Flat Combining and the Synchronization-Parallelism Tradeoff

Danny Hendler
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Itai Incze
Tel-Aviv University
itai.in@gmail.com

Nir Shavit
Tel-Aviv University
shanir@cs.tau.ac.il

Moran Tzafrir
Tel-Aviv University
moran.tzafrir@cs.tau.ac.il

ABSTRACT

Traditional data structure designs, whether lock-based or lock-free, provide parallelism via fine grained synchronization among threads.

We introduce a new synchronization paradigm based on coarse locking, which we call *flat combining*. The cost of synchronization in flat combining is so low, that having a single thread holding a lock perform the combined access requests of all others, delivers, up to a certain non-negligible concurrency level, better performance than the most effective parallel finely synchronized implementations. We use flat-combining to devise, among other structures, new linearizable stack, queue, and priority queue algorithms that greatly outperform all prior algorithms.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Algorithms

General Terms

Algorithms, Performance

Keywords

Multiprocessors, Concurrent Data-Structures, Synchronization

1. INTRODUCTION

In the near future we will see processors with multiple computing cores on anything from phones to laptops, desktops, and servers. The scalability of applications on these machines is governed by *Amdahl's Law*, capturing the idea that the extent to which we can speed up any complex computation is limited by how much of the computation cannot be parallelized and must be executed sequentially. In many applications, the parts of the computation that are difficult to parallelize are those involving inter-thread communication via shared data structures. The design of effective concurrent data structures is thus key to the scalability of applications on multicore machines.

But how does one devise effective concurrent data structures? The traditional approach to concurrent data structure design, whether lock-based or lock-free, is to provide parallelism via fine grained synchronization among threads (see for example the Java concurrency library in the Java 6.0 JDK). From the empirical literature, to date, we get a confirmation of this approach: letting threads add parallelism via hand crafted finely synchronized data structure design allows, even at reasonably low levels of concurrency, to overtake the performance of structures protected by a single global lock [4, 14, 10, 12, 7, 16].

The premise of this paper is that the above assertion is wrong. That for a large class of data structures, the cutoff point (in terms of machine concurrency) at which finely synchronized concurrent implementations outperform ones in which access to the structure is controlled by a coarse lock, is farther out than we ever anticipated. The reason, as one can imagine, is the cost of synchronization.

This paper introduces a new synchronization paradigm which we call *flat combining* (FC). At the core of flat combining is a surprisingly low cost way to allow a thread to acquire a global lock on a structure, learn about all concurrent access requests, and then perform the combined requests of all others on it. As we show, 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" effect, up to high levels of concurrency, it outweighs the loss of parallelism caused by allowing only one thread at a time to manipulate the structure.

This paper will discuss the fundamentals of the flat combining approach and show a collection of basic data structures to which it can be applied. For lack of space we will not be able to show all the ways in which the approach can be further extended to support higher levels of concurrency and additional classes of structures.

1.1 Flat Combining in a nutshell

In its simplest form, the idea behind flat combining is to have a given sequential data structure D be protected by a lock and have an associated dynamic publication list of

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.
Copyright 2010 ACM 978-1-4503-0079-7/10/06 ...$10.00.

a size proportional to the number of threads that are concurrently accessing it (see Figure 1). Each thread accessing D for the first time adds a thread-local publication record to the list, and publishes all its successive access/modification requests using a simple write to the request field of its publication record (there is no need for a store-load memory barrier). In each access, after writing its request, it checks if the shared lock is free, and if so attempts to acquire it using a *compare-and-set* (CAS) operation. A thread that successfully acquires the lock becomes a *combiner*: it scans the list, collects pending requests, applies the *combined requests* to D, writes the results back to the threads' request fields in the associated publication records, and releases the lock. A thread that detects that some other thread already owns the lock, spins on its record, waiting for the owner to return a response in the request field, at which point it knows the published request has been applied to D. Once in a while, a combining thread will perform a cleanup operation on the publication list. During this cleanup it will remove records not recently used (we will later explain in more detail the mechanism for determining use times), so as to shorten the length of the combining traversals. Thus, in each repeated access request, if a thread has an active publication record, it will use it, and if not, it will create a new record and insert it into the list.

Our implementation of the flat combining mechanism allows us to provide, for any data structure, the same clean concurrent object oriented interface as used in the Java concurrency package [14] and similar C++ libraries [24]. As we show in Section 5, for any sequential object parallelized using flat combining, the resulting implementation will be *linearizable* [9] and given some simple real world assumptions also *starvation-free*. On an intuitive level, even though some method calls could be missed in the publication list by a concurrent scanning combiner, the fact that every method waits until its request is applied by the combiner, means that a later thread cannot get a response unless the missed earlier requests are still pending. These calls are therefore overlapping and can be linearized. Finally, note that flat combining 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.[1]

As a concrete example of an application of flat combining, consider the implementation of a concurrent linearizable FIFO queue, one of the most widely used inter-thread communication structures. The applied operations on a queue are enq() and deq(). The most effective known implementations are the lock-free Michael-Scott queue [17] used in the Java concurrency package, and the basket queue of Hoffman et. al [10]. Both suffer from a complete loss of scalability beyond a rather low concurrency level, because all threads must successfully apply a CAS operation to the shared head or tail of the queue in order to complete their operation.

In contrast, in flat combining, given a sequential FIFO queue D, the combining thread can collect all pending operations, then apply the combined requests to the queue. As our empirical comparison shows, unlike the fastest known concurrent implementations, in flat combining there is virtually no synchronization overhead, and the overall cache coherence traffic is significantly reduced.

The result, as seen in Figure 2, is a highly-scalable implementation of a linearizable FIFO queue that outperforms known implementations in almost all concurrency levels, and by a factor of more than 4 in high concurrency levels.

How broadly applicable is the flat combining paradigm? We claim that any data structure such that k operations on it, each taking time δ, can be combined and then applied in time *less than* $k\delta$, is a valid candidate to benefit from flat combining.[2] This applies to counters, queues, stacks etc, but also to linked lists, priority queues and so on. For example, in a linked list, all k collected modifications can be combined and applied in a single pass over the list, as opposed to k uncombined passes. In a skip-list-based priority queue, k remove-min operations can be combined and applied in $\Theta(k + \log n)$ as opposed to $\Theta(k \log n)$ uncombined ones, which as can be seen in Figure 5, outperforms prior art priority queue implementations considerably.

A further benefit of the flat combining approach is the fact that access to the core data structure being used remains sequential. This has implications on cache performance, but perhaps more importantly, on flat combining as a programming approach. As an example, considering again priority queues, we notice that one could build in a straightforward way a linearizable concurrent implementation of a state-of-the-art structure such as a pairing heap [3] by having all requests access it using flat combining. The proof of correctness of this flat combining pairing heap would be straightforward because it is only ever accessed sequentially. In contrast, based on our experience, devising a provably correct and scalable linearizable implementation of a concurrent pairing heap using fine-grained synchronization, would in itself constitute a publishable result. As seen in Figure 5, the flat-combining pairing-heap delivers impressive performance.

On the negative side, flat combining, at least in its simplest form, is not always applicable. For example, most search trees do not fit the above flat combining formula, since the cost of applying k search operations to a tree remains $\Theta(k \log n)$, while a fine grained concurrent implementation allows multiple non-blocking parallel search traversals to proceed in parallel in $\Theta(\log n)$ time. Furthermore, even in beneficially combinable structures, ones that have high levels of mutation on the data structure, there is a point on the concurrency scale in which the level of concurrency is such that a finely synchronized parallel implementation will beat the flat combining implementation. To make headway in these cases, advanced forms of flat combining are necessary, ones that involve multiple concurrent instances of flat combining. Such structures are the subject of future research.

1.2 Related Work

The idea of having a single thread perform the announced modifications of others on a concurrent data structure is not new. However, till now, the overheads of the suggested implementations have overshadowed their benefits.

The first attempt at combining operations dates back to the original software combining tree paper of Yew et. al [25]. In combining trees, requests to modify the structure

[1] 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.

[2] There are also cases of structures in which the combined cost is $k\delta$ which can to a certain extent benefit from flat combining.

are combined from the leaves of the tree upwards. The tree outputs $\frac{n}{\log n}$ requests per time unit. The resulting data structure is linearizable. However, each thread performs $\Theta(\log n)$ CAS based synchronization operations per access. This overhead was improved by Shavit and Zemach in their work on combining funnels [22], where trees of combined operations are built dynamically and are thus shallower. Nevertheless, as our empirical evidence shows, the synchronization overhead makes combining trees non-starters as a basis for concurrent data structure design.

A slightly different approach was taken by Oyama et. al [26]. They protect the data structure with a single lock and have threads form a list of requests on the lock. The thread acquiring the lock services the pending requests of others in LIFO order, and afterwards removes them from the list. The advantage of this approach is a lower cache miss rate on the accessed data structure because only a single thread accesses it repeatedly. Its main drawback is that all threads still need to perform CAS operations on the head of the list of requests, so scalability is mostly negative, but better than a naked lock. Since Oyama et. al were looking for a general compiler based transformation, they did not consider improved combining semantics for the data structures being implemented.

The next attempt to have a single thread perform the work of all others was the predictive log-synchronization approach of Shalev and Shavit [21]. They turned the Oyama et. al LIFO list into a FIFO queue, and added a prediction mechanism: threads do not wait for a response. Instead, they traverse the list of requests, predicting the outcome of their operation using the list as a log of the future modifications to be applied to the structure. To preserve the log structure, they allow no combining of operations.

The log-synchronization technique, however, is only sequentially consistent [13], not linearizable, and is limited to data structures to which effective prediction can be applied. As with Oyama et. al, the multiple CAS operations on the head of the queue limit scalability (the bottleneck on the log is similar to that the Michael and Scott queue [17]), and must be offset by a very high fraction of successfully predicting operations.

Unlike the above prior techniques, flat combining offers two key advantages:

- Threads do not need to succeed in a CAS on a shared location in order to have their request completed. In the common case, they only need to write and spin on a thread-local publication record, and read a single shared cached location (the lock).
- Cache invalidation traffic is reduced not only because the structure is accessed by a single thread while others spin locally, but also because the combiner thread can reduce the number of accesses due to the improved semantics of the combined operations.

In Section 6 we provide empirical evidence as to how the above two advantages lead to flat combining's superior performance.

2. FLAT COMBINING

Given a sequential data structure D, we design a *flat combining* (henceforth FC) concurrent implementation of the structure as follows.

As depicted in Figure 1, a few structures are added to the sequential implementation: 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.

Each thread t accessing the structure to perform an invocation of some method m on the shared object executes the following sequence of steps:

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 (there is 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.
   - Execute a *scanCombineApply()* by traversing the publication list from the 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. As we explain later, this traversal is guaranteed to be wait-free.
   - 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 node 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 with a successful CAS to the *head*. Then proceed to step 1.

We provide the particulars of the combining function and its application to a particular structure D in *scanCombineApply()* when we describe individual structures in later sections.

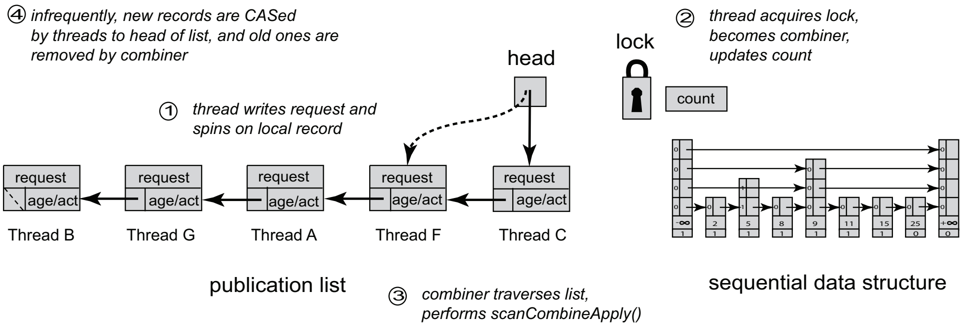
④ infrequently, new records are CASed by threads to head of list, and old ones are removed by combiner
head
lock
② thread acquires lock, becomes combiner, updates count
count
① thread writes request and spins on local record
request
age/act
Thread B
Thread G
Thread A
Thread F
Thread C
publication list
③ combiner traverses list, performs scanCombineApply()
sequential data structure

Figure 1: The 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.

There are a few points to notice about the above algorithm. 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.

In our implementation above we use a simple aging process completely controlled by combining threads. In general we expect nodes to be added and removed from the publication list infrequently. To assure this, one can implement a more sophisticated detection mechanism to control the decision process of when to remove a node from the list.

Nodes 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 [8]. 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 node is only ever changed by the thread holding the global lock. Note that the first item is not an anchor or dummy node, we are simply not removing it. Once new records are inserted, if it is unused it will be removed, and even if no new nodes are added, leaving it in the list will not affect performance.

The removal of a node is done by writing a special mark bit that indicates that the node is inactive after it is removed from the list. In order to avoid the use of memory barriers in the common case, we must take care of the case in which a late arriving thread writes its invocation after the record has been removed from the list. To this end, we make sure to mark a removed record as inactive, and make this mark visible using a memory barrier, so that the late arriving thread will eventually see it and add its record back into the publication list.

We now proceed to show how FC is applied to the design of several popular concurrent data structures.

3. FLAT COMBINING QUEUES AND STACKS

Shared queues lend themselves nicely to flat combining, since on the one hand they have inherent sequential bottlenecks that are difficult to overcome by using fine-grained synchronization, and on the other allow one to effectively apply combined sequences of operations.

We provide a linearizable FC FIFO queue implementations where the queue D is a linked list with *head* and *tail* pointers, and "fat" nodes that can contain multiple items in an indexed array (We can do this effectively because only a single combiner thread at a time will access this list. Using this approach in a concurrent setting would make the counter controlling traversal of the fat node entries a hot spot). To access the queue, a thread t posts the respective pair <ENQ,v> or <DEQ,0> to its publication record and follows the FC algorithm.

The combiner executing *scanCombineApply()* scans the publication list. For each non-null request, if it is an <ENQ,v> of value v, the item is added to the fat node at the tail of the queue, and if the queue is full a new one is allocated. An OK response is written back to the request field. If the request is a <DEQ,0>, an item is removed from the fat node at the head of the queue and written back to the request field. If the reader wonders why this is linearizable, notice that at the core of FC is the idea that if a posted request by some thread was missed by the combiner, then even if a request that started later was seen and successfully applied to the queue, the method call of the missed request is still pending, and so can be linearized after the one that succeeded.

Our concurrent linearizable LIFO stack implementation is similar to the FC queue algorithm described above. The data-structure is a linked list of fat nodes with a single *top* pointer. To push value v, a thread t writes the values-pair <PUSH,v> to its entry of the *combine* array. In the *scanCombineApply()*, the combiner services all requests from the fat node pointed to from the top.

As we show in Section 6, the low FC synchronization overhead and the locality of having a single thread at a time access a cached array of items in a fat node allow our FC stack and queue implementations to greatly outperform all prior-art algorithms.

4. FLAT COMBINING PRIORITY QUEUES

A *priority queue* is a multiset of items, where each item has an associated priority – a score that indicates its importance (by convention, the smaller the score, the higher the priority). A priority queue supports two methods: *add*, for adding an item to the multiset, and *removeMin*, for removing and returning an item with minimal score (highest importance). For presentation simplicity, in the descriptions that follow we assume that each item only stores a *key*, rather than storing both a key and priority value.

The prior art in this area includes a linearizable heap algorithm by Hunt et. al [11] and a skiplist-based priority queue due to Shavit and Lotan [15].

We present two FC based linearizable concurrent priority queue implementations, one based on a skiplist [19] and another based on a pairing heap [3].

4.1 A Flat Combining Skiplist Based Priority Queue

A *skiplist* [19] is a probabilistic data-structure, representing a multiset of nodes, ordered by their keys. A skiplist appears in the righthand side of Figure 1. The skiplist is composed of a collection of sorted linked lists. Each node consists of a key and an array of pointers that link it to a subset of the lists. Each list has a *level*, ranging from 0 to some maximum. The bottom-level list contains all the nodes, and each higher-level list is a sublist of the lower-level lists. The higher-level lists can be viewed as *shortcuts* into the lower-level lists. The skiplist has two sentinel nodes of maximal height with minimal and maximal keys. Skiplists support *add*, *remove*, and *find* methods.

One could readily implement a priority queue that supports *add* and *removeMin* using a skiplist by having *add* methods add an item to the skiplist, and have the *removeMin* method use the bottom (lowest level) pointer in the sentinel with the minimal key, then removing the node it points to from the skiplist, as this is the minimal key at any given point. The cost of both *add* and *removeMin* are $\Theta(\log n)$, as one needs to traverse the skiplist from the highest level to add or remove an item.

For our use, we implement *removeSmallestK* and *combinedAdd* methods. The *removeSmallestK* method receives as its parameter a number k and returns a list consisting of the k smallest items in the skiplist. It does so as follows:

- Traverse the lowest level of the skiplist from the minimal sentinel until k items (or fewer if the skiplist contains less than k items) have been traversed. Collect these items, they are the k minimal items to be returned.

- Perform a *find* traversal of the skiplist, searching for the key of the $k+1$-th minimal item found. The *find* operation is called for locating the predecessors and successors of *newMin* in all the linked lists.[3] In all the lists which do not contain *newMin*, the *head* node is set to point to the first node greater than *newMin*.

We observe that the combined removal performed by the *removeSmallestK* method searches only once in each linked list. It thus removes the k minimal items while performing work corresponding to *a single removal*. The overall complexity of *removeSmallestK* is thus $\Theta(k + \log n)$ as opposed to $\Theta(k \log n)$ for k uncombined removals from a skiplist.

The *combinedAdd* method will also reduce the overall cost of adding k nodes by making only a single pass through the skiplist. It receives a list of items $<i_1, i_2, \ldots, i_k>$, sorted in increasing order of their keys, and inserts them to the skiplist. This is done as follows. First, a highest level is selected at random for each of the k items. Then, the appropriate location of i_1 in the skiplist is searched, starting from the topmost list. The key idea is that, once the search arrives at a list and location where the predecessor and successor of i_1 and i_2 differ, that location is recorded before the search of i_1 proceeds. Once the search of i_1 terminates, the search of i_2 is resumed starting from the recorded location instead of starting from the top list again. This optimization is applied recursively for all subsequent items, resulting in significant time saving in comparison with regular addition of these k items.

The implementation of our *scanCombineApply* is immediate: when scanning the publication list, collect two respective lists, of the requests to *add* and of those to *removeMin*. Count the number of requests and apply the *removeSmallestK*, and sort the linked list of *add* requests[4], then apply the *combinedAdd* method, returning the appropriate results to the requesting threads.

4.2 A Flat Combining Pairing Heap Based Priority Queue

But this is not the end of the story for priority queues. The power of FC as a programming paradigm is that the data structure D at the core of the implementation is sequential. This means we can actually use the most advanced priority queue algorithm available, without a need to actually think about how to parallelize it. We thus choose to use a *pairing heap* [3] structure, a state-of-the-art heap structure with $O(1)$ amortized *add* complexity and $O(\log n)$) amortized *removeMin* complexity with very low constants, as the basis for the FC algorithm. We will not explain here why pairing heaps have impressive performance in practice and readers interested in analytic upper bounds on their complexity are referred to [18]. We simply took the sequential algorithm's code and plopped it as-is into the FC framework. The result: a fast concurrent priority queue algorithm that by Lemma 1 is provably linearizable. As we noted earlier, based on our experience, devising a provably correct and scalable linearizable implementation of a concurrent pairing heap using fine-grained synchronization would in itself constitute a publishable result. We hope a similar approach can be taken towards creating linearizable concurrent versions of other complex data structures.

5. FLAT COMBINING CORRECTNESS AND PROGRESS

In this section we outline correctness and progress arguments for the flat combining technique. For lack of space, we do not cover the details of the management of the publication list, and choose to focus on outlining the linearizable and starvation-free behavior of threads with active publication records. We also confine the discussion in this section

[3] This is the regular semantics of the *find* operation on skiplists.

[4] Our current implementation uses bubble sort.

to FC implementations that protect the publication list by a single lock.

Linearizability [9] is widely accepted as a correctness condition of shared object implementations. Informally, it states that, in every execution of the object implementation, each operation on the implemented object appears to take effect instantaneously at some (*linearization*) *point* between the operation's invocation and response (or possibly not at all if the operation is pending at the end of the execution).

To prove that an implementation is linearizable, one must show that a *linearization* can be found for each of the implementation's executions. An execution linearization is defined by specifying linearization points for all complete operations (and possibly also for some pending operations), such that the operations' total order defined by the linearization is a legal sequential history of the implemented object.

We say that an implementation of the *scanCombineApply* method is *correct* if it satisfies all the following requirements: (1) it operates as described in Section 2, (2) it does not access the FC lock protecting the publication list, and (3) it returns operation responses in the order in which it applies the operations to the data structure.

LEMMA 1. *An FC implementation using a correct scanCombineApply method is linearizable.*

Proof outline. The use of a single FC lock protecting the publication list and the correctness of the *scanCombineApply* method guarantee that executions of the *scanCombineApply* method are sequential, since there is at most a single combiner thread at any given time. A combiner thread t sequentially applies a set of combined operations to the data-structure representing the implemented object in the course of performing the *scanCombineApply* method. We linearize each applied operation just before the combiner writes its response to the respective publication record.

The total order specified by the resulting linearization is clearly a legal sequential history of the implemented object, since, from the correctness of the *scanCombineApply* method, the operations are sequentially applied to the implemented object in exactly this order.

To conclude the proof, we show that each linearization point lies between the invocation and response of the corresponding operation. This holds trivially for the operation of the combiner. As for operations of non-combiner threads, it follows from the FC algorithm and the correctness of the *scanCombineApply* method that each such thread starts spinning on its record before its linearization point and can only return a response after the combiner has written to its publication record, which happens after its linearization point.

LEMMA 2. *An FC implementation using a correct scanCombineApply method is starvation-free.*

Proof outline. Since nodes can only be inserted to the publication list immediately after its head node, *scanCombineApply* is wait-free. Let t be a thread whose publication record is constantly active. If t becomes a combiner, the claim is obvious. If t fails to acquire the lock, then there is another active combiner t'. From the correctness and wait-freedom of the *scanCombineApply* method, either t' or the subsequent combiner thread (possibly t itself) will apply t's operation and write the response to t's publication record.

6. PERFORMANCE EVALUATION

For our experiments we used two machines. The first is a 128-way Enterprise T5140® server (Maramba) machine running Solaris^{TM} 10. This is a 2-chip Niagara system in which each chip has 8 cores that multiplex 8 hardware threads each and share an on chip L2 cache. We also used an Intel Core2® i7 (Nehalem) processor with 4 cores that each multiplex 2 hardware threads. We ran benchmarks in C++ using the same compiler on both machines, and using the scalable *Hoard* memory allocator [1] to ensure that malloc calls are not a sequential bottleneck.

6.1 Flat combining versus prior techniques

We begin our empirical evaluation by examining the relative performance of FC with respect to prior known parallelization techniques. We start by presenting data for implementations of a concurrent FIFO queue. The presented graphs are the average of 3 executions, each taking 10 seconds.

We evaluated a flat combining based queue (denoted as *fc*) as described in Section 3. We compared it to the most effective known queue implementations: the lock-free queue by Michael and Scott [20] (henceforth the MS-queue) used in the Java concurrency package and the basket queue of Hoffman et. al [10]. Michael and Scott's lock-free queue algorithm represents a queue as a singly-linked list with *head* and *tail* pointers, which are modified by CAS operations. The basket queue algorithm permits higher concurrency levels by allowing enqueue operations to operate on multiple *baskets* of mixed-order items instead of a single central location. We also compared them to implementations of queues using prior global lock based techniques: Oyama et. al's algorithm [26], combining trees [25], and Shalev and Shavit's predictive log-synchronization [21], as described in Section 1.2.

As the reader may recall, the Oyama et. al technique created a list of requests, each added using a CAS operation, and then operated on these requests one after the other. It does not combine requests. In order to understand which fraction of the FC advantage comes from the publication list mechanism itself, and which from the gain in locality by having a single combiner access the structure, we added a version of Oyama et. al in which we add the same combining feature FC uses. In the case of a queue, this is the use of "fat nodes" that hold multiple items to collect requests, and add these many requests in one pointer swing of the fat node into the structure.

In the throughput graph in Figure 2, one can clearly see that from about 4 threads and on the flat combining implementation starts to increasingly outperform all others and is from 4 to 7 times faster than the MS queue, the fastest of the prior techniques. Moreover, one can see that Oyama et. al, combining trees, and log-synchronization do not scale. In fact, Oyama scales only a bit better when we add the combining feature. This leads us to the conclusion that, at least for queues, the main advantage of FC is in the low overheads of the publication mechanism itself.

The explanation for the significantly better FC behavior is provided in the other three graphs in Figure 2. As can be seen, the MS queue requires on average 1.5 successful CASes per operations (this is consistent with the MS queue code), but suffers increasing levels of CAS failures, as does the basket queue. The combining tree requires increasing numbers of successful CASes as the tree grows to accommo-

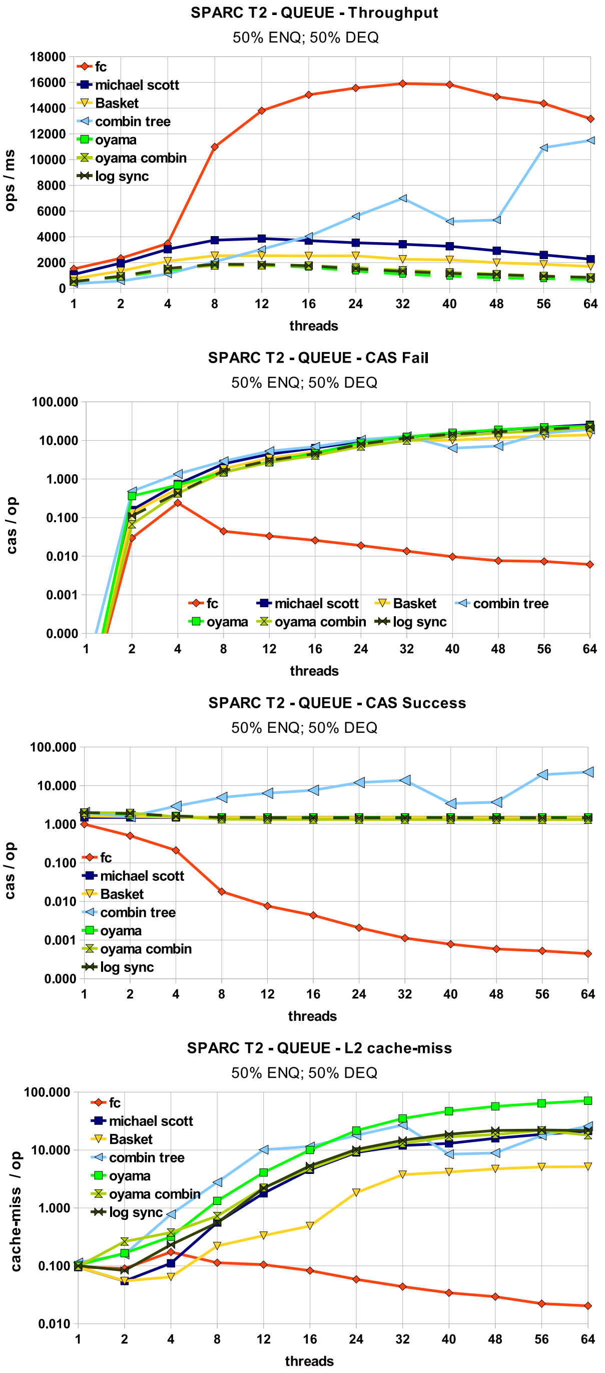

Figure 2: Concurrent FIFO Queue implementations: throughput (number of operations), average CAS failures (per operation), average CAS successes (per operation), and L2 cache misses (per operation).

date more threads. The number of failed CASes increases significantly as concurrency grows, explaining the tree's poor behavior. Similar failed CAS overheads hurt Oyama et. al. Log-synchronization requires only 3 successful CASes on average, and because of prediction threads do not compete for the lock, avoiding high CAS failure rates, and eventually overtaking the performance of the MS and basket queue algorithms.

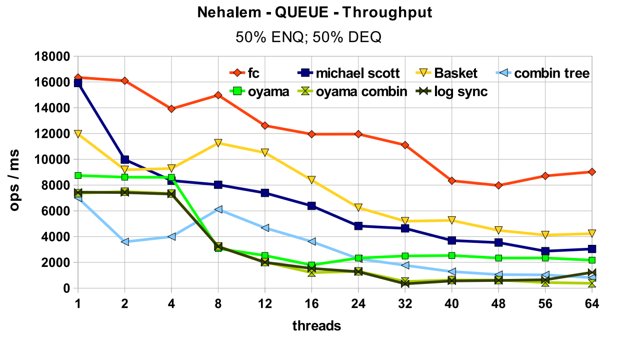

Figure 3: FIFO Queue throughput on the Nehalem architecture.

The most telling graph though is that of the L2 cache miss rates on the Niagara architecture, a dominant performance factor in multicore machines. Notice that the graph uses a logarithmic scale. As we can see, Oyama et. al and combining trees have two or more orders of magnitude more cache misses than the FC algorithm. Even the best non-FC technique, the MS-queue, has at some concurrency levels about two orders of magnitude more cache misses than the FC algorithm.

Unlike other general techniques, the flat combining implementation requires on average almost no CAS successes to complete, and has a negligible CAS failure rate. Its cache miss rate is very low. It is therefore not by chance that the FC queue outperforms the best hand crafted solution on the Niagara architecture by a wide margin.

Figure 3 shows similar behavior on the Nehalem architecture. Here we see that all the algorithms exhibit negative scalability, and yet the FC algorithm is again superior to all others. The cache miss and CAS rate graphs we do not present provide a similar picture to that on the Niagara.

6.2 Stacks

We now consider linearizable concurrent LIFO Stacks. We compare our flat-combining queue with Treiber's lock-free stack implementation [23] (denoted as 'lock free' in the graphs). Treiber's algorithm represents the stack as a singly-linked list pointed at by a top pointer which is manipulated by CAS operations. We also compare to Hendler et. al's [6] linearizable elimination-backoff stack.

Figure 4 shows the throughput of the flat combining stack, the elimination-backoff stack, and Treiber stack on the two platforms. On the Maramba (Sparc) machine flat combining clearly outperforms Treiber's algorithm by a wide margin (a factor of 9) because Treiber's algorithm *top* is a CAS synchronization bottleneck. The performance of the elimination-backoff stack algorithm improves to reach that of flat combining, since the benchmark supplies increasing

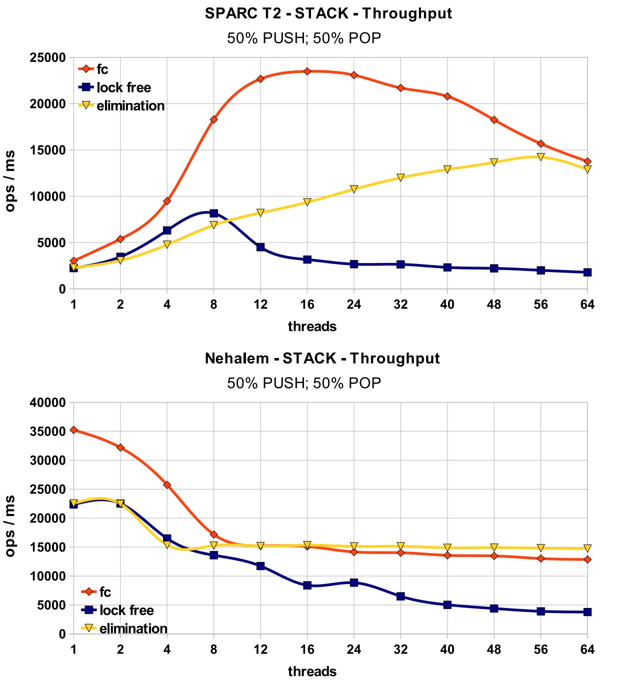

Figure 4: The throughput of LIFO stack implementations.

equal amounts of concurrent pushes and pops that can be eliminated. Note that with a different ratio of pushes and pops the elimination queue will not perform as well. However, as can be seen, at lower concurrency rates the flat combining stack delivers more than twice the throughput of elimination-backoff stack. On the Nehalem machine, flat combining is the clear winner until there are more software threads than hardware threads, at which point its performance becomes about the same as that of the elimination-backoff stack.

6.3 Priority Queues

We compare our two flat combining priority queue implementations from Section 4: the skiplist based one (denoted as *fc skiplist*) and the pairing-heap based one (denoted as *fc pairing heap*) with the best performing concurrent priority queue in the literature, the skiplist-based concurrent priority queue due to Lotan and Shavit [15] (its performance has been shown to be significantly better than that of the linearizable concurrent heap algorithm due to Hunt et. al [11]). The Lotan and Shavit algorithm has threads add items based on their priorities to the skiplist. Each item occupies a node, and the node has a special “logically deleted” bit. To remove the minimal items, threads compete in performing a series of CAS operations along the bottom level of the skiplist. Each thread does that until it encounters the first non-deleted node it manages to CAS to a deleted state. It then performs a call to the regular skiplist remove method to remove the node from the list. We compare to two versions of the [15] algorithm, one in which the skiplist is lock-free and another in which it is a lazy lock-based skiplist (see [8] for details). These priority queue algorithms are quiescently consistent but *not* linearizable [8], which gives them an advantage since making them linearizable would require to add an access to a global clock variable per insert which would cause a performance deterioration.

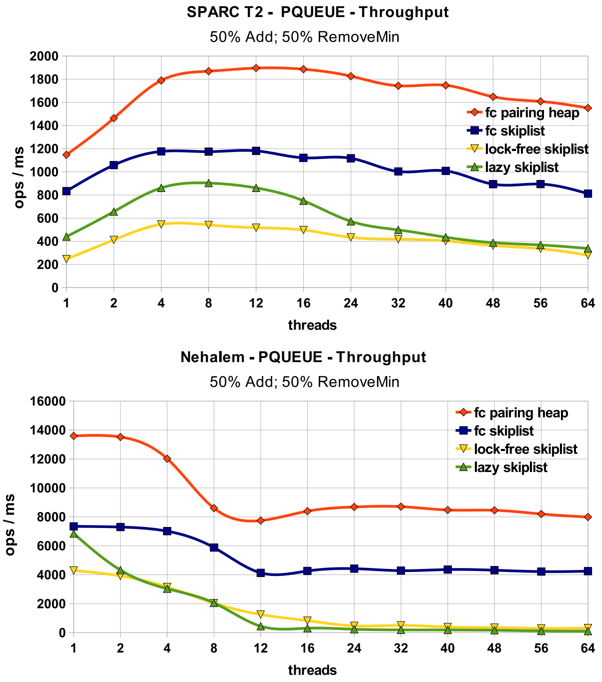

Figure 5: The throughput of Priority Queue implementations.

Figure 5 shows the throughput of the priority queue algorithms on the two platforms. The lazy lock-based skiplist outperforms the lock-free one since it requires fewer CAS synchronization operations per add and remove. This reduces its overhead and allows it to be competitive at low concurrency levels since it has a certain degree of parallelism and a reasonable overhead. However, as concurrency increases the competition for marking nodes as logically deleted, together with the fact that most remove-min operations occur on the same nodes at the head of the list, causes a deterioration. As concurrency increases, the algorithm is increasingly outperformed by the FC skiplist algorithm. The surprising element here is that that best performance comes from the pairing heap based algorithm. The use of flat combining allows us to make direct use of this sequential algorithm’s great efficiency, $O(1)$ for an insert, and an amortized $O(\log n)$ per remove, without paying a ridiculously high synchronization price in a fine grained implementation (assuming a scalable and provably linearizable fine grained pairing heap implementation is attainable at all).

6.4 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. In Figure 6, we return to our queue benchmark to show the change in throughput of the various algorithms as the method call arrival rates change when running on 32 threads. In this benchmark, we inject a “work” period between calls a thread makes to the queue. The work consists of an equal number of reads and writes to random locations. As can be seen, as the request arrival rate decreases (the total number of reads and writes depicted along the x-axis increases), the relative performance advantage of the FC algorithm decreases, while all other methods mostly remain unchanged (apart from log-synchronization, which, as it seems, benefits from the reduced arrival rate as there is less contention on the queue and less cache invalidation behavior while threads traverse the log during prediction). Nevertheless, the FC queue algorithm remains superior or equal to all other methods throughout.

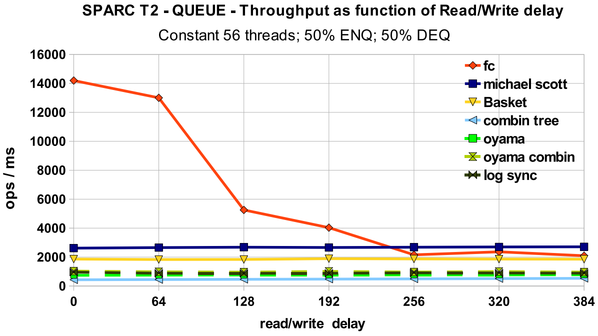

Figure 6: Performance as Arrival Rates Change. Work along the X-axis is the number of reads and writes between accesses to the shared queue.

7. DISCUSSION

We presented *flat combining*, a new synchronization paradigm based on the idea that having a single thread holding a lock perform the combined access requests of all others, delivers better performance through a reduction of synchronization overheads and cache invalidations. There are many ways in which our work can be extended, and we encourage others to continue this work.

Obviously there are many data structures that could benefit from the FC approach, we have only presented a few representative ones.

An interesting aspect is that FC will be a natural fit with the heterogenous multicore architectures of the future: the more powerful cores can get a preference in acquiring the global lock, or in some cases be assigned as combiners in a static way.

The flat combining process itself could benefit from a dynamic/reactive ability to set its various parameters: the number of combining rounds a combiner performs consecutively, the level of polling each thread performs on the combining lock, which threads get higher priority in becoming combiners, and so on. One way to do so is to perhaps use the machine learning-based approach of *smartlocks* [2].

Another interesting line of research is the use of multiple parallel instances of flat combining to speed up concurrent structures such as search trees and hash tables. Here the idea would be an extension of the style of our queue implementation: have a shared data structure but reduce overhead in bottleneck spots using flat combining. We have recently been able to show how this parallel flat-combining approach can benefit the design of unfair synchronous queues [5].

Finally, it would be interesting to build a theoretical model of why flat combining is a win, and what its limitations are.

8. ACKNOWLEDGEMENTS

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.

9. REFERENCES

[1] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not. 35*, 11 (2000), 117–128.

[2] Eastep, J., Wingate, D., Santambrogio, M., and Agarwal, A. Smartlocks: Self-aware synchronization through lock acquisition scheduling. In *4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMARTŠ10)* (2009).

[3] Fredman, M. L., Sedgewick, R., Sleator, D. D., and Tarjan, R. E. The pairing heap: A new form of self-adjusting heap. *Algorithmica 1*, 1 (1986), 111–129.

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

[5] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. Scalable flat-combining based synchronous queues, 2010.

[6] Hendler, D., Shavit, N., and Yerushalmi, L. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2004), ACM, pp. 206–215.

[7] Herlihy, M., Lev, Y., and Shavit, N. A lock-free concurrent skiplist with wait-free search, 2007.

[8] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, NY, USA, 2008.

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

[10] Hoffman, M., Shalev, O., and Shavit, N. The baskets queue. In *OPODIS* (2007), pp. 401–414.

[11] Hunt, G. C., Michael, M. M., Parthasarathy, S., and Scott, M. L. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett. 60*, 3 (1996), 151–157.

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

[13] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (September 1979), 690.

[14] Lea, D. `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/.

[15] Lotan, I., and Shavit., N. Skiplist-based concurrent priority queues. In *Proc. of the 14th International*

Parallel and Distributed Processing Symposium (IPDPS) (2000), pp. 263–268.

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

[17] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing* (1996), pp. 267–275.

[18] Pettie, S. Towards a final analysis of pairing heaps. In *Data Structures* (Dagstuhl, Germany, 2006), L. Arge, R. Sedgewick, and D. Wagner, Eds., no. 06091 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

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

[20] Scott, M. L., and Scherer, W. N. Scalable queue-based spin locks with timeout. *ACM SIGPLAN Notices 36*, 7 (2001), 44–52.

[21] Shalev, O., and Shavit, N. Predictive log synchronization. In *Proc. of the EuroSys 2006 Conference* (2006), pp. 305–315.

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

[23] Treiber, R. K. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center, April 1986.

[24] Tzafrir, M. C++ multi-platform memory-model solution with java orientation. http://groups.google.com/group/cpp-framework.

[25] Yew, P.-C., Tzeng, N.-F., and Lawrie, D. H. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput. 36*, 4 (1987), 388–395.

[26] Yonezawa, O. T., Oyama, Y., Taura, K., and Yonezawa, A. Executing parallel programs with synchronization bottlenecks efficiently. In *in Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99). Sendai, Japan: World Scientific* (1999), World Scientific, pp. 182–204.