DOI:10.1145/1897852.1897873

The advent of multicore processors as the standard computing platform will force major changes in software design.

BY NIR SHAVIT

Data Structures in the Multicore Age

"MULTICORE PROCESSORS ARE about to revolutionize the way we design and use data structures."

You might be skeptical of this statement; after all, are multicore processors not a new class of multiprocessor machines running parallel programs, just as we have been doing for more than a quarter of a century?

The answer is no. The revolution is partly due to changes multicore processors introduce to parallel architectures; but mostly it is the result of the change in the applications that are being parallelized: multicore processors are bringing parallelism to mainstream computing.

Before the introduction of multicore processors, parallelism was largely dedicated to computational problems with regular, slow-changing (or even static) communication and coordination patterns. Such problems arise in scientific computing or in graphics, but rarely in systems.

The future promises us multiple cores on anything from phones to laptops, desktops, and servers, and therefore a plethora of applications characterized by complex, fast-changing interactions and data exchanges.

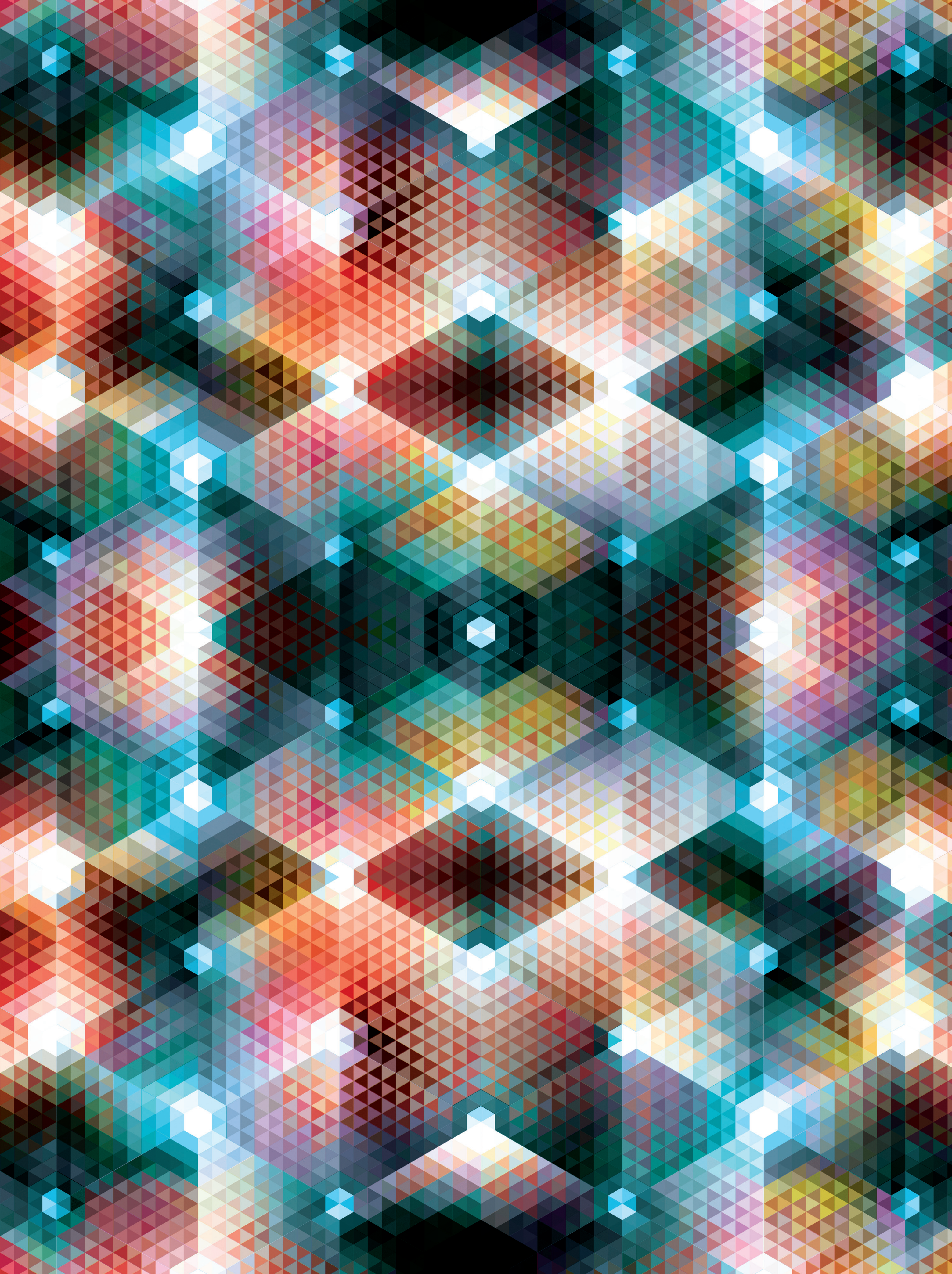
Why are these dynamic interactions and data exchanges a problem? The formula we need in order to answer this question is called *Amdahl's Law*. It captures the idea that the extent to which we can speed up any complex computation is limited by how much of the computation must be executed sequentially.

Define the *speedup* S of a computation to be the ratio between the time it takes one processor to complete the computation (as measured by a wall clock) versus the time it takes n concurrent processors to complete the same computation. Amdahl's Law characterizes the maximum speedup S that can be achieved by n processors collaborating on an application, where p is the fraction of the computation that can be executed in parallel. Assume, for simplicity, that it takes (normalized) time 1 for a single processor to complete the computation. With n concurrent processors, the parallel part takes time p/n, and the sequential part takes time $1-p$. Overall, the parallelized computation takes time $1-p+\frac{p}{n}$. Amdahl's Law says the speedup, that is, the ratio between

» key insights

- **We are experiencing a fundamental shift in the properties required of concurrent data structures and of the algorithms at the core of their implementation.**
- **The data structures of our childhood—stacks, queues, and heaps—will soon disappear, replaced by looser "unordered" concurrent constructs based on distribution and randomization.**
- **Future software engineers will need to learn how to program using these novel structures, understanding their performance benefits and their fairness limitations.**

ILLUSTRATION BY ANDY GILMORE



the sequential (single-processor) time and the parallel time, is:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

In other words, S does not grow linearly in n. For example, given an application and a 10-processor machine, Amdahl's Law says that even if we manage to parallelize 90% of the application, but not the remaining 10%, then we end up with a fivefold speedup, but not a 10-fold speedup. Doubling the number of cores to 20 will only raise us to a sevenfold speedup. So the remaining 10%, those we continue to execute sequentially, cut our utilization of the 10 processor machine in half, and limit us to a 10-fold speedup no matter how many cores we add.

What are the 10% we found difficult to parallelize? In many mainstream applications they are the parts of the program involving interthread interaction and coordination, which on multicore machines are performed by concurrently accessing shared data structures. Amdahl's Law tells us it is worthwhile to invest an effort to derive as much parallelism as possible from these 10%, and a key step on the way to doing so is to have highly parallel concurrent data structures.

Unfortunately, concurrent data structures are difficult to design. There is a kind of tension between correctness and performance: the more one tries to improve performance, the more difficult it becomes to reason about the resulting algorithm as being correct. Some experts blame the widely accepted threads-and-objects programming model (that is, threads communicating via shared objects), and predict its eventual demise will save us. My experience with the alternatives suggests this model is here to stay, at least for the foreseeable future. So let us, in this article, consider correctness and performance of data structures on multicore machines within the threads-and-objects model.

In the concurrent world, in contrast to the sequential one, correctness has two aspects: safety, guaranteeing that nothing bad happens, and liveness, guaranteeing that eventually something good will happen.

The safety aspects of concurrent data structures are complicated by the need to argue about the many possible interleavings of methods called by different threads. It is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting, where there are no interleavings. Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the structure's properties sequentially, and find a way to map its concurrent executions to these "correct" sequential ones. There are various approaches for doing this, called consistency conditions. Some familiar conditions are serializability, linearizability, sequential consistency, and quiescent consistency.

When considering liveness in a concurrent setting, the good thing one expects to happen is that method calls eventually complete. The terms under which liveness can be guaranteed are called progress conditions. Some familiar conditions are deadlock-freedom, starvation-freedom, lock-freedom, and wait-freedom. These conditions capture the properties an implementation requires from the underlying system scheduler in order to guarantee that method calls complete. For example, deadlock-free implementations depend on strong scheduler support, while wait-free ones do all the work themselves and are independent of the scheduler.

Finally, we have the performance of our data structures to consider. Historically, uniprocessors are modeled as Turing machines, and one can argue the theoretical complexity of data structure implementations on uniprocessors by counting the number of steps—the machine instructions—that method calls might take. There is an immediate correlation between the theoretical number of uniprocessor steps and the observed time a method will take.

In the multiprocessor setting, things are not that simple. In addition to the actual steps, one needs to consider whether steps by different threads require a shared resource or not, because these resources have a bounded capacity to handle simultaneous requests. For example, multiple instructions accessing the same location in memory cannot be serviced at the same time. In its simplest form, our theoretical complexity model requires us to consider a new element: stalls.[2,7–10] When threads concurrently access a shared resource, one succeeds and others incur stalls. The overall complexity of the algorithm, and hence the time it might take to complete, is correlated to the number of operations together with the number of stalls (obviously this is a crude model that does not take into account the details of cache coherence). From an algorithmic design point of view, this model introduces a continuum starting from centralized structures where all threads share data by accessing a small set of locations, incurring many stalls, to distributed structures with multiple locations, in which the number of stalls is greatly reduced, yet the number of steps necessary to properly share data and move it around increases significantly.

How will the introduction of multicore architectures affect the design of concurrent data structures? Unlike on uniprocessors, the choice of algorithm will continue, for years to come, to be greatly influenced by the underlying machine's architecture. In particular, this includes the number of cores, their layout with respect to memory and to each other, and the added cost of synchronization instructions (on a multiprocessor, not all steps were created equal).

However, I expect the greatest change we will see is that concurrent data structures will go through a substantiative "relaxation process." As the number of cores grows, in each of the categories mentioned, consistency conditions, liveness conditions, and the level of structural distribution, the requirements placed on the data structures will have to be relaxed in order to support scalability. This will put a burden on programmers, forcing them to understand the minimal conditions their applications require, and then use as relaxed a data structure as possible in the solution. It will also place a burden on data structure designers to deliver highly scalable structures once the requirements are relaxed.

This article is too short to allow a survey of the various classes of concurrent data structures (such a survey can be found in Moir and Shavit[17]) and how one can relax their definitions and implementations in order to make them scale. Instead, let us focus here on one abstract data structure—a stack—and use it as an example of how the design process might proceed.

I use as a departure point the acceptable sequentially specified notion of a `Stack<T>` object: a collection of items (of type `T`) that provides `push()` and `pop()` methods satisfying the *last-in-first-out* (LIFO) property: the last item pushed is the first to be popped.

We will follow a sequence of refinement steps in the design of concurrent versions of stacks. Each step will expose various design aspects and relax some property of the implementation. My hope is that as we proceed, the reader will grow to appreciate the complexities involved in designing a correct scalable concurrent data-structure.

A Lock-based Stack

We begin with a `LockBasedStack<T>` implementation, whose Java pseudocode appears in figures 1 and 2. The pseudocode structure might seem a bit cumbersome at first, this is done in order to simplify the process of extending it later on.

The lock-based stack consists of a linked list of nodes, each with `value` and `next` fields. A special `top` field points to the first list node or is *null* if the stack is empty. To help simplify the presentation, we will assume it is illegal to add a *null* value to a stack.

Access to the stack is controlled by a single *lock*, and in this particular case a *spin-lock*: a software mechanism in which a collection of competing threads repeatedly attempt to choose exactly one of them to execute a section of code in a mutually exclusive manner. In other words, the winner that acquired the lock proceeds to execute the code, while all the losers spin, waiting for it to be released, so they can attempt to acquire it next.

The lock implementation must enable threads to decide on a winner. This is done using a special synchronization instruction called a `compareAndSet()` (CAS), available in one form or another on all of today's mainstream multicore processors. The CAS operation executes a read operation followed by a write operation, on a given memory location, in one indivisible hardware step. It takes two arguments: an *expected* value and an *update* value. If the memory location's value is equal to the expected value, then it is replaced by the update value, and otherwise the value is left unchanged. The method call returns a Boolean indicating whether the value changed. A typical CAS takes significantly more machine cycles than a read or a write, but luckily, the performance of CAS is improving as new generations of multicore processors role out.

In Figure 1, the `push()` method creates a new node and then calls `tryPush()` to try to acquire the lock. If the CAS is successful, the lock is set to true and the method swings the top reference from the current top-of-stack to its successor, and then releases the lock by setting it back to *false*. Otherwise, the `tryPush()` lock acquisition attempt is repeated. The `pop()` method

Figure 1. A lock-based `Stack<T>`: in the `push()` method, threads alternate between trying to push an item onto the stack and managing contention by backing off before retrying after a failed push attempt.

```
public  class  LockBasedStack<T> {
  private AtomicBoolean lock =
          new AtomicBoolean(false);
  ...
  protected boolean tryPush(Node node) {
    boolean gotLock = lock.compareAndSet(false, true);
    if (gotLock)  {
     Node oldTop = top;
     node.next = oldTop;
     top = node;
     lock.set ( false );
     }
  return gotLock;
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
     if (tryPush(node)) {
       return;
     } else {
       contentionManager.backoff();
     }
  }
}
```

Figure 2. The lock-based `Stack<T>`: The `pop()` method alternates between trying to pop and backing off before the next attempt.

```
protected Node tryPop() throws EmptyException {
  boolean gotLock = lock.compareAndSet(false, true);
  if (gotLock) {
    Node oldTop = top;
    if  (oldTop == null) {
     lock . set ( false );
     throw new EmptyException();
    }
  top = oldTop.next;
  return oldTop;
  lock . set ( false );
  }
  else return null ;
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if  (returnNode != null) {
     return returnNode.value ;
    } else {
      contentionManager.backoff();
    }
  }
}
```

in Figure 2 calls tryPop(), which attempts to acquire the lock and remove the first node from the stack. If it succeeds, it throws an exception if the stack is empty, and otherwise it returns the node referenced by top. If tryPop() fails to acquire the lock it returns *null* and is called again until it succeeds.

What are the safety, liveness, and performance properties of our implementation? Well, because we use a single lock to protect the structure, it is obvious its behavior is "atomic" (the technical term used for this is *linearizable*[15]). In other words, the outcomes of our concurrent execution are equivalent to those of a sequential execution in which each push or pop take effect at some non-overlapping instant during their method calls. In particular, we could think of them taking effect when the executing thread acquired the lock. Linearizability is a desired property because linearizable objects can be composed without having to know anything about their actual implementation.

But there is a price for this obvious atomicity. The use of a lock introduces a dependency on the operating system: we must assume the scheduler will not involuntarily preempt threads (at least not for long periods) while they are holding the lock. Without such support from the system, all threads accessing the stack will be delayed whenever one is preempted. Modern operating systems can deal with these issues, and will have to become even better at handling them in the future.

In terms of progress, the locking scheme is deadlock-free, that is, if several threads all attempt to acquire the lock, one will succeed. But it is not starvation-free: some thread could be unlucky enough to always fail in its CAS when attempting to acquire the lock.

The centralized nature of the lock-based stack implementation introduces a sequential bottleneck: only one thread at a time can complete the update of the data structure's state. This, Amdahl's Law tells us, will have a very negative effect on scalability, and performance will not improve as the number of cores/threads increases.

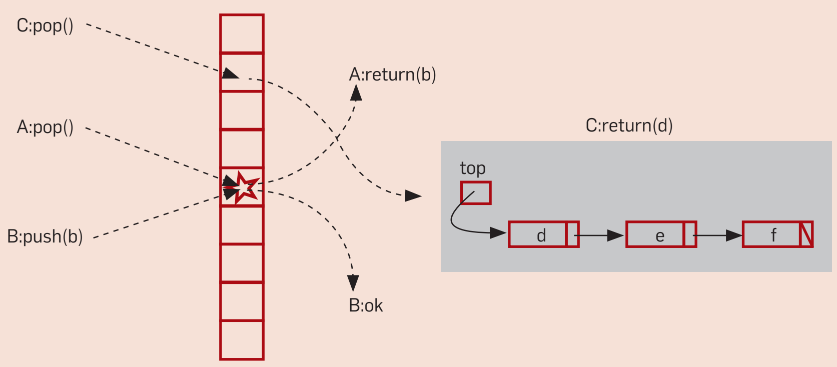
But there is another separate phenomenon here: memory contention. Threads failing their CAS attempts on the lock retry the CAS again even while the lock is still held by the last CAS "winner" updating the stack. These repeated attempts cause increased traffic on the machine's shared bus or interconnect. Since these are bounded resources, the result is an overall slowdown in performance, and in fact, as the number of cores increases, we will see performance deteriorate below that obtainable on a single core. Luckily, we can deal with contention quite easily by adding a *contention manager* into the code (Line 21 in figures 1 and 2).

The most popular type of contention manager is exponential backoff: every time a CAS fails in tryPush() or tryPop(), the thread delays for a certain random time before attempting the CAS again. A thread will double the range from which it picks the random delay upon CAS failure, and will cut it in half upon CAS success. The randomized nature of the backoff scheme makes the timing of the thread's attempts to acquire the lock less dependent on the scheduler, reducing the chance of threads falling into a repetitive pattern in which they all try to CAS at the same time and end up starving. Contention managers[1,12,19] are key tools in the design of multicore data structures, even when no locks are used, and I expect them to play an even greater role as the number of cores grows.

Figure 3. The lock-free tryPush() and tryPop() methods.

```
1   public  class   LockFreeStack<T>  {
2     private  AtomicReference<Node> top =
3                 new AtomicReference<Node>(null);
4   ...
5
6     protected boolean tryPush(Node node) {
7       Node oldTop = top.get();
8       node.next = oldTop;
9      return top.compareAndSet(oldTop, node);
10    }
11
12    protected Node tryPop() throws EmptyException  {
13      Node oldTop = top.get();
14      if (oldTop == null) {
15         throw new EmptyException();
16      }
17      Node newTop = oldTop.next;
18      if  (top.compareAndSet(oldTop, newTop)) {
19          return oldTop;
20      } else {
21          return null ;
22      }
23    }
```

Figure 4. The EliminationBackoffStack<T>.



Each thread selects a random location in the array. If thread *A*'s pop() and thread *B*'s push() calls arrive at the same location at about the same time, then they exchange values without accessing the shared lock-free stack. A thread *C*, that does not meet another thread, eventually pops the shared lock-free stack.

A Lock-Free Stack

As noted, a drawback of our lock-based implementation, and in fact, of lock-based algorithms in general, is that the scheduler must guarantee that threads are preempted infrequently (or not at all) while holding the locks. Otherwise, other threads accessing the same locks will be delayed, and performance will suffer. This dependency on the capriciousness of the scheduler is particularly problematic in hard real-time systems where one requires a guarantee on how long method calls will take to complete.

We can eliminate this dependency by designing a lock-free stack implementation.[23] In the `LockFreeStack<T>`, instead of acquiring a lock to manipulate the stack, threads agree who can modify it by directly applying a CAS to the `top` variable. To do so, we only need to modify the code for the `tryPush()` and `tryPop()` methods, as in Figure 3. As before, if unsuccessful, the method calls are repeated after backing off, just as in the lock-based algorithm.

A quick analysis shows the completion of a `push` (respectively `pop`) method call cannot be delayed by the preemption of some thread: the stack's state is changed by a single CAS operation that either completes or not, leaving the stack ready for the next operation. Thus, a thread can only be delayed by scheduling infinitely many calls that successfully modify the top of the stack and cause the `tryPush()` to continuously fail. In other words, the system as a whole will always make progress no matter what the scheduler does. We call this form of progress *lock-freedom*. In many data structures, having at least some of the structure's methods be lock-free tends to improve overall performance.

It is easy to see that the lock-free stack is linearizable: it behaves like a sequential stack whose methods "take effect" at the points in time where their respective CAS on the top variable succeeded (or threw the exception in case of a `pop` on an empty stack). We can thus compose this stack with other linearizable objects without worrying about the implementation details: as far as safety goes, there is no difference between the lock-based and lock-free stacks.

I expect the greatest change we will see is that concurrent data structures will go through a substantiative "relaxation process."

An Elimination Backoff Stack

Like the lock-based stack, the lock-free stack implementation scales poorly, primarily because its single point of access forms a *sequential bottleneck*: method calls can proceed only one after the other, ordered by successful CAS calls applied to the stack's lock or top fields. A sad fact we should acknowledge is this sequential bottleneck is inherent: in the worst case it takes a thread at least $\Omega(n)$ steps and/or stalls (recall, a stall is the delay a thread incurs when it must wait for another thread taking a step) to push or pop a linearizable lock-free stack.[9] In other words, the theory tells us there is no way to avoid this bottleneck by distributing the stack implementation over multiple locations; there will always be an execution of linear complexity.

Surprisingly, though, we can introduce parallelism into many of the common case executions of a stack implementation. We do so by exploiting the following simple observation: if a `push` call is immediately followed by a `pop` call, the stack's state does not change; the two calls *eliminate* each other and it is as if both operations never happened. By causing concurrent pushes and pops to meet and pair up in separate memory locations, the thread calling `push` can exchange its value with a thread calling `pop`, without ever having to access the shared lock-free stack.

As depicted in Figure 4, in the `EliminationBackoffStack<T>`[11] one achieves this effect by adding an `EliminationArray` to the lock-free stack implementation. Each location in the array is a coordination structure called an *exchanger*,[16,18] an object that allows a pair of threads to rendezvous and exchange values.

Threads pick random array entries and try to pairup with complementary operations. The calls exchange values in the location in which they met, and return. A thread whose call cannot be eliminated, either because it has failed to find a partner, or because it found a partner with the wrong type of method call (such as a `push` meeting a `push`), can either try again to eliminate at a new location, or can access the shared lock-free stack. The combined data structure, array and stack, is linearizable because the lock-free stack is linearizable, and we can think of the eliminated calls as if they occurred at the point in which they exchanged values.

It is lock-free because we can easily implement a lock-free exchanger using a CAS operation, and the shared stack itself is already lock-free.

In the `EliminationBackoffStack`, the `EliminationArray` is used as a backoff scheme to a shared lock-free stack. Each thread first accesses the stack, and if it fails to complete its call (that is, the CAS attempt on top fails) because there is contention, it attempts to eliminate using the array instead of simply backing off in time. If it fails to eliminate, it calls the lockfree stack again, and so on. A thread dynamically selects the subrange of the array within which it tries to eliminate, growing and shrinking it exponentially in response to the load. Picking a smaller subrange allows a greater chance of a successful rendezvous when there are few threads, while a larger range lowers the chances of threads waiting on a busy `Exchanger` when the load is high.

In the worst case a thread can still fail on both the stack and the elimination. However, if contention is low, threads will quickly succeed in accessing the stack, and as it grows, there will be a higher number of successful eliminations, allowing many operations to complete in parallel in only a constant number of steps. Moreover, contention at the lock-free stack is reduced because eliminated operations never access the stack. Note that we described a lock-free implementation, but, as with many concurrent data structures, on some systems a lock-based implementation might be more fitting and deliver better performance.

An Elimination Tree

A drawback of the elimination backoff stack is that under very high loads the number of un-eliminated threads accessing the shared lock-free stack may remain high, and these threads will continue to have linear complexity. Moreover, if we have, say, bursts of `push` calls followed by bursts of `pop` calls, there will again be no elimination and therefore no parallelism. The problem seems to be our insistence on having a linearizable stack: we devised a distributed solution that cuts down on the number of stalls, but the theoretical worst case linear time scenario can happen too often.

This leads us to try an alternative approach: relaxing the consistency condition for the stack. Instead of a linearizable stack, let's implement a quiescently consistent one.[4,14] A stack is quiescently consistent if in any execution, whenever there are no ongoing `push` and `pop` calls, it meets the LIFO stack specification for all the calls that preceded it. In other words, quiescent consistency is like a game of musical chairs, we map the object to the sequential specification when and only when the music stops. As we will see, this relaxation will nevertheless provide quite powerful semantics for the data structure. In particular, as with linearizability, quiescent consistency allows objects to be composed as black boxes without having to know anything about their actual implementation.

Consider a binary tree of objects called *balancers* with a single input wire and two output wires, as depicted in Figure 5. As threads arrive at a balancer, it repeatedly sends them to the top wire and then the bottom one, so its top wire always has one more thread than the bottom wire. The `Tree[k]` network is a binary tree of balancers constructed inductively by placing a balancer before two `Tree[k/2]` networks of balancers and not shuffling their outputs.[22]

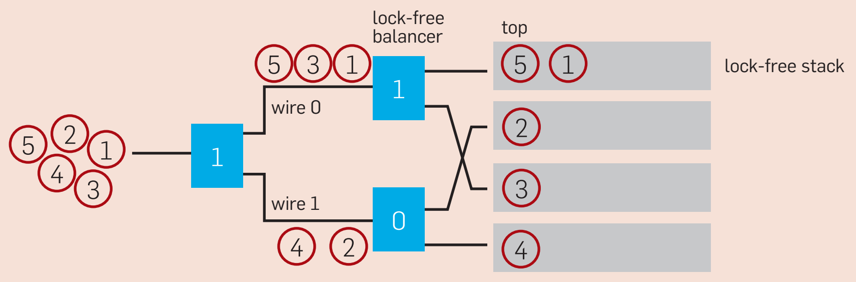
We add a collection of lock-free stacks to the output wires of the tree. To perform a `push`, threads traverse the balancers from the root to the leaves and then `push` the item onto the appropriate stack. In any quiescent state, when there are no threads in the tree, the output items are balanced out so that the top stacks have at most one more than the bottom ones, and there are no gaps.

We can implement the balancers in a straightforward way using a bit that threads toggle: they fetch the bit and then complement it (a CAS operation), exiting on the output wire they fetched (zero or one). How do we perform a pop? Magically, to perform a pop threads traverse the balancers in the opposite order of the push, that is, in each balancer, after complementing the bit, they follow this complement, the opposite of the bit they fetched. Try this; you will see that from one quiescent state to the next, the items removed are the last ones pushed onto the stack. We thus have a collection of stacks that are accessed in parallel, yet act as one quiescent LIFO stack.

The bad news is that our implementation of the balancers using a bit means that every thread that enters the tree accesses the same bit in the root balancer, causing that balancer to become a bottleneck. This is true, though to a lesser extent, with balancers lower in the tree.

We can parallelize the tree by exploiting a simple observation similar to one we made about the elimination backoff stack:

Figure 5. A `Tree[4]` network leading to four lock-free stacks.



Threads pushing items arrive at the balancers in the order of their numbers, eventually pushing items onto the stacks located on their output wires. In each balancer, a pushing thread fetches and then complements the bit, following the wire indicated by the fetched value (If the state is 0 the pushing thread it will change it to 1 and continue to wire 0, and if it was 1 will change it to 0 and continue on wire 1). The tree and stacks will end up in the balanced state seen in the figure. The state of the bits corresponds to 5 being the last item, and the next location a pushed item will end up on is the lock-free stack containing item 2. Try it! A popping thread does the opposite of the pushing one: it complements the bit and follows the complemented value. Thus, if a thread executes a pop in the depicted state, it will end up switching a 1 to a 0 at the top balancer, and leave on wire 0, then reach the top 2nd level balancer, again switching a 1 to a 0 and following its 0 wire, ending up popping the last value 5 as desired. This behavior will be true for concurrent executions as well: the sequences of values in the stacks in all quiescent states can be shown to preserve LIFO order.

If an *even* number of threads passes through a balancer, the outputs are evenly balanced on the top and bottom wires, but the balancer's state remains unchanged.

The idea behind the `EliminationTree<T>`[20,22] is to place an `EliminationArray` in front of the bit in every balancer as in Figure 6. If two popping threads meet in the array, they leave on opposite wires, without a need to touch the bit, as anyhow it would have remained in its original state. If two pushing threads meet in the array, they also leave on opposite wires. If a push or pop call does not manage to meet another in the array, it toggles the bit and leaves accordingly. Finally, if a push and a pop meet, they eliminate, exchanging items as in the `EliminationBackoffStack`. It can be shown that this implementation provides a quiescently consistent stack,[a] in which, in most cases, it takes a thread $O(\log k)$ steps to complete a push or a pop, where k is the number of lock-free stacks on its output wires.

A Pool Made of Stacks

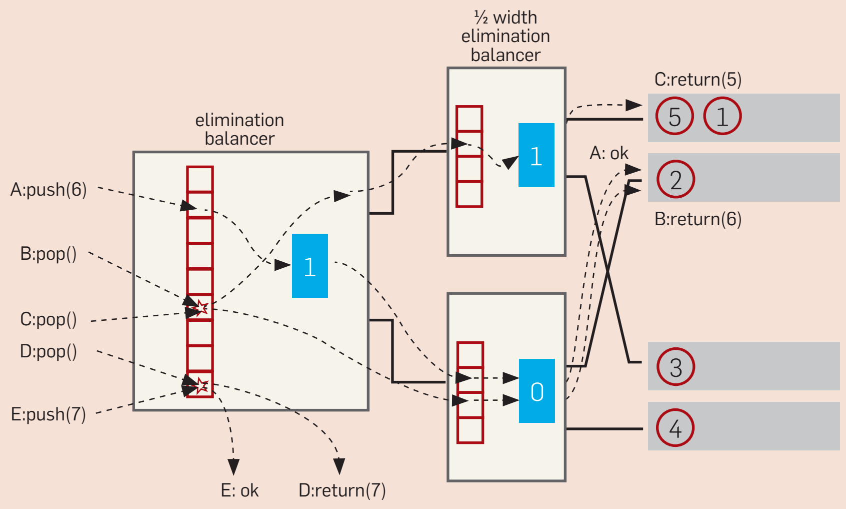
The collection of stacks accessed in parallel in the elimination tree provides quiescently consistent LIFO ordering with a high degree of parallelism. However, each method call involves a logarithmic number of memory accesses, each involving a CAS operation, and these accesses are not localized, that is, threads are repeatedly accessing locations they did not access recently.

This brings us to the final two issues one must take into account when designing concurrent data structures: the machine's memory hierarchy and its coherence mechanisms. Mainstream multicore architectures are cache coherent, where on most machines the L2 cache (and in the near future the L3 cache as well) is shared by all cores. A large part of the machine's performance on shared data is derived from the threads' ability to find the data cached. The shared caches are unfortunately a bounded resource, both in their size and in the level of access parallelism they offer. Thus, the data structure design needs to attempt to lower the overall number of accesses to memory, and to maintain locality as much as possible.

What are the implications for our stack design? Consider completely relaxing the LIFO property in favor of a `Pool<T>` structure in which there is no temporal ordering on `push()` and `pop()` calls. We will provide a concurrent lock-free implementation of a pool that supports high parallelism, high locality, and has a low cost in terms of the overall number of accesses to memory. How useful is such a concurrent pool? I would like to believe that most concurrent applications can be tailored to use pools in place of queues and stacks (perhaps with some added liveness conditions)…time will tell.

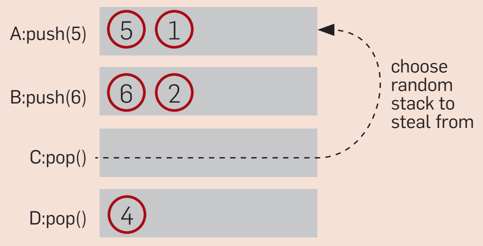
Our overall concurrent pool design is quite simple. As depicted in Figure 7, we allocate a collection of n concurrent lock-free stacks, one per computing thread (alternately we could allocate one stack per collection of threads on the same core, depending on the specific machine architecture). Each thread will push and pop from its own assigned stack. If, when it attempts to pop, it finds its own stack is empty, it will repeatedly attempt to "steal" an item from another randomly chosen stack.[b] The pool has, in

Figure 6. The `EliminationTree<T>`.



Each balancer in `Tree[4]` is an elimination balancer. The state depicted is the same as in Figure 5. From this state, a push of item 6 by thread A will not meet any others on the elimination arrays and so will toggle the bits and end up on the 2nd stack from the top. Two pops by threads B and C will meet in the top balancer's array and end up going up and down without touching the bit, ending up popping the last two values 5 and 6 from the top two lock-free stacks. Finally, threads D and E will meet in the top array and "eliminate" each other, exchanging the value 7 and leaving the tree. This does not ruin the tree's state since the states of all the balancers would have been the same even if the threads had both traversed all the way down without meeting: they would have anyhow followed the same path down and ended up exchanging values via the same stack.

Figure 7. The concurrent `Pool<T>`.



Each thread performs `push()` and `pop()` calls on a lock-free stack and attempts to steal from other stacks when a `pop()` finds the local stack empty. In the figure, thread *C* will randomly select the top lock-free stack, stealing the value 5. If the lock-free stacks are replaced by lock-free deques, thread *C* will pop the oldest value, returning 1.

a To keep things simple, pop operations should block until a matching push appears.

the common case, the same $O(1)$ complexity per method call as the original lockfree stack, yet provides a very high degree of parallelism. The act of stealing itself may be expensive, especially when the pool is almost empty, but there are various techniques to reduce the number of steal attempts if they are unlikely to succeed. The randomization serves the purpose of guaranteeing an even distribution of threads over the stacks, so that if there are items to be popped, they will be found quickly. Thus, our construction has relaxed the specification by removing the causal ordering on method calls and replacing the deterministic liveness and complexity guarantees with probabilistic ones.

As the reader can imagine, the $O(1)$ step complexity does not tell the whole story. Threads accessing the pool will tend to pop items that they themselves recently pushed onto their own designated stack, therefore exhibiting good cache locality. Moreover, since chances of a concurrent stealer are low, most of the time a thread accesses its lock-free stack alone. This observation allows designers to create a lockfree "stack-like" structure called a `Deque`[c] that allows the frequently accessing local thread to use only loads and stores in its methods, resorting to more expensive CAS based method calls only when chances of synchronization with a conflicting stealing thread are high.[3,6]

The end result is a pool implementation that is tailored to the costs of the machine's memory hierarchy and synchronization operations. The big hope is that as we go forward, many of these architecture-conscious optimizations, which can greatly influence performance, will move into the realm of compilers and concurrency libraries, and the need for everyday programmers to be aware of them will diminish.

What Next?

The pool structure ended our sequence of relaxations. I hope the reader has come to realize how strongly the choice of structure depends on the machine's size and the application's concurrency requirements. For example, small collections of threads can effectively share a lock-based or lock-free stack, slightly larger ones an elimination stack, but for hundreds of threads we will have to bite the bullet and move from a stack to a pool (though within the pool implementation threads residing on the same core or machine cluster could use a single stack quite effectively).

In the end, we gave up the stack's LIFO ordering in the name of performance. I imagine we will have to do the same for other data structure classes. For example, I would guess that search structures will move away from being comparison based, allowing us to use hashing and similar naturally parallel techniques, and that priority queues will have a relaxed priority ordering in place of the strong one imposed by deleting the minimum key. I can't wait to see what these and other structures will look like.

As we go forward, we will also need to take into account the evolution of hardware support for synchronization. Today's primary construct, the CAS operation, works on a single memory location. Future architectures will most likely support synchronization techniques such as transactional memory,[13,21] allowing threads to instantaneously read and write multiple locations in one indivisible step. Perhaps more important than the introduction of new features like transactional memory is the fact that the relative costs of synchronization and coherence are likely to change dramatically as new generations of multicore chips role out. We will have to make sure to consider this evolution path carefully as we set our language and software development goals.

Concurrent data structure design has, for many years, been moving forward at glacial pace. Multicore processors are about to heat things up, leaving us, the data structure designers and users, with the interesting job of directing which way they flow. Let's try to get it right.

c This `Deque` supports `push()` and `pop()` methods with the traditional LIFO semantics and an additional `popTop()` method for stealers that pops the first-in (oldest) item.[5]

References

1. Agarwal, A. and Cherian, M. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture* (May 1989), 396–406.
2. Anderson, J. and Kim, Y. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing* (2001), 90–99.
3. Arora, N.S., Blumofe, R.D. and Plaxton, C.G. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems 34*, 2 (2001), 115–144.
4. Aspnes, J., Herlihy, M. and Shavit, N. Counting networks. *J. ACM 41*, 5 (1994), 1020–1048.
5. Blumofe, R.D. and Leiserson, C.E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.
6. Chase, D. and Lev, Y. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2005). ACM Press, NY, 21–28.
7. Cypher, R. The communication requirements of mutual exclusion. In *ACM Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures* (1995), 147-156.
8. Dwork, C., Herlihy, M. and Waarts, O. Contention in shared memory algorithms. *J. ACM 44*, 6 (1997), 779–805.
9. Fich, F.E., Hendler, D. and Shavit, N. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science* (2005).IEEE Computer Society, Washington, D.C., 165–173.
10. Gibbons, P.B., Matias, Y. and Ramachandran, V. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Computing 28*, 2 (1999), 733–769.
11. Hendler, D., Shavit, N. and Yerushalmi, L. A scalable lock-free stack algorithm. *J. Parallel and Distributed Computing 70*, 1 (Jan. 2010), 1–12.
12. Herlihy, M., Luchangco, V., Moir, M. and Scherer III, W.N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. ACM, NY, 2003, 92–101.
13. Herlihy, M. and Moss, E. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News 21*, 2 (1993), 289–300.
14. Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, San Mateo, CA, 2008.
15. Herlihy, M. and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems 12*, 3 (July 1990), 463–492.
16. Moir, M., Nussbaum, D., Shalev, O. and Shavit, N. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM Press, NY, 2005, 253–262.
17. Moir, M. and Shavit, N. Concurrent data structures. *Handbook of Data Structures and Applications*, D. Metha and S. Sahni, eds. Chapman and Hall/CRC Press, 2007, 47-14, 47-30.
18. Scherer III, W.N., Lea, D. and Scott, M.L. Scalable synchronous queues. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, NY, 2006, 147–156.
19. Scherer III, W.N. and Scott, M.L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*. ACM, NY, 2005, 240–248.
20. Shavit, N. and Touitou, D. Elimination trees and the construction of pools and stacks. Theory of Computing Systems 30 (1997), 645–670.
21. Shavit, N. and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (Feb. 1997), 99–116.
22. Shavit, N. and Zemach, A. Diffracting trees. *ACM Transactions on Computer Systems 14*, 4 (1996), 385–428.
23. Treiber, R.K. Systems programming: Coping with parallelism. Technical Report RJ 5118 (Apr. 1986). IBM Almaden Research Center, San Jose, CA.

Nir Shavit is a professor of computer science at Tel-Aviv University and a member of the Scalable Synchronization Group at Oracle Labs. He is a recipient of the 2004 ACM/EATCS Gödel Prize.