Tel Aviv University

The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Building Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees

A Thesis submitted for the degree

Master of Computer Science

by

Maria Natanzon

Supervised by

Professor Yehuda Afek, Professor Nir Shavit

July 2010

CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. Yehuda Afek and Prof. Nir Shavit , for their guidance and support during my research. I would also like to thank Mr. Guy Korland, for guiding and assisting me throughout every stage of my work.

This thesis paper was supported in part by grants from Sun Microsystems, Intel Corporation, as well as a grant 06/1344 from the Israeli Science Foundation and European Union grant FP7-ICT-2007-1 (project VELOX).

ABSTRACT

Producer-consumer pools, that is, collections of unordered objects or tasks, are a fundamental element of modern multiprocessor software and a target of extensive research and development. For example, there are three common ways to implement such pools in the Java JDK6.0: the *SynchronousQueue*, the *LinkedBlockingQueue*, and the *ConcurrentLinkedQueue*. Unfortunately, most pool implementations, including the ones in the JDK, are based on centralized structures like a queue or a stack, and thus are limited in their scalability.

This thesis presents the *ED-Tree*[1], a distributed pool structure based on a combination of the elimination-tree and diffracting-tree paradigms, allowing high degrees of parallelism with reduced contention. We use the ED-Tree to provide new pool implementations that compete with those of the JDK.

In experiments on a 128 way Sun Maramba multicore machine, we show that ED-Tree based object pools scale well, outperforming the corresponding algorithms in the JDK6.0 by a factor of 10 or more at high concurrency levels, while providing similar performance at low levels.

This thesis is based on the paper "Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees" of Afek, Korland, Natanzon and Shavit, presented on EURO-PAR 2010[1].

1. INTRODUCTION

Producer-consumer pools[5], that is, collections of unordered objects or tasks, are a fundamental element of modern multiprocessor software and a target of extensive research and development.

Pools show up in many places in concurrent systems. For example, in many applications, one or more *producer* threads produce items to be consumed by one or more *consumer* threads. These items may be jobs to perform, keystrokes to interpret, purchase orders to execute, or packets to decode. Another type of widely used pools are "resource pools", when allocated items are not destroyed but recycled after their usage in order to be used again.

Pools are accessed by two types of entities: Producers and Consumers and suggest two main types of operations: *Insert* (performed by Producers) and *Remove* (Performed by consumers). These operations can obey different semantics: can be blocking or non blocking, depends on the type of the pool definition. When Insert is non blocking, producer puts it's object in the pool and leaves, whereas when it's blocking, producer is obliged to block and wait until some consumer removes his object, and only then is free lo leave. Similarly, when Remove operation is not blocking, consumer collects an object and leaves, returning a null value in case the pool is empty, whereas blocking Consumer will block and wait till an object becomes available.

There are several ways to implement such pools. In the Java JDK6.0 for example they are called "queues": the *SynchronousQueue*, the *LinkedBlockingQueue*, and the *ConcurrentLinkedQueue*. The synchronous queue provides a "pairing up" function without buffering; it is entirely symmetric: Producers and consumers wait for one another, rendezvous, and leave in pairs, i.e. both Insert and Remove operations are defined as blocking. The other queues provide a buffering mechanism and allow threads to sleep while waiting for their requests to be fulfilled. LinkedBlocking queue implements blocking consumers and non blocking producers, in ConcurentLinkedQueue both producers and consumers are non blocking. Unfortunately, all these pools, including the new scalable synchronous queues of Lea, Scott, and Shearer [7], are based on centralized structures like a lock-free queue or a stack, and thus are limited in their scalability: the head of the stack or queue is a sequential bottleneck and source of contention. In more detail, since queues and stacks enforce LIFO or FIFO semantics, each operation on such data structure must pass through it's head and "register" it's request to obey a certain order. Thus, when used in multi-threaded environment, threads that access the data structure in parallel will create a contention on the head, while each waiting it's turn to access it.

Pools on the other hand, by their definition, don't have to obey a certain order semantics, items can be inserted and removed by any possible order. This work shows how to overcome previously mentioned limitation by devising highly distributed pools based on an *ED-Tree*, a combined variant of the diffracting-tree structure of Shavit and Zemach [12] and the elimination-tree structure of Shavit and Touitou [10]. The ED-Tree does not have a central place through which all threads pass, and thus allows both parallelism and reduced contention. As we explain in Chapter 3, an ED-Tree uses randomization to distribute the concurrent requests of threads onto many locations so that they collide with one another and can exchange values. It has a specific combinatorial structure called a counting tree [12, 2], that allows requests to be properly

distributed even if such successful exchanges did not occur.

We use a diffracting tree as basic construction of the ED-tree, adding queues at the leaves of the tree so that requests are either matched up or end up properly distributed on the queues at the tree leaves. By properly distributed we mean that requests that do not eliminate always end up in the queues: the collection of all the queues together has the behavior of one large queue. Since the nodes of the tree will form a bottleneck if one uses the naive implementation of diffracting tree, we replace them with highly distributed nodes that use elimination and diffraction on randomly chosen array locations.

The elimination and diffraction tree structures were each proposed years ago [12, 10] and claimed to be effective through simulation [6]. A single level of an elimination array was also used in implementing shared concurrent stacks and queues [4, 9]. However, elimination trees and diffracting trees were never used to implement real world structures. This is mostly due the fact that there was no need for them: machines with a sufficient level of concurrency and low enough interconnect latency to benefit from them did not exist. Today, multicore machines present the necessary combination of high levels of parallelism and low interconnection costs. Indeed, this thesis work is the first to show that ED-Tree based implementations of data structures from the *java.util.concurrent* scale impressively on a real machine (a Sun Maramba multicore machine with 2x8 cores and 128 hardware threads), delivering throughput that at high concurrency levels is 10 times that of the existing JDK6.0 algorithms.

1.1 ED-Tree and Low Concurrency Levels

But what about low concurrency levels? In their elegant paper describing the JDK6.0 synchronous queue, Lea, Scott, and Shearer [7], suggest that using elimination techniques may indeed benefit the design of synchronous queues at high loads. However, they wonder whether the benefits of reduced contention achievable by using elimination under high loads, can be made to work at lower levels of concurrency because of the possibility of threads not meeting in the array locations.

This thesis paper shows that elimination and diffraction techniques can be combined to work well at both high and low loads. There are two main techniques that our ED-Tree implementation uses to make this happen. The first is to have each thread adaptively choose an exponentially varying array range from which it randomly picks a location(backoff in space), and the duration it will wait for another thread at that location. This means that, without coordination, threads will tend to map into a smaller array range as the load decreases, thus increasing chances of a collision. The second component is the introduction of diffraction for colliding threads that do not eliminate because they are performing the same type of operation. The diffraction mechanism allows threads to continue down the tree at a low cost. By the described techniques we upgrade our algorithm to adopt itself to current system state, whether it suggests high or low load. The end result is an ED-Tree structure, that, as our empirical testing shows, performs well at both high and low concurrency levels.

2. ELIMINATION AND DIFFRACTION

Before explaining how the ED-Tree works, let us review its predecessors, the *diffracting tree*, and *elimination tree* and the use of elimination and diffraction techniques in building scalable data structures.

2.1 Diffracting Trees

Diffracting trees [12] were designed to provide shared counting and load balancing in a distributed parallel environment, and originated from Counting networks[2].

Consider a binary tree of objects called *balancers* with a single input wire and two output wires, as depicted in Figure 2.1. Threads arrive at a balancer and it repeatedly sends them up and down, so its top wire always has the same or at most one more than the bottom one. The *Tree[k]* network of width k is a binary tree of balancers constructed inductively by taking two *Tree[k/2]* networks of balancers and perfectly shuffling their outputs [12].

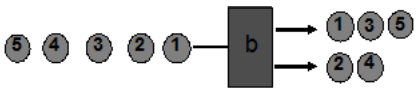

Fig. 2.1: A *Balancer*.

A tree constructed in that manner maintains the *step property*: In any quiescent state, i.e. when every thread that entered the tree arrived at it's output wire, and there are no threads in inner levels of the tree, the following is correct: if we call the output wires of the tree $y_1 \ldots y_n$ then either $y_i = y_j$ for all i, j, or there exists some c such that for any $i < c$ and $j \leq c$, $y_i - y_j = 1$. [12] In other words, the upper output wires will always have the same number of exited threads as the bottom ones, or at most one more. Thus, the diffracting three spreads the threads in a uniform manner across the output wires. See Figure 2.2

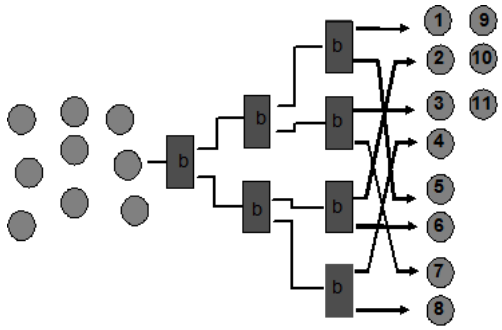

Fig. 2.2: A diffracting tree obeys the step property: in a quiescent state the upper output wires will always have the same number of exited threads as the bottom ones.

One could 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).

The bad news is that the above implementation of the balancers using a bit means that every thread that enters the tree accesses the same bit at 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:

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

Thus, if one could have pairs of tokens collide and then diffract in a coordinated manner, one to the top wire and one to the bottom wire both could leave the balancer without even having to toggle the shared bit.This bit will only be accessed by processors that did not collide. This approach can be implemented by adding a software *prism* in front of the toggle bit of every balancer. The prism is an inherently distributed data structure that allows many diffractions to occur in parallel. When a thread T enters the balancer, it first selects a location, l, in prism uniformly at random. T tries to "collide" with the previous token to select l or, by waiting for a fixed time, with the next token to do so. If a collision occurs, both tokens leave the balancer on separate wires; otherwise the undiffracted token T toggles the bit and leaves accordingly. Diffracting trees thus provide high degree of parallelism, allowing many threads reach their destination without passing spots of contention.

Although diffracting trees in many cases solve the contention problem, they have some parameters that are sensitive to the number of threads that access the data structure in parallel [11]. One such parameter is the depth of the tree (number of tree levels). If the tree is too small, it will be overloaded, bringing contention and less parallelism that possible. If it's too deep, the counters at its leafs will not be fully utilized, achieving less than optimal throughput. Another sensitive parameter is the the prism width, i.e. the total number of prism locations in the balancers at a given level of the tree. This parameter affects the chances of a successful pairing-off. If the prism is too large then threads will tend to miss each other, failing to pair-off and causing contention on the toggle bit. If it's too small, contention and sequential bottlenecking will occur as too many threads will be trying to access the same prism locations at the same time.

Thus using a diffracting tree to construct data structure make the data structure sensitive to the system load. therefore additional techniques must be used to force the algorithm to adopt itself to the current level of contention.

One such technique was suggested by Della-Libera and Shavit in the paper "Reactive diffracting trees" [3]. The reactive diffracting tree is a diffracting tree which can grow and shrink to better handle changing access patterns. In such tree, each node will act like a counter(i.e. when reached, the traversal of the tree stops even if this node is not a leaf) or a regular sub tree. The decision about each node is made periodically, according to current system load. In cases of high concurrency, all the nodes will be "unfolded", allowing high distribution of the threads, in cases of low loads, the tree will "shrink", sparing the high overhead of traversing through all the tree levels. The main drawback of such algorithm is that each node has to maintain its current state, which again, returns us to centralized location in memory, everyone has to access, making it a hot-spot and reducing concurrency.

2.2 Elimination

Elimination is a state when two threads carrying a *token* and *anti-token* meet, and "eliminate" each other [10]. For example a producer thread can be defined as one carrying a token, accordingly a consumer is carrying an anti token. When such two threads meet in a data structure, they can exchange the information they carry.

A common way to use elimination is to build an *elimination array*. Similarly to the prism array presented in previous section, elimination array is a set of cells, where each thread randomly chooses a location and spins waiting another thread to "collide" with. When collision occurred, in case the two collided threads carry a token and anti-token, they can exchange information and leave the data structure.

In their paper about the construction of elimination trees [10],Shavit and Touito suggest to combine a diffracting tree with the elimination paradigm. If, while traversing the tree, a producer thread collides with a consumer thread in one of the prism arrays, the producer will hand out its item to the consumer and they both can stop their activity on the tree. Thus, threads that are of opposite types eliminate each other, without reaching the bottom of the tree. Elimination was used since then in construction of many concurrent data structures. Various types of concurrent queues and stacks combine elimination techniques.

In the paper about concurrent stacks construction [4] of Hendler, Shavit and Yerushalmi, elimination array is used in combination with a lock free stack to design a concurrent non-blocking, linearizable stack. A single elimination array is put in front of a lock free stack. Each thread that performs operation on the stack, first approaches the elimination array and tries to pair up with an opposite type thread. If a removing and an inserting thread meet, one takes the item of the other accordingly and they leave. Otherwise the thread continues to the lock free stack. This way a stack semantics is preserved, while introducing highly concurrent behavior at high loads. A similar technique was used later to construct a concurrent linearizable queue[9].

The main drawback of all the listed algorithms is that while they provide good performance in high load, in low loads the additional data structures like the elimination array or the elimination tree provide unnecessary overhead and have negative effect on the the performance. Additionally, similarly to the diffracting trees, elimination algorithms use parameters that are sensitive to a system concurrency level. For best performance, the size of elimination array should be affected by the number of threads concurrently accessing it. The length of the spin period, when one thread waits to pair-off with another, before it decides to move on is also a sensitive parameter, if it's too small, it will cause contention, but if it's too big threads will waste their time spinning when the chance to collision is small. Various techniques were suggested to handle those issues, as backoff in time and space etc. We adopted some of those techniques to make our construction adaptive to a system load.

3. THE ED-TREE

The idea behind the ED-Tree is combining the modified diffracting [12] tree as above with the elimination-tree techniques [10].

As a first step in constructing the ED-Tree we add to the diffracting tree a collection of lock-free queues at the output wires of the tree leaves as shown in Figure 3.1. Since the nodes of the tree will form a bottleneck if one uses the naive implementation of diffracting tree, we put prism arrays in each balancer as explained in Section 2.1. To perform an insert operation, producer threads traverse the balancers from the root to the leaves and then inserts the item onto the appropriate queue. In any quiescent state, when there are no threads in the tree, the output items are balanced out so that the top queues have at most one more element than the bottom ones, and there are no gaps. That way we get a structure where all requests end up properly distributed on the queues at the tree leaves, and the collection of all the queues together has the behavior of one large queue.

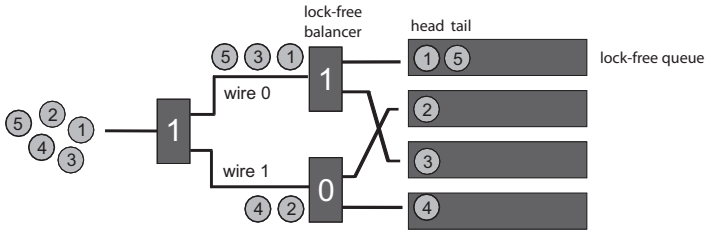

Fig. 3.1: A *Tree[4]* [12] leading to 4 lock-free queues. Threads inserting items arrive at the balancers in the order of their numbers, eventually inserting items onto the queues located on their output wires. In each balancer, a producer thread fetches and then complements the bit, following the wire indicated by the fetched value (If the state is 0 the producer thread it will change it to 1 and continue to up wire (wire 0), and if it was 1 will change it to 0 and continue to bottom wire (wire 1)). The tree and queues will end up in the balanced state seen in the figure. The state of the bits corresponds to 5 being the last inserted item, and the next location an inserted item will end up on is the queue containing item 2. Try it! We can add a similar tree structure for removing threads, so that the first will end up on the top queue, removing 1, and so on. This behavior will be true for concurrent executions as well: the sequences values in the queues in all quiescent states, when all threads have exited the structure, can be shown to preserve FIFO order.

One can keep a second, identical tree for consumers, where the output wires of the tree reference the same queues as the first tree, and you will see that from one quiescent state to the next, the items removed are the first ones inserted onto the queue. We thus have a collection of queues that are accessed in parallel, yet act as one quiescent FIFO queue.

The second step is to merge the two trees to a single tree. To remove or insert an item a thread enters the balancer and tries to diffract using the prism array. If diffraction succeeds it moves to the next level, otherwise it approaches the toggle bit and advances to the next level.

Next we insert the elimination technique to the our construction. We use the

observation, that if a producer and a consumer "collided" in a prism array any time during their tree traversal, there is no need for them to continue traversing the tree, as they can fulfill each other's request. The producer will hand his item to the consumer, and each can leave the tree and return.

To achieve that, we replace each node of the tree with highly distributed nodes that use elimination and diffraction on randomly chosen array locations. Every prism in the tree is turned to "elimination prism", which allows not only diffraction, but also elimination, meaning that when two threads collide in a cell of such an array, they decide to eliminate or diffract according to the "partner's" type.

We put an EDArray in front of the toggle bits in every balancer. If two removing threads meet in the array, they leave on opposite wires, without a need to touch the bit, as anyhow it would remain in its original state. If two inserting threads meet in the array, they leave on opposite wires at the same manner. If an insert or remove call does not manage to meet another in the array, it toggles the toggle bit and leaves accordingly. Finally, if a producer and a consumer meet, they eliminate, exchanging items. Threads that were "unlucky" to find an elimination partner all the way down the tree and reach the leafs, approach the queue referenced by the leaf. As we show later, in contended cases the percentage of such threads is very low, i.e, most of the threads eliminate successfully in the tree without reaching the queues at the output wires.

Our last modification to the tree, is adding an additional toggle bit to each node in the tree. That way each balancer will maintain an elimination prism array and two toggle bits, one for producer threads and one for consumer threads, thus allowing "inserting" and "removing" threads follow each other and not going to different directions if they weren't diffracted. In this sense it differs from prior Elimination and/or Diffraction balancer algorithms [12, 10] which had a single toggle bit instead of separate ones, and provided LIFO rather than FIFO like access through the bits. Figure 3.2 presents the final ED-tree construction.

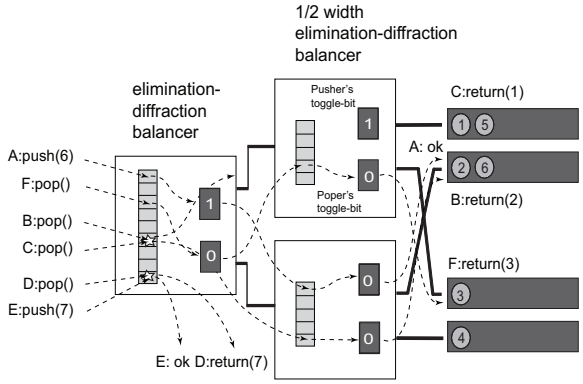

Fig. 3.2: An *ED-Tree*. Each balancer in *Tree[4]* is an elimination-diffraction balancer. The start state depicted is the same as in Figure 3.1, as seen in the producer's toggle bits. From this state, an insert of item 6 by Thread A will not meet any others on the elimination-diffraction arrays, and so will toggle the bits and end up on the 2nd stack from the top. Two removes by Threads B and C will meet in the top balancer's array, diffract to the sides, and end up going up and down without touching the bit, ending up popping the first two values values 1 and 2 from the top two lock-free queues. Thread F which did not manage to diffract or eliminate, will end up as desired on the 3rd queue, returning a value 3. Finally, Threads D and E will meet in the top array and "eliminate" each other, exchanging the value 7 and leaving the tree. This is our exception to the FIFO rule, to allow good performance at high loads, we allow threads with concurrent insert and remove requests to eliminate and leave, ignoring the otherwise FIFO order.

It can be shown that all insert and remove requests that do not eliminate each other provide a quiescently consistent FIFO queue behavior. Moreover, while the worst case time is $\log k$ where k is the number of lock-free queues at the leaves, in contended cases, as our test show, $1/2$ the requests are eliminated in the first balancer, another $1/4$ in the second, $1/8$ on the third, and so on, which converges to an average of 2 steps to complete insert or remove, independent of k.

4. IMPLEMENTATION

4.1 The tree nodes

As described above each balancer (see the pseudo-code in Listing 4.1) is composed of an *EliminationArray*, a pair of *toggle* bits, and two references one to each of its child nodes. The last field, *lastSlotRange* (which has to do with the adaptive behavior of the elimination array), will be described later in Section 4.2.

```
1   public class Balancer{
2       ToggleBit producerToggle, consumerToggle;
3       Exchanger[] eliminationArray;
4       Balancer topChild, bottomChild;
5       ThreadLocal<Integer> lastSlotRange;
6   }
```

Listing 4.1: Balancer

The implementation of a toggle bit as shown in Listing 4.2 is based on an AtomicBoolean which provides a compareAndSet API (CAS). To access it a thread fetches the current value (Line 5) and tries to atomically replace it with the complementary value (Line 6). In case of a failure the thread retries (Line 6).

```
1   AtomicBoolean toggle = new AtomicBoolean(true);
2   public boolean toggle(){
3       boolean result;
4       do{
5           result  = toggle.get();
6       }while(!toggle.compareAndSet(result, !result));
7       return result;
8   }
```

Listing 4.2: Toggle bit

The implementation of an *eliminationArray* is based on an array of *Exchangers*. Each exchanger maintains a place in memory where a thread can "declare" himself and spin waiting for another thread to arrive. When a thread enters the exchanger and sees that another thread is already waiting, the two threads exchange information, based on which each of them later decides weather to eliminate or diffract. Each exchanger (Listing 4.3) contains a single AtomicReference which is used as an Atomic placeholder for exchanging ExchangerPackage, where the ExchangerPackage is an object used to wrap the actual data and to mark its state and type. Each thread approaching ED-tree, wraps it's relevant data in a new instance of ExchangerPackage and the exchangers help threads to approach each others package in order to understand what type of thread was met and in case of elimination for consumer thread to reach the item that the producer thread tries to insert.

```
1   public class Exchanger{
2       AtomicReference<ExchangerPackage> slot;
3   }
4
5   public class ExchangerPackage{
6       Object value;
7       State state; // WAITING/ELIMINATION/DIFFRACTION,
8       Type type; // PRODUCER/CONSUMER
9   }
```

Listing 4.3: Toggle bit

Each thread performing either insert or remove traverses the tree as follows. Starting from the root balancer, the thread tries to exchange its package with a thread with an opposing operation, a producer tries to exchange with a consumer and vice versa. In each balancer, each thread chooses a random slot in the eliminationArray, enters the exchanger, publishes its package, and then backs off in time, waiting in a loop to be eliminated. In case of failure, a back-off in "space" is performed several times. The type of space back off depends on the cause of the failure: If a timeout is reached without meeting any other thread, a new slot is randomly chosen in a smaller range. However, if a timeout is reached after repeatedly failing in the CAS while trying to either pair or just to swap in, a new slot is randomly chosen in a larger range. If collision occurs, the thread examines it's partner's package and decides to diffract if the partner is of the same type and eliminate otherwise. The direction of the diffraction is determined by the order in which CAS operations occurred. Thread that succeeded to "publish" itself first in the exchanger will go up and the other one will go down.

The result of the meeting of two threads in each balancer is one of the following four states: *ELIMINATED*, *TOGGLE*, *DIFFRACTED0*, or *DIFFRACTED1*. In case of *ELIMINATED*, a Consumer and a Producer successfully paired-up, and the method returns. If the result is *TOGGLE*, the thread failed to pair-up with any other type of request, so the *toggle*() method shown in Listing 4.1 is called, and according to its result the thread accesses one of the child balancers. Lastly, if the state is either *DIFFRACTED0* or *DIFFRACTED1*, this is a result of two operations of the same type meeting in the same location, and the corresponding child balancer, either 0 or 1, is chosen.

4.2 Adaptivity

In the back off mechanism described above the thread senses the level of contention and depending on it selects randomly an appropriate range of the eliminationArray to work on (by iteratively backing off). However, each time a thread starts a new operation, it initializes the back-off parameters, wasting the same unsuccessful rounds of back-off in place until sensing the current level of contention. To avoid this, each thread saves its last used range between invocations (Listing 4.1 line 5). This saved range is used as (a good guess of) the initial range at the beginning of the next operation. Using this method proved to be a major factor in reducing the overhead in low contention situations and allowing the EDTree to yield good performance under high contention.

Additional adaptation mechanism we add to the backoff in space described earlier, is the following: If a thread failed to collide after certain amount of timeouts, implying that there are small amount of threads in the balancer and the load is low, it moves to spin on the first cell of the array for a certain amount of spins, before approaching the toggle bit and moving to the next level, that way increasing the chance of successful collision. Additionally a special flag is set to this thread indicating that in the next levels of the tree it should skip the elimination array and approach toggle bit immediately in order to reach the bottom of the tree as fast as possible without wasting unnecessary time, spinning and waiting to other threads. Since this type of scenario indicates that the contention in the tree is probably low, there will be no contention on the toggle bit, and moving to the next level will occur fast. Thus, the described mechanism helps indicating cases when the contention is low and the chance of meeting another thread in the tree is small, already at the first level of the tree, making threads avoid unnecessary overhead while traversing the rest of the tree.

As a final step, a thread that reaches one of the tree leaves, performs it's action(insert or remove) on a queue. A queue can be one of the known queue implementations: a *SynchronousQueue*, a *LinkedBlockingQueue*, or a *ConcurrentLinkedQueue*. Based on ED-Tree with the different queues we implemented the following three types of pools:

4.3 An Unfair Synchronous Pool [7]

When setting the leafs to hold an unfair *SynchronousQueue*, we get a *unfair synchronous pool*. As the synchronous queue, an unfair synchronous pool provides a "pairing up" function without the buffering. Producers and consumers wait for one another, rendezvous, and leave in pairs. Thus, though it has internal queues to handle temporary overflows of mismatched items, the unfair synchronous pool does not require any long-term internal storage capacity.

4.4 An Object Pool

With a simple replacement of the former SynchronousQueue with a *LinkedBlockingQueue*, or a *ConcurrentLinkedQueue* we get *Blocking Object Pool*, or a *NonBlocking Object Pool* in respect. An *object pool* is a software design pattern. It consists of a multi-set of initialized objects that are kept ready to use, rather than allocated and destroyed on demand. A client of the object pool will request an object from the pool and perform operations on the returned object. When the client finishes work on an object, it returns it to the pool rather than destroying it. Thus, it is a specific type of factory object.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when the creation of the new objects (especially over a network) may take variable time. In this thesis paper we show two versions of an object pool: blocking and a non blocking. The only difference between these pools is the behavior of the consumer thread when the pool is empty. While in the blocking version a consumer is forced to wait until an available resource is inserted back to Pool in the unblocking version a consumer thread can leave without an object.

An example of a widely used object pool is a *connection pool*. A connection pool is a cache of database connections maintained by the database so that the connections can be reused when the database receives new requests for data. Connection pools are used to enhance the performance of executing commands on the database. Opening and maintaining a database connection for each user, especially of requests made to a dynamic database-driven website application, is costly and wastes resources. In connection pooling, after a connection is created, it is placed in the pool and is used again so that a new connection does not have to be established. If all the connections are being used, a new connection is made and is added to the pool. Connection pooling also cuts down on the amount of time a user waits to establish a connection to the database.

4.5 Starvation Avoidance

Finally, in order to avoid starvation in our pool, we limit the time a thread can be blocked in the queues at the bottom of the tree before it retries the whole *Tree[k]* traversal again. Starvation (Though it has never been observed in all our tests) can take place when threads that reached the queues on the leafs wait too long for their request to be fulfilled, while most of the new threads entering pool eliminate on the way down the tree and leave fast.

5. PERFORMANCE EVALUATION

We evaluated the performance of our new algorithms on a Sun UltraSPARC T2 Plus multicore machine. This machine has 2 chips, each with 8 cores running at 1.2 GHz, each core with 8 hardware threads, so 64 way parallelism on a processor and 128 way parallelism across the machine. There is obviously a higher latency when going to memory across the machine (a two fold slowdown). In all tests, the ED-tree we used is constructed from three levels, with 8 queues at the leaves. We found this construction optimal for the range of the contention levels that was tested (2 threads to 256 threads approaching in parallel), adding not too much overhead in case of low contention and providing a sufficient level of distribution in case of high contention.

We begin in Figure 5.1 by comparing the new unfair synchronous queue of Lea et. al [7] scheduled to be added to the *java.util.concurrent* library of JDK, to our ED-Tree based version of an unfair synchronous queue. As we explained earlier, an *unfair synchronous queue* provides a symmetric"pairing up" function without buffering: Producers and consumers wait for one another, rendezvous, and leave in pairs.

One can see that the ED-Tree version behaves similarly to the JDK version up to 8 threads (left figure). Above this number of threads the ED-Tree scales nicely while the JDK queue's overall throughput declines. At its peak at 64 threads the ED-Tree delivers more than 10 times the performance of the JDK.

Beyond 64 threads the threads are no longer bound to a single CPU, and traffic across the interconnect causes a moderate performance decline for the ED-Tree version.

We next compare two versions of an object Pool. An *object pool* is a set of initialized objects that are kept ready to use, rather than allocated and destroyed on demand. A consumer of the pool will request an object from the pool and perform operations on the returned object. When the consumer has finished with an object, it returns it to the pool, rather than destroying it. The object pool is a specific type of factory object. Unlike the unfair synchronous queue, the consumers wait in case there is an available object, while producers never wait for consumers, they add the object to the pool and leave.

We compared an ED-Tree *BlockingQueue* implementation to the *LinkedBlock-*

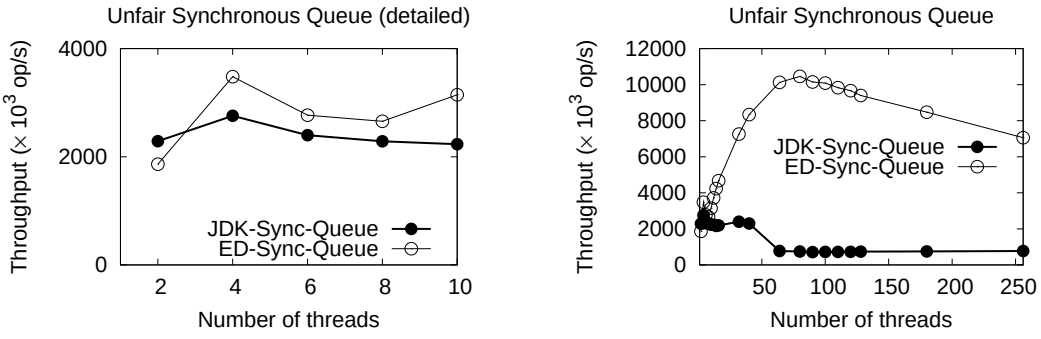

Fig. 5.1: Unfair synchronous queue benchmark: a comparison of the latest JDK 6.0 algorithm and our novel ED-Tree based implementation. The graph on the left is a zoom in of the low concurrency part of the one on the right.

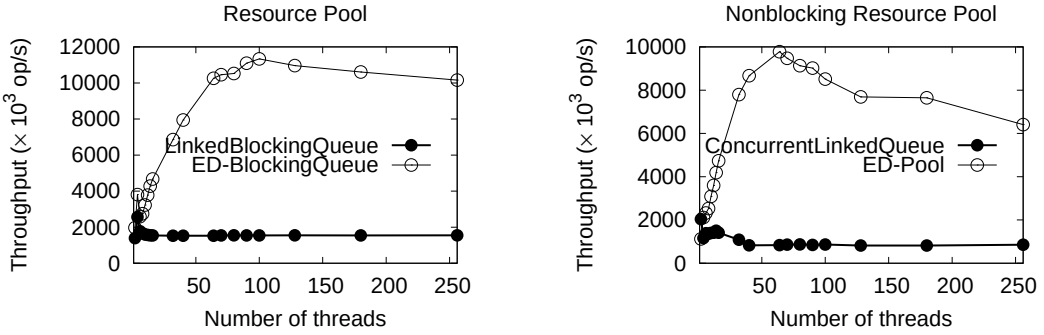

Fig. 5.2: Throughput of *blockingQueue* object pool implementations.

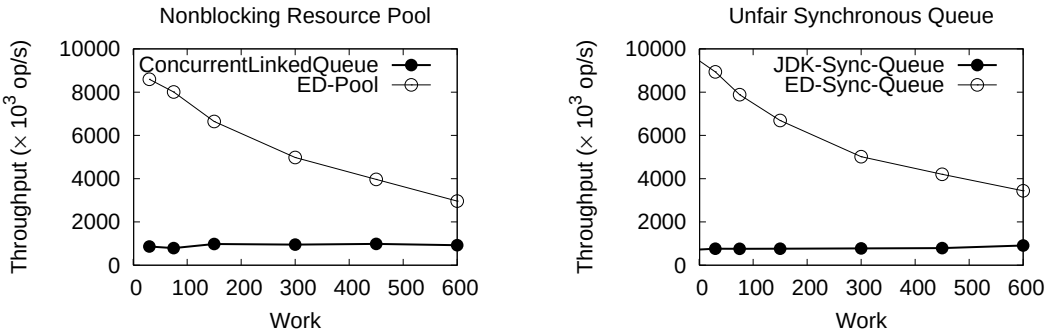

Fig. 5.3: Throughput of a synchronous queue as the work load changes for 64 threads.

ingQueue of JDK6.0. Comparison results for the object pool benchmark are shown on the lefthand side of Figure 5.2.

The results are pretty similar to those in the unfair synchronous queue. The JDK's *LinkedBlockingQueue* performs better than its unfair synchronous queue, yet it still does not scale well beyond 4 threads. In contrast, our ED-Tree version scales well even up to 80 threads because of its underlying use of the *LinkedBlockingQueue*. At its peak at 64 threads it has 10 times the throughput of the JDK's *LinkedBlockingQueue*.

Next, we evaluated implementations of *concurrent queue*, a more relaxed version of an object pool in which there is no requirement for the consumer to wait in case there is no object available in the Pool. We compared the java.util.concurrent.ConcurrentLinkedQueue (which in turn is based on Michael's lock-free linked list algorithm [8]) to an ED-Tree based *ConcurrentQueue*. Again, the results show a similar pattern: the JDK's ConcurrentLinkedQueue scales up to 14 threads, and then drops, while the ED-Tree based ConcurrentQueue scales well up to 64 threads. At its peak at 64 threads, it has 10 times the throughput of the JDK's ConcurrentLinkedQueue.

Since the ED-Tree object pool behaves well at very high loads, we wanted to test how it behaves in scenarios where the working threads are not pounding the pool all the time. To this end we emulate varying work loads by adding a delay between accesses to the pool. We tested 64 threads with a different set of dummy delays due to work, varying it from 30-600ms. The comparison results in Figure 5.3 show that even as the load decreases the ED-Tree synchronous queue outperforms the JDK's synchronous queue. This is due to the low overhead adaptive nature of the randomized mapping into the EDArray: as the load decreases, a thread tends to dynamically shrink the range of array locations into which it tries to map.

Another work scenario that was tested is the one when the majority of the pool users are consumers, i.e. the rate of inserting items to the pool is a lot lower than

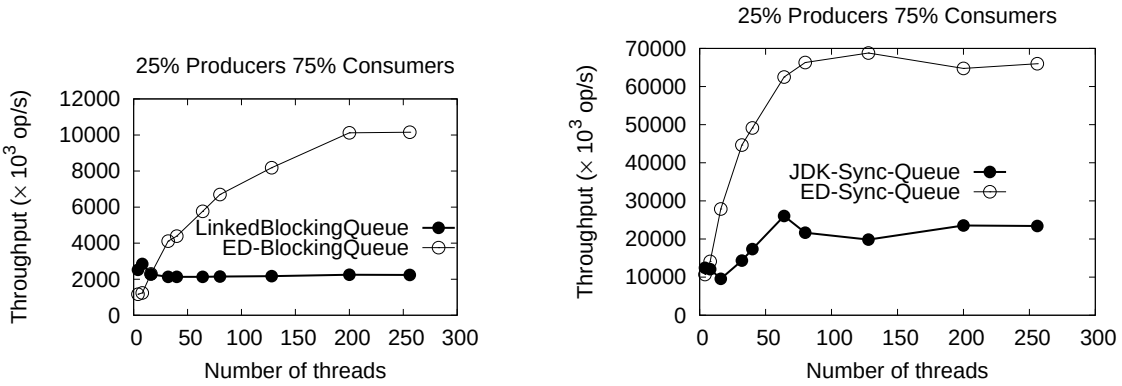

Fig. 5.4: Performance of Linked-Blocking queue and Synchronous queue approached by 25% Producers and 75% consumers.

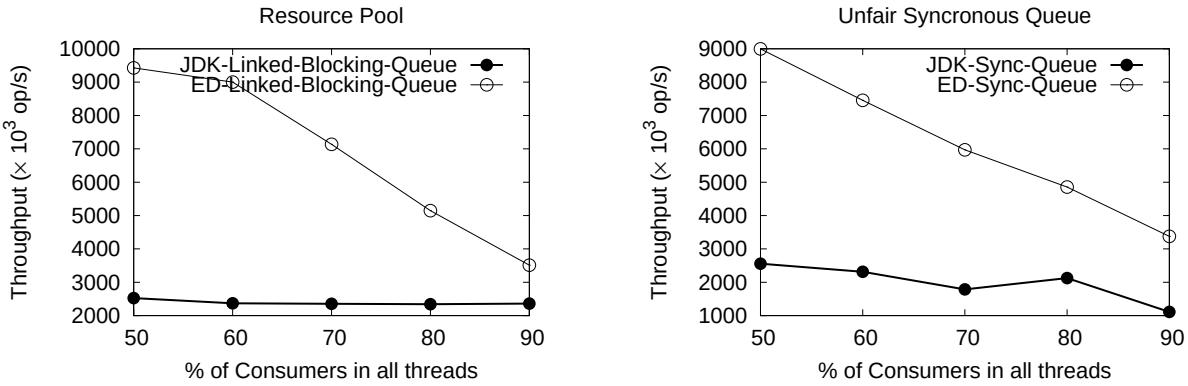

Fig. 5.5: Performance changes of Resource pool and Unfair SynchronousQueue when total number of threads is 64, as the ratio of consumer threads grows from 50% to 90% of total thread amount.

the one demanded by consumers and they have to wait until items become available. Figure 5.4 shows a workload in which 25% of the threads are producers, performing insert operations, and 75% of the threads are consumers, removing items from the pool. One can see, that while ED-pool is outperformed in low contention, it still scales nicely and outperforms both JDK's linked-blocking queue and synchronous queue in more contended workloads.

Figure 5.5 shows what happens when number of threads using the pool is steady, but the ratio of consumers changes from 50% to 90%, e.i from the scenario when number of producers and consumers is equal to one where there is a majority of consumers, meaning more remove that insert requests.

Then, investigated the internal behavior of the ED-Tree with respect to the number of threads, we check the elimination rate at each level of the tree. The results appear in Figure 5.6. Surprisingly, we found out that the higher the concurrency, that is, the more threads added, the more threads get all the way down the tree to the queues. At 4 threads, all the requests were eliminated at the top level, and throughout the concurrency range, even at 265 threads, 50% or more of the requests were eliminated at the at the top level of the tree, at least 25% at the next level, and at least 12.5% at the next. This, as we mentioned earlier, forms a sequence that converges to less than 2 as n, the number of threads, grows. In our particular 3-level ED-Tree tree the average is 1.375 balancer accesses per sequence, which explains the great overall performance.

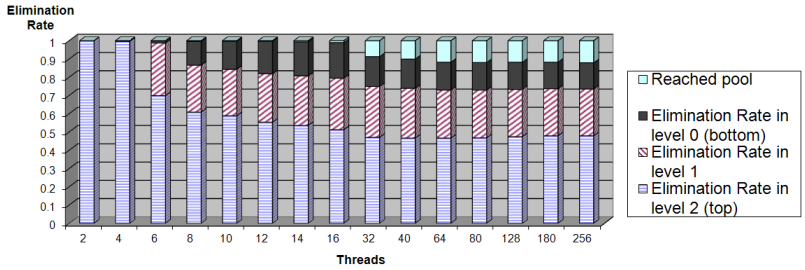

Fig. 5.6: Elimination rate by levels as concurrency increases.

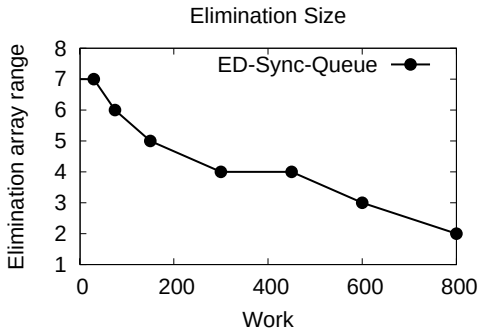

Fig. 5.7: Elimination range as the work load changes for 64 threads.

Lastly, investigated how the adaptive method of choosing the elimination range behaves with different contention loads. Figure 5 shows that as we expected the algorithm dynamically adapts the working range according to the contention load. The more each thread spent doing work not related to the pool, the more the contention decrease and respectively the default range used by the threads decrease.

APPENDIX

A. CODE IMPLEMENTATIONS

This Appendix contains pseudocode describing implementation of several key functionalities of the ED-TREE.

A.1 Balancer

Listing A.1 presents a simplified description of balancer's method *traverseBalancer*, which is called each time a producer or consumer thread enters the balancer's node in the tree. A thread, carrying an exchanger package with it's information, detailed in Chapter 4, approaches the array of exchangers, trying to eliminate or diffract, and in case of failure uses the toggle bit and leaves the balancer. As explained in Chapter 4, the thread local variable lastSlotRange is the range of the elimination array used by the thread last time when it entered the specific balancer.

```
1
2  void traverseBalancer(ExchangerPackage myPackage)
3  {
4      myPackage.CollisionResult = NONE;
5      visitEliminationPrism(myPackage);
6
7      //If there was no collision, use toggle bit to determine direction
8      if (myPackage.CollisionResult == NONE)
9      {
10         if (myPackage.packageType == PRODUCER)
11             myPackage.direction = producerToggle.traverse();
12         else //CONSUMER
13             myPackage.direction = consumerToggle.traverse();
14
15     }
16 }
17
18 void visitEliminationPrism(ExchangerPackage myPackage)
19 {
20     int slotRange = lastSlotRange.get(); //start from lastly remembered range of the array
21     int maxLength = exchangerArray.length;
22
23     int  leftCollisionTrials  = MAX_COLLISION_TRIALS; //Maximum allowed attempts to collide
24     int leftTimeOuts = MAX_TIMEOUTS; //Maximum allowed timeouts
25
26     ExchangeStatus status = SUCCESS;
27     int slot ;
28     while (true)
29     {
30         if (slotRange == 1)
31             slot  = 0;
32         else
33             slot  = Random.Next(slotRange - 1); //Randomly choose a location in the prism array
34
35         status  = exchangerArray[slot].exchange(myPackage); //Try to collide
36
37         if (status == FAILURE)
38         {
39             slotRange = Math.min(maxLength, slotRange * 2);
40              leftCollisionTrials --;
41
42         }
43
44         if (status == TIMEOUT)
45         {
46             slotRange = Math.max(1, slotRange / 2);
47             leftTimeOuts--;
48         }
49
```

```
50          if (status == SUCCESS || leftColisionTrials == 0 || leftTimeOuts == 0)
51              return;
52      }
53      lastSlotRange.set(slotRange);
54  }
```

Listing A.1: Balancer

A.2 Exchanger

Listing A.2 presents a simplified description of the *exchange* method of an Exchanger. Each exchanger references a slot in the prism array, implemented by an atomic reference. Threads that arrive, try to occupy the slot and wait for another thread to collide with. If collision occurs, the thread checks it's "collision partner" type, and decides to eliminate or diffract. To perform diffraction it sets a direction to which it will continue, in case of elimination the threads "exchange" information: a consumer takes producers item.

The occupation of the slot is performed using special flags of type ExchangerStateFlag. Each thread package(of the type ExchangerPackage) contains three unique flags of this type. Each flag contains information about it's carrying thread: the type of the thread: Producer/Consumer, a value(in case the carrying thread is a producer) and the type of the flag WAITING/ELIMINATING/DIFFRACTING. When a thread tries to occupy the slot, it sets there it's unique flag with appropriate state. When collision occurs, the flag is exchanged to signal the second thread about the type of the collision.

```
1
2   AtomicReference<ExchangerStateFlag> slot = new AtomicReference<ExchangerStateFlag>(null);
3   public ExchangeStatus exchange(ExchangerPackage myPackage)
4   {
5       myItem.CollisionResult == NONE;
6       int maxRounds = MAX_EXCHANGER_SPINS;
7       ExchangerStateFlag hisFlag = slot.get();
8       if (hisFlag == null) //The slot is available, no thread is waiting
9       {
10          if (slot.compareAndSet(null, myPackage.getWaitingFlag())) //set waiting flag
11          {
12              //spin waiting for another thread to arrive
13              while (maxRounds-- > 0)
14              {
15                  hisFlag = slot.get();
16                  if (hisFlag != myPackage.getWaitingFlag())
17                  {
18                      exchangeInformation(myPackage, hisFlag);
19                      return SUCCESS;
20                  }
21              }
22
23              if (slot.compareAndSet(myPackage.getWaitingFlag(), null))
24                  return TIMEOUT;
25              else
26              {
27                  exchangeInformation(myPackage, slot.get());
28                  return SUCCESS;
29              }
30          }
31      }
32      else
33      {
34          if(hisFlag.FlagType == WAITING_FLAG) //The slot is taken by some thread, waiting to "collision"
35          {
36              //if the two collided threads are of the same type, diffract and return.
37              if(myPackage.IsProducer() == hisFlag.IsProducer())
38              {
39                  if (slot.compareAndSet(hisFlag, myPackage.getDiffractingFlag()))
40                  {
41                      myPackage.CollisionResult = DIFFRACTED;
42                      myPackage.Direction = 1;
43                      return SUCCESS;
44                  }
45              }
```

```
46              else //if the collided threads are of different type, eliminate.
47              {
48                  if (slot.compareAndSet(hisFlag, myItem.getEliminatingFlag()))
49                  {
50                      myPackage.CollisionResult = ELIMINATED;
51
52                      //If the current thread is a producer, take the other thread's item
53                      if (!myPackage.IsProducer())
54                          myPackage.Value = hisFlag.Value);
55                      return SUCCESS;
56                  }
57              }
58          }
59      }
60      return FAILURE;
61  }
62  private void exchangeInformation(ExchangerPackage myPackage, ExchangerStateFlag hisFlag)
63  {
64          if (hisFlag.FlagType == DIFFRACTION_FLAG)
65          {
66                  myPackage.Direction = 0;
67                  myPackage.CollisionResult = DIFFRACTED;
68          }
69          else //Elimination
70          {
71                  if (!myPackage.IsProducer())
72                          myPackage.Value = hisFlag.Value);
73
74                  myPackage.CollisionResult = ELIMINATED;
75          }
76          slot.set(null);
77  }
```

Listing A.2: Exchanger

BIBLIOGRAPHY

[1] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *The Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, Ischia, Naples, Italy, 8 2010.

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

[3] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 24–32, New York, NY, USA, 1997. ACM.

[4] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM.

[5] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, NY, USA, 2008.

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

[7] William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.

[8] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[9] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262, New York, NY, USA, 2005. ACM.

[10] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 54–63, New York, NY, USA, 1995. ACM.

[11] Nir Shavit, Eli Upfal, and Asaph Zemach. A steady state analysis of diffracting trees (extended abstract). In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 33–41, New York, NY, USA, 1996. ACM.

[12] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.