Scalable Flat-Combining Based Synchronous Queues

Danny Hendler[1], Itai Incze[2], Nir Shavit[2,3] and Moran Tzafrir[2]

[1] Ben-Gurion University
[2] Tel-Aviv University
[3] Sun Labs at Oracle

Abstract. In a *synchronous queue*, producers and consumers handshake to exchange data. Recently, new scalable unfair synchronous queues were added to the Java JDK 6.0 to support high performance thread pools.
This paper applies flat-combining to the problem of designing a synchronous queue algorithm. We first use the original flat-combining algorithm, a single "combiner" thread acquires a global lock and services the other threads' combined requests with very low synchronization overheads. As we show, this single combiner approach delivers superior performance up to a certain level of concurrency, but unfortunately does not continue to scale beyond that point.
In order to continue to deliver scalable performance as concurrency increases, we introduce a new *parallel flat-combining* algorithm. The new algorithm dynamically adds additional concurrently executing flat-combiners that coordinate their work. It enjoys the low coordination overheads of sequential flat combining, with the added scalability that comes with parallelism.
Our novel unfair synchronous queue using parallel flat combining exhibits scalability far and beyond that of the JDK 6.0 algorithm: it matches it in the case of a single producer and consumer, and is superior throughout the concurrency range, delivering up to 11 (eleven) times the throughput at high concurrency.

1 Introduction

This paper presents a new highly scalable design of an *unfair synchronous queue*, a fundamental concurrent data structure used in a variety of concurrent programming settings.

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. As noted in [7], many applications require "poll" and "offer" operations which take an item only if a producer is already present, or put an item only if a consumer is already waiting (otherwise, these operations return an error). The *synchronous queue* provides such a "pairing up" of items, 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 the queue is actually a pool [5]: it does not impose an order on the servicing of requests, and permits starvation. Previous synchronous queue algorithms were presented by Hanson [3], by Scherer, Lea and Scott [8, 7, 9] and by Afek et al. [1]. A survey of past work on synchronous queues can be found in [7].

New scalable implementations of synchronous queues were recently introduced by Scherer, Lea, and Scott [7] into the Java 6.0 JDK, available on more than 10 million desktops. They showed that the unfair version of a synchronous queue delivers scalable performance, both in general and when used to implement the JDK's thread pools.

In a recent paper [4], we introduced a new synchronization paradigm called *flat combining* (FC). At the core of flat combining is a low cost way to allow a single "combiner" thread at a time to acquire a global lock on the data structure, learn about all concurrent access requests by other threads, and then perform their combined requests on the data structure. This technique has the dual benefit of reducing the synchronization overhead on "hot" shared locations, and at the same time reducing the overall cache invalidation traffic on the structure. The effect of these reductions is so dramatic, that in a kind of "anti-Amdahl's law" effect, they outweigh the loss of parallelism caused by allowing only one combiner thread at a time to manipulate the structure.

This paper applies the flat-combining technique to the synchronous queue implementation problem. We begin by presenting a scalable flat-combining implementation of an *unfair synchronous queue* using the technique suggested in [4]. As we show, this implementation outperforms the new Java 6.0 JDK implementation at all concurrency levels (by a factor of up to 3 on a Sun Niagara 64 way multicore). However, it does not continue to scale beyond some point, because in the end it is based on a single sequentially executing combiner thread that executes the operations of all others.

Our next implementation, and the core result in this paper, is a synchronous queue based on *parallel flat-combining*, a new flat combining algorithm in which multiple instances of flat combining are executed in parallel in a coordinated fashion. The parallel flat-combining algorithm spawns new instances of flat-combining dynamically as concurrency increases, and folds them as it decreases. The key problem one faces in such a scheme is how to deal with imbalances: in a synchronous queue one must "pair up" requests, without buffering the imbalances that might occur. Our solution is a dynamic two level exchange mechanism that manages to take care of imbalances with very little overhead, a crucial property for making the algorithm work at both high and low load, and at both even and uneven distributions. We note that a synchronous queue, in particular a parallel one, requires a higher level of coordination from a combiner than that of queues, stacks, or priority queues implemented in our previous paper [4], which introduced flat combining. This is because the lack of buffering means there is no "slack": the combiner must actually match threads up before releasing them.

As we show, our parallel flat-combining implementation of an unfair synchronous queue outperforms the single combiner, continuing to improve with

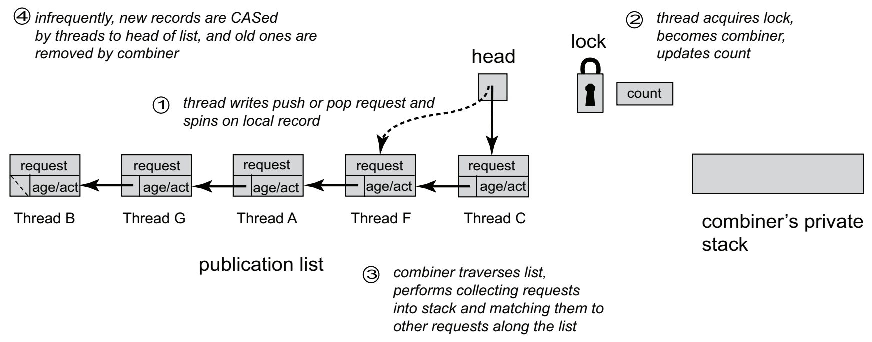
④ infrequently, new records are CASed by threads to head of list, and old ones are removed by combiner
lock
② thread acquires lock, becomes combiner, updates count
head
count
① thread writes push or pop request and spins on local record
request
age/act
Thread B
Thread G
Thread A
Thread F
Thread C
combiner's private stack
publication list
③ combiner traverses list, performs collecting requests into stack and matching them to other requests along the list

Fig. 1. A synchronized-queue using a *single combiner* flat-combining structure. Each record in the publication list is local to a given thread. The thread writes and spins on the request field in its record. Records not recently used are once in a while removed by a combiner, forcing such threads to re-insert a record into the publication list when they next access the structure.

the level of concurrency. On a Sun Niagara 64 way multicore, it reaches up to 11 times the throughput of the JDK implementation at 64 threads.

The rest of this paper is organized as follows. We outline the basic sequential flat-combining algorithm and describe how it can be used to implement a synchronous queue in Section 2. In Section 3, we describe our parallel FC synchronous queue implementation. Section 4 reports on our experimental evaluation. We conclude the paper in Section 5 with a short discussion of our results.

2 A Synchronous Queue Using Single-Combiner Flat-Combining

In a previous paper [4], we showed how given a sequential data structure D, one can design a (single-combiner) *flat combining* (henceforth FC) concurrent implementation of the structure. For presentation completeness, we outline this basic sequential flat combining algorithm in this section and describe how we use it to implement a synchronous queue. Then, in the next section, we present our parallel FC algorithm that is based on running multiple dynamically maintained instances of this basic algorithm in a two level hierarchy.

As depicted in Figure 1, to implement a single instance of FC, a few structures are added to a sequential structure D: a *global lock*, a *count* of the number of combining passes, and a pointer to the *head* of a *publication list*. The publication list is a list of thread-local records of a size proportional to the number of threads that are concurrently accessing the shared object. Though one could implement the list in an array, the dynamic publication list using thread local pointers is necessary for a practical solution: because the number of potential threads is unknown and typically much greater than the array size, using an array one would have had to solve a renaming problem [2] among the threads accessing it. This would imply a CAS per location, which would give us little advantage over existing techniques.

Each thread t accessing the structure to perform an invocation of some method m on the shared object executes the following sequence of steps. We describe only the ones important for coordination so as to keep the presentation as simple as possible. The following then is the *single combiner algorithm* for a given thread executing a method m:

1. Write the invocation opcode and parameters (if any) of the method m to be applied sequentially to the shared object in the *request* field of your thread local publication record (no need to use a load-store memory barrier). The *request* field will later be used to receive the response. If your thread local publication record is marked as active continue to step 2, otherwise continue to step 5.
2. Check if the global lock is taken. If so (another thread is an active combiner), spin on the *request* field waiting for a response to the invocation (one can add a yield at this point to allow other threads on the same core to run). Once in a while, while spinning, check if the lock is still taken and that your record is active. If your record is inactive proceed to step 5. Once the response is available, reset the request field to null and return the response.
3. If the lock is not taken, attempt to acquire it and become a combiner. If you fail, return to spinning in step 2.
4. Otherwise, you hold the lock and are a combiner.
   - Increment the combining pass *count* by one.
   - Traverse the publication list (our algorithm guarantees that this is done in a wait-free manner) from the publication list head, combining all non-null method call invocations, setting the *age* of each of these records to the current *count*, applying the combined method calls to the structure D, and returning responses to all the invocations.
   - If the *count* is such that a cleanup needs to be performed, traverse the publication list from the *head*. Starting from the second item (as we explain below, we always leave the item pointed to by the *head* in the list), remove from the publication list all records whose *age* is much smaller than the current *count*. This is done by removing the record and marking it as inactive.
   - Release the *lock*.
5. If you have no thread local publication record allocate one, marked as active. If you already have one marked as inactive, mark it as active. Execute a store-load memory barrier. Proceed to insert the record into the list by repeatedly attempting to perform a successful CAS to the *head*. If and when you succeed, proceed to step 1.

Records are added using a CAS only to the head of the list, and so a simple wait free traversal by the combiner is trivial to implement [5]. Thus, removal will not require any synchronization as long as it is not performed on the record pointed to from the head: the continuation of the list past this first record is only ever changed by the thread holding the global lock. Note that the first item is not an anchor or dummy record, we are simply not removing it. Once a new record is inserted, if it is unused it will be removed, and even if no new records are added, leaving it in the list will not affect performance.

The common case for a thread is that its record is active and some other thread is the combiner, so it completes in step 2 after having only performed a store and a sequence of loads ending with a single cache miss. This is supported by the empirical data presented later.

Our implementation of the FC mechanism allows us to provide the same clean concurrent object-oriented interface as used in the Java concurrency package [6] and similar C++ libraries [13], and the same consistency guarantees. We note that the Java concurrency package supports a time-out capability that allows operations awaiting a response to give up after a certain elapsed time. It is straightforward to modify the push and pop operations we support in our implementation into dual operations and to add a time-out capability. However, for the sake of brevity, we do not describe the implementation of these features in this extended abstract.

To access the synchronous queue, a thread t posts the respective pair <PUSH,v> or <POP,0> to its publication record and follows the FC algorithm. As seen in Figure 1, to implement the synchronous queue, the combiner keeps a *private stack* in which it records push and pop requests (and for each also the publication record of the thread that requested them). As the combiner traverses the publication list, it compares each requested operation to the top operation in the private stack. If the operations are complementary, the combiner provides the requestor and the thread with the complementary operation with their appropriate responses, and releases them both. It then pops the operation from the top of the stack, and continues to the next record in the publication list. The stack can thus alternately hold a sequence of pushes or a sequence of pops, but never a mix of both.

In short, during a single pass over the publication list, the combiner matches up as best as possible all the push and pop pairs it encountered. The operations remaining in the stack upon completion of the traversal are in a sense the "overflow" requests of a certain type, that were not serviced during the current combining round and will remain for the next.

In Section 4 a single instance of the single combiner FC algorithm is shown to provide a synchronous queue with superior performance to the one in JDK6.0, but it does not continue to scale beyond a certain number of threads. To overcome this limitation, we now describe how to implement a highly scalable generalization of the FC paradigm using multiple concurrent instances of the single combiner algorithm.

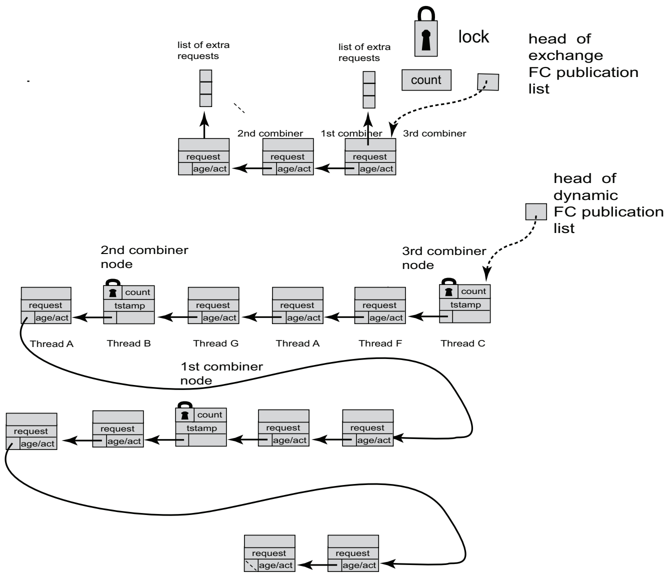

Fig. 2. A synchronized-queue based on *parallel flat combining*. There are two main interconnected elements: the *dynamic* FC structure and the *exchange* FC structure. As can be seen, in this example there are three combiner sublists with approximately 4 records per sublist. Each of the combiners also has a record in the exchanger FC structure.

3 Parallel Flat Combining

In this section we provide a description of the parallel flat combining algorithm. We extend the single combiner algorithm to multiple parallel combiners in the following way. We use two types of flat combining coordination structures, depicted in Figure 2. The first is a *dynamic* FC structure that has the ability to split the publication list into shorter sublists when its length passes a certain threshold, and collapse publication sublists if their lengths go below a certain threshold. The second is an *exchange* single combiner FC structure that implements a synchronous queue in which each request can involve a collection of several push or pop requests.

Each of the multiple combiners that operate on sublists of the dynamic FC structure may fail to match all its requests and be left with an "overflow" of operations of the same type. The exchange FC structure is used for trying to match these overflows. The key technical challenge of the new algorithm is to allow coordination between the two levels of structures: multiple parallel dynamically-created single combiner sublists, and the exchange structure that deals with their overflows. Any significant overhead in this mechanism will result in a performance deterioration that will make the scheme as a whole work poorly.

Each record in the dynamic flat combining publication list is augmented with a pointer (not shown in Figure 2) to a special *combiner node* that contains the lock and other accounting data associated with the sublist currently associated with the record; the request itself is now also a separate *request structure* pointed to by the publication record (this structural change is not depicted in Figure 2). Initially there is a single combiner node and the initial publication records are added to the list starting with this node and point to it.

Each thread t performing an invocation of a push or a pop starts by accessing the head of the *dynamic FC publication list* and executes the following sequence of steps:

1. Write the invocation opcode of the operation and its parameters (if any) to a newly created request structure. If your thread local publication record is marked as active, continue to step 3., otherwise mark it as active and continue to step 2.
2. Publication record is not in list: count the number of records in the sublist pointed to by the currently first combining node in the dynamic FC structure. If less than the threshold (in our case 8, chosen statically based on empirical testing), set the combiner pointer of your publication record to this combining node, and try to CAS yourself into this sublist. Otherwise:
   - Create a new combiner node, pointing to the currently first combiner node.
   - Try to CAS the head pointer to point to the new combiner node.

   Repeat the above steps until your record is in the list and then proceed to step 3.
3. Check if the lock associated with the combiner node pointed at by your publication record is taken. If so, proceed similarly to step 2. of the single FC algorithm: spin on your request structure waiting for a response and, once in while, check if the lock is still taken and that your publication record is active. If your response is available, reset the request field to null and return the response. If your record is inactive, mark it as active and proceed to step 2; if the lock is not taken, try to capture it and, if you succeed, proceed to step 4.
4. You are now a combiner in your sublist: run the combiner algorithm using a local stack, matching pairs of requests in your sublist. If, after a few rounds of combining, there are leftover requests in the stack that cannot be matched, access the exchanger FC structure, creating a record pointing to a list of excess request structures and add it to the exchanger's publication list using the single FC algorithm. The excess request structures are no longer pointed at from the corresponding records of the dynamic FC list.
5. If you become a combiner in the exchanger FC structure, traverse the exchanger publication list using the single combiner algorithm. However, in this single combiner algorithm, each request record points to a list of overflow requests placed by a combiner of some dynamic list, and so you must either match (in case of having previously pushed counter-requests) or push (in other cases) all items in each list before signaling that the request is

complete. This task is simplified by the fact that the requests will always be all pushes or all pops (since otherwise they would have been matched in the dynamic list and never posted to the exchange).

In our implementation we chose to split lists so that they contain approximately 8 threads each (in Figure 2 the threshold is 4). Making the lengths of the sublists and thresholds vary dynamically in a reactive manner is a subject for further research. For lack of space, detailed pseudo-codes of our algorithms are not presented in this extended abstract and will appear in the full paper.

3.1 Correctness

Though there is no obvious way to specify the "rendezvous" property of synchronous queues using a sequential specification, it is straightforward to see that our implementation is linearizable to the next closest thing, an object whose histories consist of a sequence of pairs consisting of a push followed by a pop of the matching value (i.e. push, pop, push, pop...). This follows immediately because each thread only leaves the structure after the flat combiner has matched it to a complementary concurrent operation, and we can linearize the operations at the point of the release of the first of them by the flat combiner.

In terms of robustness, our flat combining implementation is as robust as any global lock based data structure: in both cases a thread holding the lock could be preempted, causing all others to wait. [4]

4 Performance Evaluation

For our experiments we used two machines. The first is an Oracle 64-way Niagara II multicore machine with 8 SPARC cores that multiplex 8 hardware threads each, and share an on chip L2 cache. The second is an Intel Nehalem 8-way machine, with 4 cores that each multiplex 2 hardware threads. We ran benchmarks in Java using the Java 6.0 JDK. In the figures we will refer to these two architectures respectively as SPARC and INTEL.

Our empirical evaluation is based on comparing the relative performance of our new flat combining implementations to the most efficient known synchronous queue implementations: the current Java 6.0 JDK `java.util.concurrent` implementation of the unfair synchronous queue, and the recently introduced *Elimination-Diffraction* trees of Afek et al. [1].

The JDK algorithm, due to Scherer, Lea, and Scott [7], was recently added to Java 6.0 and was reported to provide a three fold improvement over the Java 5.0 unfair synchronous queue implementation. The JDK implementation uses a lock-free linked list based stack in the style of Treiber [12] in order to queue either producer requests or consumer requests but never both at the same time.

[4] On modern operating systems such as Solaris^{TM}, one can use mechanisms such as the *schetdl* command to control the quantum allocated to a combining thread, significantly reducing the chances of it being preempted.

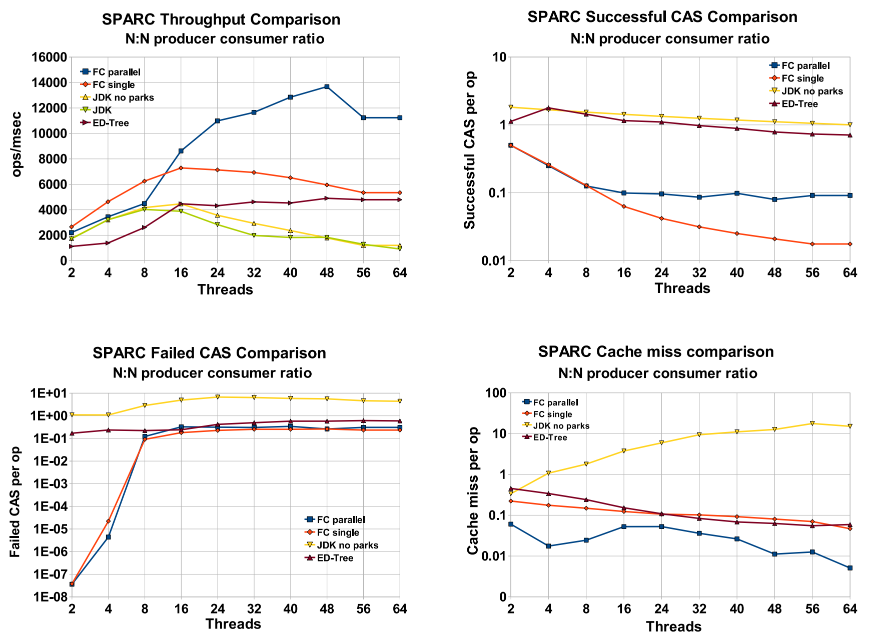

Fig. 3. A Synchronous Queue Benchmark with N consumers and N producers. The graphs show throughput, average CAS failures, average CAS successes, and L2 cache misses (all but the throughput are logarithmic and per operation). We show the throughput graphs for the JDK6.0 algorithm with parks, to make it clear that removing the parks helps performance in this benchmark.

Whenever a request appears, the queue is examined - if it is empty or has nodes which have the same type of requested operation, the request is enqueued using a CAS to the top of the list. Otherwise, the requested operation at the top of the stack is popped using a CAS operation on the head end of the list.

This JDK version must provide the option to *park* a thread (i.e. context switch it out) while it waits for its chance to rendezvous in the queue. A park operation is costly and involves a system call. This, as our graphs show, hurts the JDK algorithm's performance. To make it clear that the performance improvement we obtain relative to the JDK synchronous queue is not a result of the park calls, we implemented a version of the JDK algorithm with the parks neutralized, and use it in our comparison.

The *Elimination-Diffracting tree* [1] (henceforth called ED-Tree), recently introduced by Afek et al., is a distributed data structure that can be viewed as a combination of an elimination-tree [10] and a diffracting-tree [11]. ED-Trees are randomized data-structures that distribute concurrent thread requests onto the nodes of a binary tree consisting of *balancers* (see [11]).

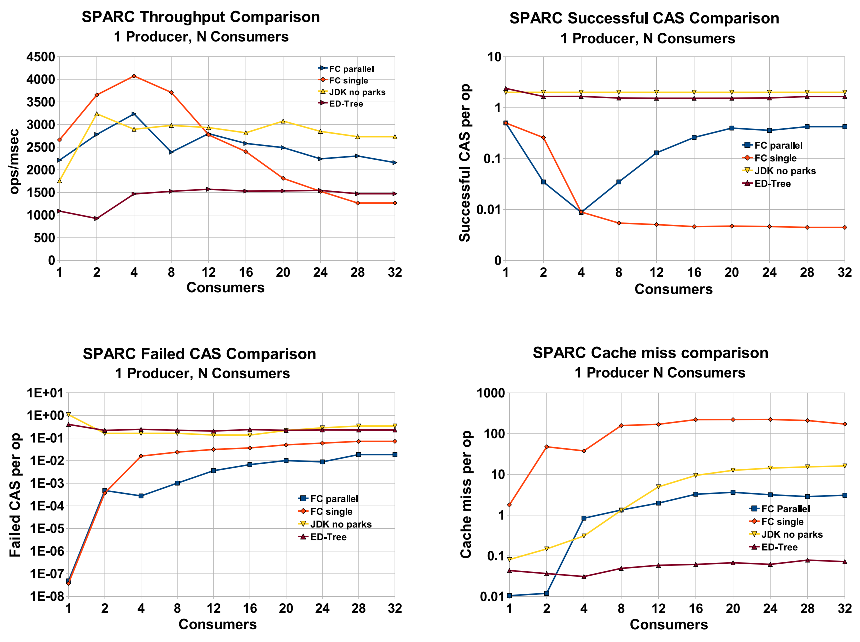

Fig. 4. Concurrent Synchronous Queue implementations with N consumers and 1 producer configuration: throughput, average CAS failures, average CAS successes, and L2 cache misses (all but throughput are logarithmic and per operation). Again, we show the throughput graphs for the JDK6.0 algorithm with parks, to make it clear that removing the parks helps performance in this benchmark.

We compared the JDK and an ED-Tree based implementation of a synchronous queue to two versions of flat combining based synchronous queues, an FC synchronous queue using a single FC mechanism (denoted in the graphs as *FC single*), and our dynamic parallel version denoted as *FC parallel*. The FC parallel threshold was set to spawn a new combiner sublist for every 8 new threads.

4.1 Producer-Consumer Benchmarks

Our first benchmark, whose results are presented in Figure 3, is similar to the one in [7]. Producer and consumer threads continuously push and pop items from the queues. In the throughput graph in the upper lefthand corner, one can clearly see that both FC implementations outperform JDK's algorithm even with the park operations removed. Thus, in the remaining sections, we will no longer compare to the inferior JDK6.0 algorithm with parks.

As can be seen, the FC single version throughput exceeds the JDK's throughput by almost 80% at 16 threads, and remains better as concurrency levels grow up to 64. However, because there is only a single combiner, the FC single algorithm does not continue to scale beyond 16 threads. On the other hand, the FC parallel version is the same as the JDK up to 8 threads, and from 16 threads and onwards it continues to scale, reaching peak performance at 48 threads. At 64 threads, the FC parallel algorithm is about 11 times faster than the JDK.

The explanation for the performance becomes clear when one examines the other three graphs in Figure 3. The numbers of both successful and failed CAS operations in the FC algorithms are orders of magnitude lower (the scale is logarithmic) than in the JDK, as is the number of L2 cache misses. The gap between the FC parallel and the FC single and JDK continues to grow as its overall cache miss levels are consistently lower than their, and its CAS levels remain an order of magnitude smaller then theirs as parallelism increases. The low cache miss rates of the FC parallel when compared to FC single can be attributed to the fact that the combining list is divided and is thus shorter, having a better chance of staying in cache. This explains why the FC parallel algorithm continues to improve while the others slowly deteriorate as concurrency increases. At the highest concurrency levels, however, FC parallel's throughput also starts to decline, since the cost incurred by a combiner thread that accesses the exchange increases as more combiner threads contend for it.

Since the ED-Tree algorithm is based on a static tree (that is, a tree whose size is proportional to the number of threads sharing the implementation rather than the number of threads that actually participate in the execution), it incurs significant overheads and has the worst performance among all evaluated algorithms in low concurrency levels.

However, for larger numbers of threads, the high parallelism and low contention provided by the ED-Tree allow it to significantly scale up to 16 threads, and to sustain (and even slightly improve) its peak performance in higher concurrency levels. Starting from 16 threads and on, the performance of the ED-Tree exceeds that of the JDK and it is almost 5-fold faster than the JDK for 64 threads.

Both FC algorithms are superior to the ED-Tree in all concurrency levels. Since ED-Tree is highly parallel, the gap between its performance and that of FC-single decreases as concurrency increases, and at 64 threads their performance is almost the same. The FC-parallel implementation, on the other hand, outperforms the ED-Tree implementation by a wide margin in all concurrency levels and provides almost three times the throughput at 48 threads.

Also here, the performance differences becomes clearer when we examine CAS and cache miss statistics (see Figure 3). Similarly to the JDK, the ED-Tree algorithm performs a relatively high number of successful CAS operations but, since its operations are spatially spread across the nodes of the tree, it incurs a much smaller number of failed CAS operations. The number of cache misses it incurs is close to that of FC-single implementation, yet is about 10-fold higher than FC-parallel implementation at high concurrency levels.

Figure 5-(a) shows the throughput on the Intel Nehalem architecture. As can be seen, the behavior is similar to that on SPARC up to 8 threads (recall that

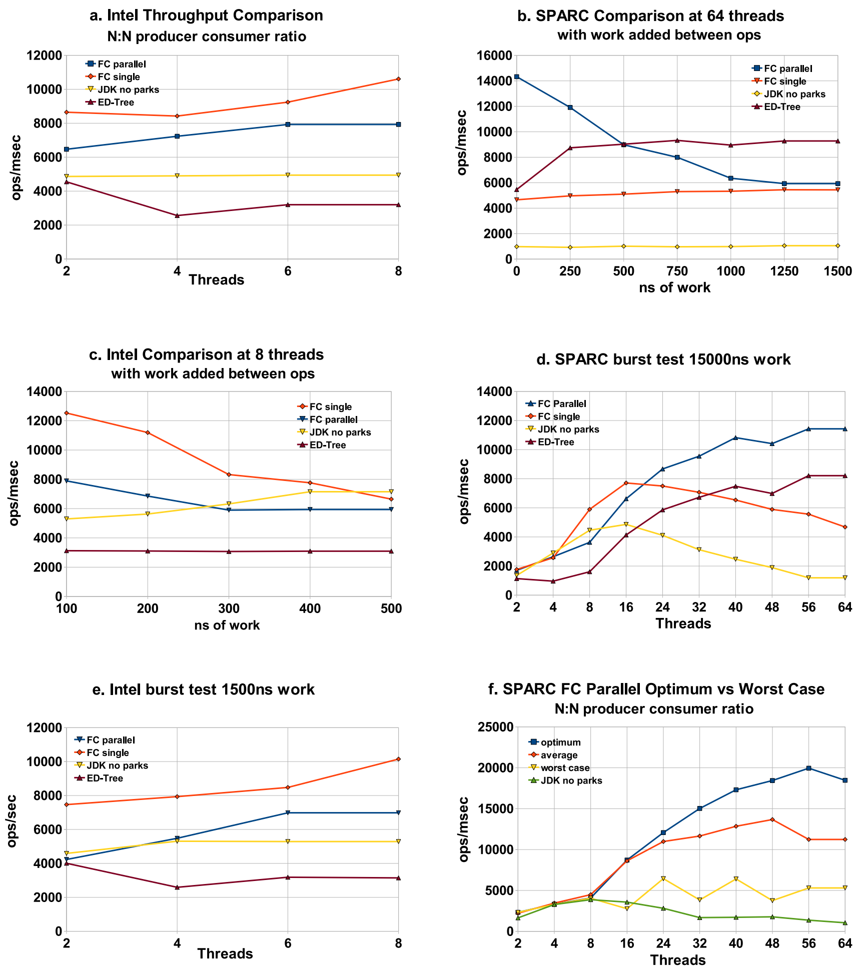

Fig. 5. (a) Throughput on the Intel architecture; (b) Decreasing request arrival rate on SPARC; (c) Decreasing request arrival rate on Intel; (d) Burst test throughput: SPARC; (e) Intel burst test throughput; (f) Worst-case vs. optimum distribution of producers and consumers.

the Nehalem has 8 hardware threads): the FC-single algorithm performs better than the FC-parallel, and both FC algorithms significantly outperform the JDK and ED-Tree algorithms. The cache miss and CAS rate graphs, not shown for lack of space, provide a similar picture.

Our next benchmark, in Figure 4, has a single producer and multiple consumers. This is not a typical use of a synchronous queue since there is much waiting and little possibility of parallelism. However, this benchmark, introduced in [7], is a good stress test. In the throughput graph in Figure 4, one can see that the imbalance in the single producer test stresses the FC algorithms, making the performance of the FC parallel algorithm more or less the same as that of the JDK. However, we find it encouraging that an algorithm that can deliver up to 11 times the performance of the JDK in the balanced case, delivers comparable performance when there is a great imbalance among producers and consumers.

What is the explanation for this behavior? In this benchmark there is a fundamental lack of parallelism even as the number of threads grows: in all of the algorithms, all of the threads but 2 - the producer and its previously matched consumer – cannot make progress. Recall that FC wins by having a single low overhead pass over the list service multiple threads. With this in mind, consider that in the single FC case, for every time a lock is acquired, about two requests are answered, and yet all the threads are continuously polling the lock. This explains the high cache invalidation rates, which together with a longer publication list traversed each time, explains why the single FC throughput deteriorates.

For the parallel FC algorithm, we notice that its failed CAS and cache miss rates are quite similar to those of the JDK. The parallel FC algorithm keeps cache misses and failed CAS rates lower than the single FC because threads are distributed over multiple locks and after failing as a combiner a thread goes to the exchange. In most lists no combining takes place, and requests are matched at the exchange level (an indication of this is the successful CAS rate which is close to 1), not in the lists. Combiners accessing the exchange take longer to release their list ownership locks, and therefore cause other threads less cache misses and failed CAS operations. The exchange itself is again a single combiner situation (only accessed by a fraction of the participating threads) and thus with less overhead. The result is a performance very similar to that of the JDK.

4.2 Performance as Arrival Rates Change

In earlier benchmarks, the data structures were tested at very high arrival rates. These rates are common to some uses of concurrent data structures, but not to all.

Figures 5-(b) and 5-(c) show the change in throughput of the various algorithms as the method call arrival rates change when running on 64 threads on SPARC, or on 8 threads on Intel, respectively. In this benchmark, we inject a "work" period between calls a thread makes to the queue. The work consists of a loop which is measured to take a specific amount of time.

On SPARC, at all work levels, both FC implementations perform better or the same as the JDK, and on the Nehalem, where the cost of a CAS operation is lower, they converge to a point with JDK winning slightly over the FC parallel algorithm. The ED-Tree is the worst performer on Nehalem. On SPARC, on the other hand, ED-Tree is consistently better than JDK and FC single and its performance surpasses that of FC parallel as the amount of work added between operations exceeds 500 nanoseconds.

In Figures 5-(d) an 5-(e) we stress the FC implementations further. We show a burst test in which a longer "work period" is injected frequently, after every 50 operations. This causes the nodes on the combining lists to be removed frequently, thus putting more stress onto the combining allocation algorithm. Again this slows down the FC algorithms, but, as can be seen, they still perform well.[5]

4.3 The Pitfalls of the Parallel Flat Combining Algorithm

The algorithmic process used to create the parallel combining lists is another issue that needs further inspection.

Since the exchange shared by all sublists is at its core a simple flat combining algorithm, its performance relies on the fact that on average not many of the requests are directed to it because of an imbalance in the types of operations on the various sublists. This raises the question of what happens at the best case - when every combiner enjoys an equal number of producers and consumers, and the worst case - in which each combiner is unfortunate enough to continuously have requests of the same type.

Figure 5-(f) compares runs in which the combining lists are prearranged for the worst and best cases prior to execution. As can be seen, the worst case scenario performance is considerately poor compared to the average and optimal ones. In cases were the number of combiners is even (16, 32, 48, 64 threads), performance solely relies on the exchange, and at one point it is worse than the JDK - this is most likely due to the overhead introduced by the parallel FC algorithm prior to the exchange algorithm. When the number of combiners is odd, there is a combiner which has both consumers and producers, which explains the gain in performance at 8, 24, 40, and 56 threads when compared to their successors. This yields the "saw" like pattern seen in the graph. Unsurprisingly, the regular run (denoted as "average") is much closer to the optimum.

In summary, our benchmarks show that the parallel flat-combining synchronous queue algorithm has the potential to deliver in the most common cases scalability beyond that achievable using fastest prior algorithms, and in the exceptional worst cases, under stress, they continue to deliver comparable performance.

[5] Due to the different speeds of the SPARC and INTEL machines, different "work periods" were required in the tests on the two machines in order to demonstrate the effect of bursty workloads.

5 Discussion

We presented a new parallel flat combining algorithm and used it to implement synchronous queues. The full code of our Java based implementation is available at `http://mcg.cs.tau.ac.il/projects/parallel-flat-combining`.

We believe that, apart from providing a highly scalable implementation of a fundamental data structure, our new parallel flat combining algorithm is an example of the potential for using multiple instances of flat combining in a data structure to allow continued scaling of the overhead-reducing properties provided by the flat combining technique. Applying the parallel flat combining paradigm to additional key data-structures is an interesting venue for future research.

Acknowledgments

We thank Doug Lea for allowing us to use his Sun Niagara 2 multicore machine. We also thank the anonymous reviewers for their many helpful comments. This paper was supported by the European Union grant FP7-ICT-2007-1 (project VELOX), as well as grants from Sun Microsystems, Intel Corporation, and a grant 06/1344 from the Israeli Science Foundation.

References

1. Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par '10, to appear*, June 2010.
2. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
3. D. R. Hanson. *C interfaces and implementations: techniques for creating reusable software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
4. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA '10: Proceedings of the Twenty Third annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, 2010.
5. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, NY, USA, 2008.
6. D. Lea. `util.concurrent.ConcurrentHashMap` in *java.util.concurrent* the Java Concurrency Package. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/-src/main/java/util/concurrent/.
7. W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.
8. W. N. Scherer III. *Synchronization and concurrency in user-level software systems*. PhD thesis, Rochester, NY, USA, 2006. Adviser-Scott, Michael L.
9. W. N. Scherer III and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *DISC*, pages 174–187, 2004.
10. N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.
11. N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

12. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
13. M. Tzafrir. C++ multi-platform memory-model solution with java orientation. http://groups.google.com/group/cpp-framework.