks held in a shared mode to be "upgraded" to exclusive mode. This way, if an update operation discovers that it does need to modify nodes other than the leaf, it can upgrade locks it already holds in shared mode to exclusive mode, and avoid completely restarting the operation from the top of the tree [14]. Various combinations of the above techniques can be used because nodes near the top of the tree are more likely to conflict with other operations and less likely to be modified, while the opposite is true of nodes near the leaves [14].

As we employ some of the more sophisticated techniques described above, the algorithms become more complicated, and it becomes more difficult to avoid deadlock, resulting in even further complications. Nonetheless, all of these techniques maintain the invariant that operations exclusively lock the subtrees that they modify, so operations do not encounter states that they would not encounter in a sequential implementation. Significant improvements in concurrency and performance can be made by relaxing this requirement, at the cost of making it more difficult to reason that the resulting algorithms are correct.

A key difficulty we encounter when we attempt to relax the strict subtree locking schemes is that an operation descending the tree might follow a pointer to a child node that is no longer the correct node because of a modification by a concurrent operation. Various techniques have been developed that allow operations to recover from such "confusion", rather than strictly avoiding it.

An important example in the context of B^+-trees is due to Lehman and Yao [91], who define B^{link}-trees: B^+-trees with "links" from each node in the tree to its right neighbor at the same level of the tree. These links allow us to "separate" the splitting of a node from modifications to its parent to reflect the splitting. Specifically, in order to split a node n, we can create a new node n' to its right, and install a link from n to n'. If an operation that is descending the tree reaches node n while searching for a key position that is now covered by node n' due to the split, the operation can simply follow the link from n to n' to recover. This allows a node to be split without preventing access by concurrent operations to the node's parent. As a result, update operations do not need to simultaneously lock the entire subtree they (potentially) modify. In fact, in the Lehman-Yao algorithm, update operations as well as `search` operations use the lock coupling technique so that no operation ever holds more than two locks at a time, which significantly improves concurrency. This technique has been further refined, so that operations never hold more than one lock at a time [120].

Lehman and Yao do not address how nodes can be merged, instead allowing `delete` operations to leave nodes underfull. They argue that in many cases `delete` operations are rare, and that if space utilization becomes a problem, the tree can occasionally be reorganized in "batch" mode by exclusively locking the entire tree. Lanin and Shasha [84] incorporate merging into the `delete` operations, similarly to how `insert` operations split overflowed nodes in previous implementations. Similar to the Lehman-Yao link technique, these implementations use links to allow recovery by operations that have mistakenly reached a node that has been evacuated due to node merging.

In all of the algorithms discussed above, the maintenance operations such as node splitting and merging (where applicable) are performed as part of the regular update operations. Without such tight coupling between the maintenance operations and the regular operations that necessitate them, we cannot guarantee strict balancing properties. However, if we relax

the balance requirements, we can separate the tree maintenance work from the update operations, resulting in a number of advantages that outweigh the desire to keep search trees strictly balanced. As an example, the B^{link}-tree implementation in [120] supports a *compression process* that can run concurrently with regular operations to merge nodes that are underfull. By separating this work from the regular update operations, it can be performed concurrently by threads running on different processors, or in the background.

The idea of separating rebalancing work from regular tree operations was first suggested for red-black trees [40], and was first realized in [72] for AVL trees [2] supporting `insert` and `search` operations. An implementation that also supports `delete` operations is provided in [109]. These implementations improve concurrency by breaking balancing work down into small, local tree transformations that can be performed independently. Analysis in [85] shows that with some modifications, the scheme of [109] guarantees that each update operation causes at most $O(\log N)$ rebalancing operations for an N-node AVL tree. Similar results exist for B-trees [88, 109] and red-black trees [20, 108].

The only nonblocking implementations of balanced search trees have been achieved using Dynamic Software Transactional Memory mechanisms [33, 55]. These implementations use transactions translated from sequential code that performs rebalancing work as part of regular operations.

The above brief survey covers only basic issues and techniques involved with implementing concurrent search trees. To mention just a few of the numerous improvements and extensions in the literature, [105] addresses practical issues for the use of B^+-trees in commercial database products, such as recovery after failures; [75] presents concurrent implementations for *generalized search trees* (GiSTs) that facilitate the design of search trees without repeating the delicate work involved with concurrency control; and [86, 87] present several types of trees that support the efficient insertion and/or deletion of a group of values. Pugh [112] presents a concurrent version of his skiplist randomized search structure [113]. *Skiplists* are virtual tree structures consisting of multiple layers of linked lists. The expected search time in a skiplist is logarithmic in the number of elements in it. The main advantage of skiplists is that they do not require rebalancing: insertions are done in a randomized fashion that keeps the search tree balanced.

Empirical and analytical evaluations of concurrent search trees and other data structures can be found in [41, 67].

1.8 Priority Queues

A concurrent *priority queue* is a data structure linearizable to a sequential priority queue that provides `insert` and `delete-min` operations with the usual priority queue semantics.

Heap-Based Priority Queues

Many of the concurrent priority queue constructions in the literature are linearizable versions of the heap structures described earlier in this book. Again, the basic idea is to use fine-grained locking of the individual heap nodes to allow threads accessing different parts of the data structure to do so in parallel where possible. A key issue in designing such concurrent heaps is that traditionally `insert` operations proceed from the bottom up and `delete-min` operations from the top down, which creates potential for deadlock. Biswas and Brown [17] present such a lock-based heap algorithm assuming specialized "cleanup" threads to overcome deadlocks. Rao and Kumar [116] suggest to overcome the drawbacks of [17] using an algorithm that has both `insert` and `delete-min` operations proceed from the top down. Ayani [11] improved on their algorithm by suggesting a way to have consecutive

insertions be performed on opposite sides of the heap. Jones [68] suggests a scheme similar to [116] based on a skew heap.

Hunt et al. [61] present a heap-based algorithm that overcomes many of the limitations of the above schemes, especially the need to acquire multiple locks along the traversal path in the heap. It proceeds by locking for a short duration a variable holding the size of the heap and a lock on either the first or last element of the heap. In order to increase parallelism, insertions traverse the heap bottom-up while deletions proceed top-down, without introducing deadlocks. Insertions also employ a left-right technique as in [11] to allow them to access opposite sides on the heap and thus minimize interference.

On a different note, Huang and Weihl [60] show a concurrent priority queue based on a concurrent version of Fibonacci Heaps [34].

Nonblocking linearizable heap-based priority queue algorithms have been proposed by Herlihy [50], Barnes [12], and Israeli and Rappoport [65]. Sundell and Tsigas [133] present a lock-free priority queue based on a lock-free version of Pugh's concurrent skiplist [112].

Tree-Based Priority Pools

Huang and Weihl [60] and Johnson [66] describe concurrent *priority pools*: priority queues with relaxed semantics that do not guarantee linearizability of the `delete-min` operations. Their designs are both based on a modified concurrent B^+-tree implementation. Johnson introduces a "delete bin" that accumulates values to be deleted and thus reduces the load when performing concurrent `delete-min` operations. Shavit and Zemach [130] show a similar pool based on Pugh's concurrent skiplist [112] with an added "delete bin" mechanism based on [66]. Typically, the weaker pool semantics allows for increased concurrency. In [130] they further show that if the size of the set of allowable keys is bounded (as is often the case in operating systems) a priority pool based on a binary tree of combining funnel nodes can scale to hundreds (as opposed to tens) of processors.

1.9 Summary

We have given an overview of issues related to the design of concurrent data structures for shared-memory multiprocessors, and have surveyed some of the important contributions in this area. Our overview clearly illustrates that the design of such data structures provides significant challenges, and as of this writing, the maturity of concurrent data structures falls well behind that of sequential data structures. However, significant progress has been made towards understanding key issues and developing new techniques to facilitate the design of effective concurrent data structures; we are particularly encouraged by renewed academic and industry interest in stronger hardware support for synchronization. Given new understanding, new techniques, and stronger hardware support, we believe significant advances in concurrent data structure designs are likely in the coming years.

Index

References

References

[1] U. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 2000.

[2] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[3] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[4] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. *Distributed Computing*, 8(4):163–169, 1995.

[5] B. Aiello, R. Venkatesan, and M. Yung. Coins, weights and contention in balancing networks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 193–214, August 1994.

[6] J. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[7] J. Anderson and J. Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, 1996.

[8] T. Anderson. The performance implications of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[9] N. Arora, R. Blumofe, and G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

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

[11] R. Ayani. LR-algorithm: concurrent operations on priority queues. In *In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 22–25, 1991.

[12] G. Barnes. Wait free algorithms for heaps. Technical Report TR-94-12-07, University of Washington, 1994.

[13] K. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[14] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1979.

[15] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[16] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[17] J. Biswas and J. Browne. Simultaneous update of priority structures. In *In Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, August 1987.

[18] G. Blelloch, P. Cheng, and P. Gibbons. Room synchronizations. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 122–133. ACM Press, 2001.

[19] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[20] J. Boyar and K. Larsen. Efficient rebalancing of chromatic search trees. *Journal of*

Computer and System Sciences, 49(3):667–682, December 1994.

[21] E.D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, October 1986.

[22] C. Busch and M. Herlihy. A survey on counting networks.

[23] C. Busch and M. Mavronicolas. A combinatorial treatment of balancing networks. *Journal of the ACM*, 43(5):794–839, September 1996.

[24] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.

[25] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.

[26] G. Della-Libera and N. Shavit. Reactive diffracting trees. *Journal of Parallel Distributed Computing*, 60(7):853–890, 2000.

[27] E. Dijkstra and C. Sholten. Termination detection for diffusing computations. *nformation Processing Letters*, 11(1):1–4, 1980.

[28] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[29] C. Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[30] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–355, September 1979.

[31] E. Felten, A. Lamarca, and R. Ladner. Building counting networks from larger balancers. Technical Report TR-93-04-09, University of Washington, 1993.

[32] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, pages 374–382, 1985.

[33] K. Fraser. *Practical Lock-Freedom*. Ph.D. dissertation, Kings College, University of Cambridge, Cambridge, England, September 2003.

[34] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[35] P. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2):733–769, 1998.

[36] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75. ACM Press, April 1989.

[37] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[38] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[39] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data structures using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[40] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *In Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978.

[41] S. Hanke. The performance of concurrent red-black tree algorithms. *Lecture Notes in Computer Science*, 1668:286–300, 1999.

[42] T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–??, 2001.

[43] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[44] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on DIStributed Computing*, pages 265–279, 2002.

[45] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.

[46] D. Hendler and N. Shavit. Work dealing. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 164–172, 2002.

[47] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. Technical Report TR-2004-128, Sun Microsystems Laboratories, 2004.

[48] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 0885-7458 1988.

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

[50] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[51] M. Herlihy, B. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995.

[52] M. Herlihy, B. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[53] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.

[54] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

[55] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, 2003.

[56] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[57] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

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

[59] M. Hsu and W. Yang. Concurrent operations in extendible hashing. In *Symposium on very large data bases*, pages 241–247, 1986.

[60] Q. Huang and W. Weihl. An evaluation of concurrent priority queue algorithms. In *IEEE Parralel and Distributed Computing Systems*, pages 518–525, 1991.

[61] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An efficient algorithm for

concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, November 1996.

[62] IBM. System/370 principles of operation. Order Number GA22-7000.

[63] IBM. Powerpc microprocessor family: Programming environments manual for 64 and 32-bit microprocessors, version 2.0, 2003.

[64] Intel. *Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual*, 1994.

[65] A. Israeli and L. Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *The 7th International Workshop on Distributed Algorithms*, pages 1–17, 1993.

[66] T. Johnson. A highly concurrent priority queue based on the b-link tree. Technical Report 91-007, University of Florida, August 1991.

[67] T. Johnson and D. Sasha. The performance of current b-tree algorithms. *ACM Transactions on Database Systems (TODS)*, 18(1):51–101, 1993.

[68] D. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.

[69] Y. Joung. The congenial talking philosophers problem in computer networks. *Distrib. Comput.*, 15(3):155–175, 2002.

[70] G. Kane. *MIPS RISC Architecture*. Prentice Hall, New York, 1989.

[71] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32. ACM Press, 1999.

[72] J. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26(11):895–901, November 1983.

[73] M. Klugerman and G. Plaxton. Small-depth counting networks. In *(STOC)*, pages 417–428, 1992.

[74] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1968.

[75] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 62–72. ACM Press, 1997.

[76] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, 1993. CRC Press.

[77] C. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[78] V. Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6):681–694, 1990.

[79] H. Kung and P. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Programming Languages and Systems*, 5:354–382, September 1980.

[80] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[81] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

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

[83] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Com-*

mun. ACM, 17(8):453–455, 1974.

[84] V. Lanin and D. Shasha. A symmetric concurrent b-tree algorithm. In *Proceedings of the Fall Joint Computer Conference 1986*, pages 380–389. IEEE Computer Society Press, November 1986.

[85] K. Larsen. Avl trees with relaxed balance. In *Eighth International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, April 1994.

[86] K. Larsen. Relaxed multi-way trees with group updates. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 93–101. ACM Press, 2001.

[87] K. Larsen. Relaxed red-black trees with group updates. *Acta Informatica*, 38(8):565–586, 2002.

[88] K. Larsen and R. Fagerberg. B-trees with relaxed balance. In *Ninth International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, April 1995.

[89] D. Lea. Concurrent hash map in JSR166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[90] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern.* Addison-Wesley, second edition edition, 1999.

[91] P. Lehman and S.B̃ing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[92] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases*, pages 212–223. IEEE Computer Society, October 1980.

[93] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, CT, 1987.

[94] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS)*, pages 165–171. IEEE Computer Society, April 1994.

[95] Udi Manber. On maintaining dynamic information in a concurrent environment. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 273–278. ACM Press, 1984.

[96] P. Martin, M. Moir, and G. Steele. Dcas-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.

[97] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[98] J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *SIGPLAN Not.*, 26(7):106–113, 1991.

[99] J. Metzger. Managing simultaneous operations in large ordered indexes. Technical report, Technische Universität München, Institut für Informatik, TUM-Math, 1975.

[100] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.

[101] M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *The 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

[102] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[103] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on

multiprogrammed shared - memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[104] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.

[105] C. Mohan and F. Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 371–380. ACM Press, 1992.

[106] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[107] Motorola. *MC68020 32-bit microprocessor user's manual*. Prentice-Hall, 1986.

[108] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 192–198. ACM Press, May 1991.

[109] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 170–176. ACM Press, 1987.

[110] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, second edition edition, 1997.

[111] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[112] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989.

[113] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[114] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, 2001.

[115] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2003.

[116] V. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37:1657–1665, December 1988.

[117] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, Cambridge, MA, 1986.

[118] M. Riedel and J. Bruck. Tolerating faults in counting networks.

[119] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *In Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245. ACM Press, July 1991.

[120] Y. Sagiv. Concurrent operations on b-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[121] B. Samadi. B-trees in a system with multiple users. *Information Processing Letter*, 5(4):107–112, October 1976.

[122] W. Savitch and M. Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM*, 26:103–118, 1979.

[123] M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of*

the twenty-first annual symposium on Principles of distributed computing, pages 31–40. ACM Press, 2002.

[124] M. Scott and J. Mellor-Crummey. Fast, contention-free combining tree barriers. *International Journal of Parallel Programming*, 22(4), August 1994.

[125] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 44–52, 2001.

[126] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. In *The 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.

[127] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[128] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[129] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[130] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM Press, 1999.

[131] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, 2000.

[132] R. Sites. *Alpha Architecture Reference Manual*, 1992.

[133] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 84a. IEEE press, 2003.

[134] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[135] L. Valiant. *Bulk-Synchronous Parallel Computers*. Wiley, New York, NY, 1989.

[136] J. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[137] D. Weaver and T.G̃ermond (Editors). *The SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, Englewood Cliffs, NJ), 1994.

[138] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.