Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees

Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit

Computer Science Department
Tel-Aviv University, Israel,
contact email: guy.korland@cs.tau.ac.il

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 paper presents the *ED-Tree*, 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 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.

1 Introduction

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.

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. A pool allows push and pop with the usual pool semantics [4]. We call the pushing threads *producers* and the popping threads *consumers*.

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 SynchronousQueue provides a "pairing up" function without buffering; it is entirely symmetric: Producers and consumers wait for one another, rendezvous, and leave in pairs. The term unfair refers to the fact that it allows starvation. The other queues provide a buffering mechanism and allow threads to sleep while waiting for their requests to be fulfilled. Unfortunately, all these pools, including the new scalable SynchronousQueue of Lea, Scott, and Shearer [6], 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.

This paper shows how to overcome this limitation by devising highly distributed pools based on an *ED-Tree*, a combined variant of the diffracting-tree structure of Shavit and Zemach [8] and the elimination-tree structure of Shavit and Touitou [7].

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 Section 2, 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 [8, 1], that allows requests to be properly distributed if such successful exchanges did not occur. As shown in Figure 1, one can add queues at the leaves of the trees 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 in Figure 1, we replace them with highly distributed nodes that use elimination and diffraction on randomly chosen array locations as in Figure 2.

The elimination and diffraction tree structures were each proposed years ago [7, 8] and claimed to be effective through simulation [3]. A single level of an elimination array was also used in implementing shared concurrent stacks [2]. 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 paper is the first to show that 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 10 times that of the existing JDK6.0 algorithms.

But what about low concurrency levels? In their elegant paper describing the JDK6.0 SynchronousQueue, Lea, Scott, and Shearer [6], 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 paper shows that elimination and diffraction techniques can be combined to work well at both high and low loads. There are two main components 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, 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. The end result is an ED-Tree structure, that, as our empirical testing shows, performs well at both high and low concurrency levels.

2 The ED-Tree

Before explaining how the ED-Tree works, let us review its predecessor, the *diffracting tree* [8] (see Figure 1). Consider a binary tree of objects called *balancers* with a single input wire and two output wires, as depicted in Figure 1. Threads arrive at a balancer and it sends them alternately 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 [8].

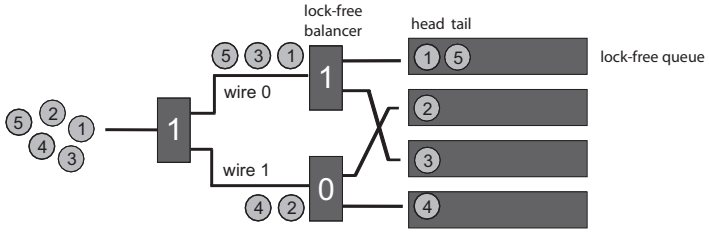

Fig. 1. A *Tree*[4] [8] leading to 4 lock-free queues. Threads pushing items arrive at the balancers in the order of their numbers, eventually pushing items onto the queues located on their output wires. In each balancer, a pushing thread fetches and then complements the bit, following the wire indicated by the fetched value (If the state is 0 the pushing thread it will change it to 1 and continue to top wire (wire 0), and if it was 1 will change it to 0 and continue on bottom wire (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 inserted item, and the next location a pushed item will end up on is the queue containing item 2. Try it! We can add a similar tree structure for popping 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.

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. To perform a push, threads traverse the balancers from the root to the leaves and then push 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.

One could implement the balancers in a straightforward way using a bit that threads toggle: they fetch the bit and then complement it using a *compareAndSet* (CAS) operation, exiting on the output wire they fetched (zero or one). One could keep a second, identical tree for pops, and you would see that from one quiescent state to the next, the items removed are the first ones pushed onto the queue. Thus, we have created a collection of queues that are accessed in parallel, yet act as one quiescent FIFO queue.

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 similar to the one made about the elimination backoff stack:

> 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.

The idea behind the ED-Tree is combining the modified diffracting [8] tree as above with the elimination-tree techniques [7]. We use an *eliminationArray* in front of the bit in every balancer as in Figure 2. If two popping threads meet in the array, they leave on opposite wires, without a need to touch the bit, as anyhow it would remain in

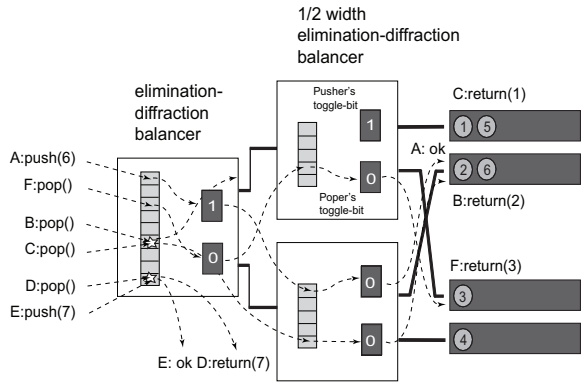

Fig. 2. An *ED-Tree*. Each balancer in *Tree*[4] is an elimination-diffraction balancer. The start state depicted is the same as in Figure 1, as seen in the pusher's toggle bits. From this state, a push of item 6 by Thread A will not meet any others on the elimination-diffraction arrays, and so will toggle the bits and end up on the 2nd stack from the top. Two pops 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 push and pop requests to eliminate and leave, ignoring the otherwise FIFO order.

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 respective push or pop bit (in this sense it differs from prior elimination and/or diffraction balancer algorithms [7, 8] which had a single toggle bit instead of separate ones, and provided LIFO rather than FIFO like access through the bits) and leaves accordingly. Finally, if a push and a pop meet, they eliminate, exchanging items. It can be shown that all push and pop 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, $1/2$ 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 a push or a pop, independent of k.

3 Implementation

As described above, each balancer (see the pseudo-code in Listing 1.1) is composed of an *eliminationArray*, a pair of *toggle* bits, and two pointers, 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 this section.

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

Listing 1.1. A Balancer

The implementation of a toggle bit as shown in Listing 1.2 is based on an AtomicBoolean which provides a CAS operation. 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 1.2. The Toggle of a Balancer

The implementation of an eliminationArray is based on an array of *Exchangers*. Each exchanger (Listing 1.3) contains a single AtomicReference which is used as a placeholder for exchanging, and an ExchangerPackage, where the ExchangerPackage is an object used to wrap the actual data and to mark its state and type.

```
1    public class Exchanger{
2        AtomicReference<ExchangerPackage> slot;
3    }
4
5    public class ExchangerPackage{
6        Object value;
7        State state;
8        Type type;
9    }
```

Listing 1.3. An Exchanger

Each thread performing either a push or a pop, traverses the tree as follows. Starting from the root balancer, the thread tries to exchange its package with a thread with a complementary operation, a popper tries to exchange with a pusher and vice versa. In each balancer, each thread chooses a random slot in the eliminationArray, publishes its package, and then backs off in time, waiting in a loop to be eliminated. In case of failure, a backoff 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.

Adaptivity In the backoff mechanism described above, a 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 backoff parameters, wasting the same unsuccessful rounds of backoff in place until sensing the current level of contention. To avoid this, we let each thread save its last-used range between invocations (Listing 1.1 line 5). This saved range is used as (a good guess of) the initial range at the beginning of the next operation. 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.

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

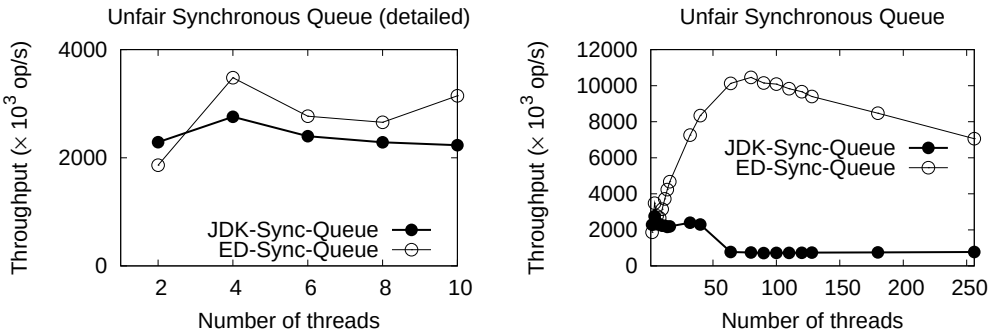

Fig. 3. The unfair synchronous queue benchmark: a comparison of the latest JDK 6.0 algorithm and our novel ED-Tree based implementation. The graph on the left is a zoom in of the low concurrency part of the one on the right. Number of producers and consumers is equal in each of the tested workloads.

ELIMINATED, a popper and a pusher 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 1.1 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.

As a final step, the item of a thread that reaches one of the tree leaves is placed in the corresponding queue. A queue can be one of the known queue implementations: a SynchronousQueue, a LinkedBlockingQueue, or a ConcurrentLinkedQueue. Using ED-Trees with different queue implementations we created the following three types of pools:

An Unfair Synchronous Queue When setting the leaves to hold an unfair SynchronousQueue, we get a *unfair synchronous queue* [6]. An unfair synchronous queue 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 queue does not require any long-term internal storage capacity.

An Object Pool With a simple replacement of the former SynchronousQueue with a LinkedBlockingQueue. With a ConcurrentLinkedQueue we get a *blocking object pool*, or a *non-blocking object pool* respectively. 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 paper we show two versions of an object pool: blocking and

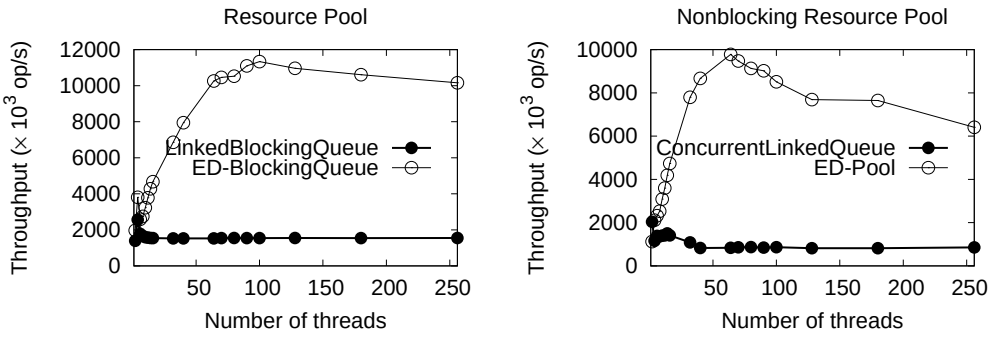

Fig. 4. Throughput of BlockingQueue and ConcurrentQueue object pool implementations. Number of producers and consumers is equal in each of the tested workloads.

a non blocking. The only difference between these pools is the behavior of the popping thread when the pool is empty. While in the blocking version a popping thread is forced to wait until an available resource is pushed back to Pool, in the unblocking version it 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. Such 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.

Starvation avoidance Finally, in order to avoid starvation in the queues (Though it has never been observed in all our tests), we limit the time a thread can be blocked in these queues before it retries the whole $Tree[k]$ traversal again.

4 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).

We begin our evaluation in Figure 3 by comparing the new unfair SynchronousQueue of Lea et. al [6], scheduled to be added to the *java.util.concurrent* library of JDK6.0, 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 behaves similarly to the JDK version up to 8 threads(left figure). Above this concurrency level, the ED-Tree scales nicely while the

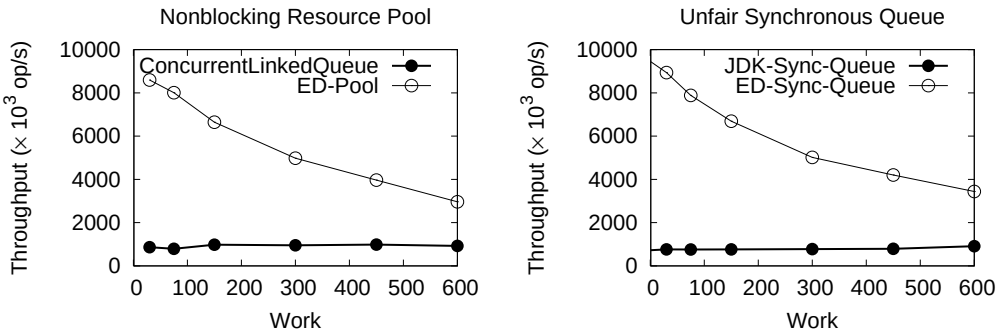

Fig. 5. Throughput of a SynchronousQueue as the work load changes for 32 producer and 32 consumer threads.)

JDK implementation's overall throughput declines. At its peak, at 64 threads, the ED-Tree delivers more than 10 times the performance of the JDK implementation.

Beyond 64 threads, the threads are no longer placed on a single chip, 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 using an object, it returns it to the pool, rather than destroying it. The object pool is thus a type of factory object. The consumers wait in case there is no available object, while the producers, unlike producers of unfair synchronous queue, never wait for consumers, they add the object to the pool and leave.

We compared an ED-Tree BlockingQueue implementation to the LinkedBlockingQueue of JDK6.0. Comparison results for the object pool benchmark are shown on the lefthand side of Figure 4.

The results are pretty similar to those in the unfair SynchronousQueue. The JDK's LinkedBlockingQueue performs better than its unfair SynchronousQueue, 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 ConcurrentQueue, 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 ConcurrentLinkedQueue of JDK6.0 (which in turn is based on Michael's lock-free linked list algorithm [5]) to an ED-Tree based ConcurrentQueue (righthand side of Figure 4). 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 show that even as the load decreases the ED-Tree synchronous queue outperforms the JDK's synchronous

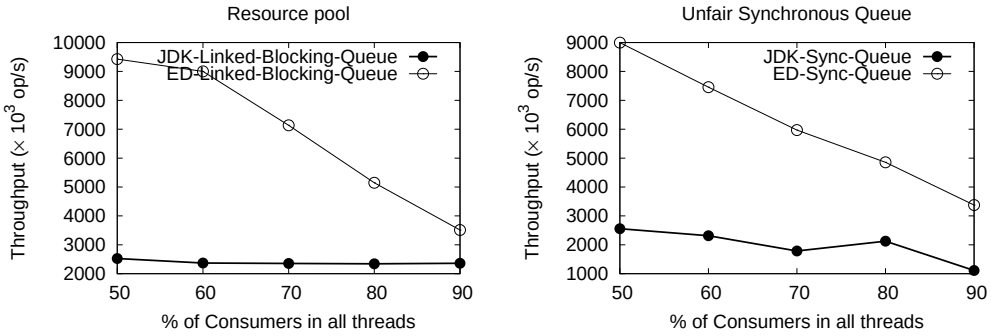

Fig. 6. Performance changes of a Resource pool and unfair SynchronousQueue when total number of threads is 64, as the ratio of consumer threads grows from 50% to 90% of total thread amount.

queue. , This is due to the low overhead adaptive nature of the randomized mapping into the eliminationArray: 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 lower than the one demanded by consumers and they have to wait until items become available. Figure 6 shows what happens when number of threads using the pool is steady(64 threads), but the ratio of consumers changes from 50% to 90%. One can see that ED-tree outperforms JDK's structures both in case when the number of producer and consumer thread equals and in cases where there are a lot more consumer threads than producer threads (for example 90% consumers and 10% producers) .

Next, we 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 7. 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 top level of the tree, at least 25% at the next level, and at least 12.5% at the

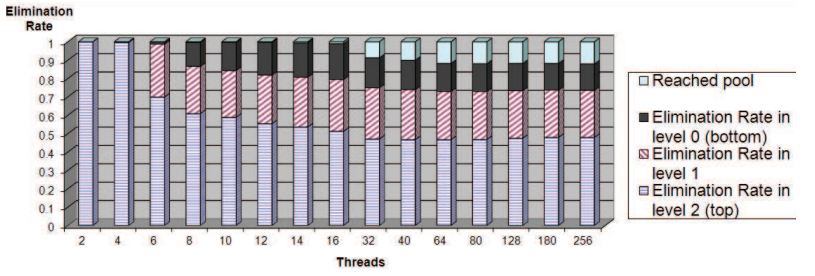

Fig. 7. Elimination rate by levels, as concurrency increases.

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.

Lastly, we investigated how the adaptive method of choosing the elimination range behaves under different loads. Figure 8 shows that, as we expected, the algorithm adapts the working range to the load reasonably well. The more each thread spent doing work not related to the pool, the more the contention decreased, and respectively, the default range used by the threads decreased.

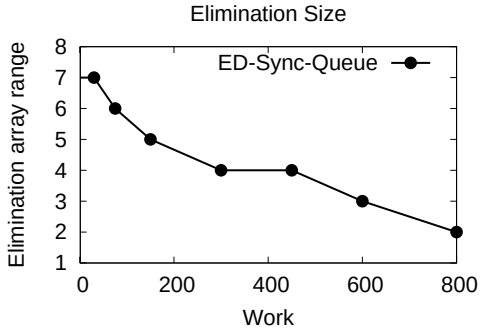

Fig. 8. Elimination range as the work load changes for 32 producer and 32 consumer threads.

Acknowledgements. This 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).

References

1. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.
2. D. Hendler, N. Shavit, and L. 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.
3. M. Herlihy, B. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.
4. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, NY, USA, 2008.
5. M. M. Michael and M. 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.
6. W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.
7. N. Shavit and D. 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.
8. N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.