Lock Oscillation: Boosting the Performance of Concurrent Data Structures[†]

Panagiota Fatourou[1] and Nikolaos D. Kallimanis[2]

1 **Institute of Computer Science (ICS), Foundation of Research and Technology-Hellas (FORTH), and Department of Computer Science, University of Crete, Greece**
faturu@csd.uoc.gr

2 **Institute of Computer Science, Foundation of Research and Technology-Hellas, Crete, Greece**
nkallima@ics.forth.gr

Abstract

In *combining-based synchronization*, two main parameters that affect performance are the *combining degree* of the synchronization algorithm, i.e. the average number of requests that each *combiner* serves, and the number of expensive synchronization *primitives* (like `CAS`, `Swap`, etc.) that it performs. The value of the first parameter must be high, whereas the second must be kept low.

In this paper, we present Osci, a new combining technique that shows remarkable performance when paired with cheap context switching. We experimentally show that Osci significantly outperforms all previous combining algorithms. Specifically, the throughput of Osci is higher than that of previously presented combining techniques by more than *an order of magnitude*. Notably, Osci's throughput is much closer to the ideal than all previous algorithms, while keeping the average latency in serving each request low. We evaluated the performance of Osci in two different multiprocessor architectures, namely AMD and Intel.

Based on Osci, we implement and experimentally evaluate implementations of concurrent queues and stacks. These implementations outperform by far all current state-of-the-art concurrent queue and stack implementations. Although the current version of Osci has been evaluated in an environment supporting user-level threads, it would run correctly on any threading library, preemptive or not (including kernel threads).

1998 ACM Subject Classification D.1.3 Programming Techniques - Concurrent Programming

Keywords and phrases Synchronization, concurrent data structures, combining

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.8

1 Introduction

The development of efficient parallel software has become a necessity due to the dominance of multicore machines. One obstacle in achieving good performance when introducing parallelism in modern applications comes from the synchronization cost incurred by those parts of the application that cannot be parallelized. Efficient synchronization mechanisms are then required to maintain this synchronization cost low.

[†] This work has been supported by the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the ExaNoDe project (grant agreement 671578) and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

Whenever a parallel application wants to access shared data, a synchronization request is initiated. In order to avoid races, these requests must be executed in mutual exclusion. Therefore, a lower bound on the time to execute m such requests is the time it takes for a single thread to sequentially execute them, sidestepping the cost of the synchronization protocol. An *ideal* synchronization protocol would not require more time than this, independently of the number of the active threads and despite any contention on the accessed data. In practice, even the best current synchronization protocols cause a drastic reduction in performance, even in low contention.

Recent work [8, 9, 14] has focused on developing synchronization protocols implementing the *combining* synchronization technique (first realized in [11, 31]). This technique has been argued [8, 9, 14] to be very efficient. A combining synchronization protocol maintains a list to store the pending synchronization requests issued by the active threads. A thread first announces its request by placing a node in the list, and then tries to acquire a global lock. If it does so, it becomes a *combiner*, and serves not only its own synchronization request, but also active requests announced by other threads. Each thread that has not acquired the lock, busy waits until either its request is executed by a combiner or the global lock is released.

CC-Synch [9] is a simple implementation of the combining technique which outperforms previous state-of-the-art combining algorithms including flat-combining [14] and OyamaAlg [26], and other synchronization mechanisms such as CLH-locks [7, 19] and a simple lock-free algorithm [9]. Nevertheless, Figure 1 indicates that the performance of CC-Synch is still far from the ideal in a multicore machine equipped with 64 processing cores (more details on this experiment are provided in Section 5); the ideal performance is measured by calculating the time that it takes to a single thread to execute the total number of synchronization requests (that are to be executed by all threads) sidestepping the synchronization protocol, as well as the local work that follows each of the synchronization requests that it has to perform itself.

In this paper, we present a technique that significantly enhances the performance of combining-based synchronization by reducing the number of expensive synchronization primitives such as `Compare&Swap` (`CAS`) or `Swap` that are performed on the same shared memory location, resulting in much fewer cache-misses and cpu backend stalls. Yet, this technique results in algorithms that are as simple as using CC-Synch or any other synchronization technique. The technique enables batching of the synchronization requests initiated by threads running on the same core. The requests are batched in a single "fat" node, which is then appended in the list of the announced synchronization requests by performing just a single expensive synchronization primitive.

We study the impact on performance of this technique when combined with cheap context switching. Specifically, we experimentally argue that its performance power is remarkable when employed in an environment supporting user-level threads. We present Osci, a new combining synchronization protocol which exploits this technique. Osci exhibits performance that is surprisingly closer to the ideal than previous combining algorithms. We experiment with Osci in a setting where a kernel-level thread running on each core has spawned a number of user-level threads. Osci appropriately schedules the threads, using a fair implementation of `Yield` (whenever a `Yield` is executed, the running thread voluntarily gives the control of the core that it is running to some other thread), to achieve better performance.

Osci ensures that a thread p from the set P_c of threads running on a core c, initializes the contents of a node nd and informs other threads in P_c that they can record their requests there. So, p initiates a recording period for nd. Next time p is scheduled, it informs threads in P_c that the recording period for nd is over and appends nd to the shared list of requests. Thus, when nd is appended in the list, more than one requests by threads in P_c may have been recorded in it. The combiner traverses the request list and serves all requests recorded

Table 1 Summary of the known combining techniques.

Algorithm	Primitives	Scheduling-Aware
Blocking Algorithms		
OyamaAlg [26]	`CAS`, `read-write`	No
flat-combining [14]	`CAS`, `read-write`	No
CC-Synch [9]	`Swap`, `read-write`	No
DSM-Synch [9]	`Swap`, `CAS`, `read-write`	No
Osci (this paper)	`Swap`, `CAS`, `read-write`	Yes
Wait-Free Algorithms		
PSim [8]	`LL/SC`, `read-write`, `Fetch&Add`	No
PSim-x (this paper)	`LL/SC`, `read-write`, `Fetch&Add`	Yes

in each of the list nodes. Each of the other threads performs local spinning until either its request is served by a combiner or the thread becomes the new combiner.

In modern NUMA architectures, the execution of a synchronization primitive (`CAS`, `Swap`, etc.) on a shared memory location causes expensive cache misses and costs at least a few thousands of cpu cycles. Specifically, such primitives usually involve a flooding of invalidation messages performed by the coherence protocol [17] to those cores that keep a copy of the contents of the shared memory location in their caches (`read` is usually cheaper since it avoids causing any invalidation messages). In contrast to the expensive synchronization primitives, a context switch (`Yield`) among user-level threads running on the same core may cost no more than a hundred cpu cycles. In most locking and combining protocols, every request issued by a thread performs at least one expensive synchronization instruction on a common shared variable, resulting in a cache-line invalidation. For instance, in a queue lock, this shared variable is the Tail of the queue. Thus, the application (or the announcement) of k requests results in (at least) k cache invalidations, causing contention and increased traffic on the interconnection network. By using cheap context switching, Osci creates fat nodes containing batches of requests. It does so by attempting to pass the control of the processor to all threads that are running on the same core, thus enabling them to announce new requests at a very low cost. Each fat node may contain up to $t > 1$ requests (all issued by the user-level threads that are running on the same core). In this way, Osci substantially reduces the synchronization cost for announcing and applying batches of requests. Specifically, k requests may cause only k/t cache line invalidations for their announcement, resulting to substantially reduced coherence traffic on the interconnection network. As an immediate consequence, Osci does not only achieve high combining degree but it also causes the smallest amount of cache misses (all cache levels), due to low number of cache invalidations, and the smallest amount of backend stalls than all the other protocols (see Section 5 for a more detailed performance analysis).

We experimentally compare Osci with known synchronization algorithms, i.e. CC-Synch [9], PSim [8], flat-combining [14], OyamaAlg [26], a blocking `Fetch&Multiply` implementation based on CLH spin-locks [7, 19], and a simple lock-free implementation (see Table 1). (MCS locks [21] exhibit similar performance to CLH locks.) Osci outperforms CC-Synch by a factor of up to 11x (Figure 4b) without increasing the latency of CC-Synch. The performance advantages of Osci over all other algorithms are even higher. It is noticeable that Osci's performance is not worse than that of CC-Synch when context switching is expensive (e.g. in systems that do not support user-level threads). It is worth noting that employing user-level threads in previous algorithms does not significantly improve their performance (see Table 2).

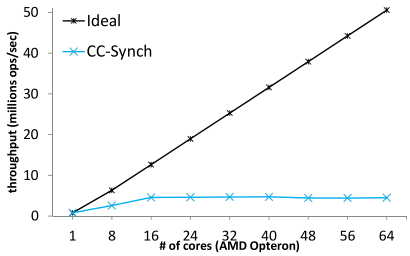


Figure 1 Average throughput of the ideal implementation, and the CC-Synch [9] implementation of a `Fetch&Multiply` object on a 64-core machine.

Osci is linearizable [15] (see Section 4 for a sketch of proof). *Linearizability* ensures that in every execution α, each synchronization request executed in α, appears to take effect, instantaneously, at some point in its execution interval.

We used Osci to get an implementation of a queue (OsciQueue) and a stack (OsciStack). Experiments show that OsciQueue outperforms all current state-of-the-art implementations of queues (combining-based or not), including LCRQ [25], CC-Queue [9], SimQueue [8], and the lock-free queue in [22]. Section 5 reveals that OsciQueue is more than 4 times faster than LCRQ [25], the state-of-the-art concurrent queue implementation. OsciStack exhibits performance advantages similar to that of OsciQueue.

We also present PSim-x, a simple variant of PSim [8]. PSim is a practical universal construction which implements the combining technique in a *wait-free* manner (*wait-freedom* ensures that each active thread manages to complete the execution of each of its requests in a finite number of *steps*). PSim can simulate any concurrent object given that only a small part of the object's state is updated by each request. In environments providing fast context switching, PSim-x achieves a significantly increased combining degree in comparison to PSim. This is done by using oversubscribing and applying appropriate scheduling on threads. As a result, the performance of PSim-x is highly enhanced compared to that of PSim. Specifically, PSim-x outperforms previous synchronization techniques (other than Osci). It also improves upon Osci by being wait-free (assuming that failures occur at the core level rather than at the thread level). Its performance, albeit lower than that of Osci, is still close to the ideal. The same holds for PSimQueue-x, a concurrent queue implementation that is based on PSim-x.

It is noticeable that based on PSim-x, it is straightforward to implement, in a wait-free manner and at a very low cost, useful complex synchronization *primitives* such as `CAS` on multiple words (and many others), that are not provided by current architectures.

2 Related Work

The current state-of-the-art combining algorithm is CC-Synch [9]. CC-Synch employs a single list to (1) store the synchronization requests and (2) implements the lock as a fast queue-based CLH-like spin-lock [7, 19]. This made CC-Synch simpler than previous protocols [14, 26]. Moreover, CC-Synch causes a bounded number of cache misses per request and its *combining degree* (i.e. the average number of requests served by a combiner) is argued [9] to be higher than that of previous techniques. Osci shares some ideas with CC-Synch. However, Osci applies a different technique from that of CC-Synch for announcing requests, batching them in fat nodes before placing them in the list of the pending requests, and it features an additional synchronization layer for the coordination of user-level threads in a way that the algorithm works even with preemptive schedulers.

A hierarchical version of CC-Synch, called H-Synch [9], exploits the hierarchical communication nature of some systems which organize their cores into clusters and provide fast

communication to the threads running on the same cluster, and much slower communication among threads running on cores of different clusters. Our experiments that have been conducted on (a) an AMD machine equipped with 4 AMD Opteron 6272 processors, and on (b) an Intel machine equipped with 4 Intel Xeon E5-4610 processors, show that H-Synch does not perform much better than CC-Synch in this kind of machines (similar results are also presented in [9]). However, batching the requests of the threads running on the same core can be considered as a form of performing hierarchical combining with the cores playing the role of clusters. We have experimented with a variant of H-Synch which employs many user-level threads per core and its performance is much worse than Osci (almost 2.5x slower). The reason is that Osci applies combining in two levels, among the threads of a core and across cores, whereas H-Synch uses a lock to synchronize threads across cores.

Although the employed thread model has been studied in other contexts [5, 30], to the best of our knowledge, this is the first paper that studies the performance impact of fast context switching in the context of combining-based synchronization. Fast context switching is realized here by employing user-level threads. However, our algorithms can be utilized efficiently in several other contexts, like, the Glasgow Haskell compiler [20], applications based on several JAVA implementations [1, 23], OpenMP tasks [12], operating systems that support scheduler activations [3], applications using User-Mode scheduling provided by the recent versions of the Windows operating system [2], and tasks in Cilk-like environments [10]. The use of `Yield` does not play any role in the correctness of Osci and PSim-x. So, they could work efficiently in any environment that exhibits fast context switching, preemptive or not (even if the thread scheduling decisions were made by the operating system [3]).

Blocking implementations of the combining technique are presented in [9, 14, 26]. All are outperformed by CC-Synch [9] and PSim [8]. Sagonas *et. al* [18] have designed a combining technique, optimized for concurrent objects that support operations, which do not require to return some value (e.g. the ENQUEUE operation of a concurrent queue). Their experiments show that their implementation outperforms CC-Synch and flat-combining in that case.

Holt *et. al* [16] present a generic framework based on flat-combining [14] suitable for systems that communicate through message passing (clusters). With this framework, they efficiently implement concurrent data structures much faster than their locking analogs.

Fast concurrent stack and queue implementations appear in [8, 9, 14, 25, 22, 27, 28, 29]. In [25], Morrison and Afek present a lock-free implementation of a concurrent queue, called LCRQ. LCRQ outperforms CC-Queue [9] and the queue based on flat-combining [14]. Our experiments show that OsciQueue outperforms LCRQ by a factor of more than 4 and PSimQueue-x outperforms LCRQ by a factor of more than 2. In [28], Tsigas and Zhang present a lock-free queue implementation which outperforms the lock-free queue of [22], as well as a blocking queue based on spin-locks; the queue is implemented as a circular array.

3 Model

We consider a system of m processing cores on which n threads $p_0, \ldots p_{n-1}$ (where n can be much larger than m) are executed. On each of the m cores, $t = n/m$ threads are executed (for simplicity, let $n \bmod m = 0$); one of these t threads is a kernel thread which spawns the other $t-1$ threads as user-level threads (by calling a function called `CreateThread`). We assume that thread p_i is executed on core i/t. Without loss of generality, we assume that thread $p_{j \cdot t}$ is the kernel thread of core j. We remark that the operating system may decide to move a kernel thread p (and all the user level threads that p has spawned) to another core. Notice also that it is only for the simplicity of the presentation that we make the above assumptions regarding the placement of the threads.

The kernel is aware only of the kernel threads and makes decisions about scheduling. If the kernel decides to switch context when one of these threads p has the CPU, p and all threads it has spawned, stop executing until the operating system decides to allocate the CPU to p again. A thread calls `Yield` whenever it wants to give the CPU. Then, a user-level scheduler, implemented by the corresponding kernel thread, is activated to decide which of the t threads of the core will next occupy the CPU. We assume that this choice is made in a *fair manner*.

We consider a *failure model*, where if a kernel thread p fails, then all threads executing in the same core also fail at the same point in time as p.

4 Description of Algorithms

Description of Osci. Osci (Algorithm 1) maintains a linked list of nodes (initially empty) for storing active requests. A shared pointer $Tail$, initially `NULL`, points to the last inserted node in the list. Each node v is of type ListNode and contains the requests announced by the active threads running on a single core c. These requests are recorded in the $reqs$ field of v, which is an array with as many elements as the number of threads running on c. A thread p_i, running on c, records its requests in the i mod t position of $reqs$ (we note that for cases where different number of threads run on each core, Osci still works).

One of the threads (let it be p) that have recorded requests in the head node of the list plays the role of the combiner. A combiner traverses the list and serves the requests recorded in each of the list nodes (lines 30-36), until either it has traversed all elements of the list or it has served h requests in total* (line 37). If any of these conditions holds, the combiner thread gives up its combining role by identifying one of the threads from those that have recorded their requests in the next node to be the new combiner (lines 43-44). We remark that at each point in time, if the list is non-empty, then there is exactly one combiner. On the other hand, if the list is empty, then no combiner exists.

Each thread owns (only) two nodes of type ListNode and uses them interchangeably to perform subsequent requests (lines 1 and 4), so the nodes are recycled in an efficient way. The first thread (let it be p_i) among those running on core $c = \lfloor i/t \rfloor$, that wants to apply a request, tries to store a pointer to one of its nodes (let this node be nd) in $Announce[c]$ (line 2) using `CAS`†. Notice that $Announce[c]$ may also be accessed simultaneously by other threads running on c. If p_i's `CAS` succeeds, p_i records a struct of type ThreadReq in $nd.reqs[i \bmod t]$. This struct contains a pointer req pointing to the request that p_i has initiated, the return value ret for this request and two boolean variables $completed$ and $locked$ (to which we refer as the $locked$ and $completed$ fields of p_i). If both $locked$ and $completed$ are equal to `false`, p_i becomes the new combiner. Whenever p_i performs spinning, it spins on its $locked$ field; if $locked$ becomes `false` while $completed$ is `true`, then a combiner has served p_i's request.

After p_i has recorded its request, it changes the $door$ field of nd from `LOCKED` (which was initially) to `OPEN` (line 9) and calls `Yield` (line 10) to allow other threads running on c to announce their requests in nd. We say that p_i is a *director* of core j. A director executes lines 4-18 and it is the only thread among those that run on the same core c that can later be a combiner (no other thread on c can be a combiner while applying this current batch of

* In order to prevent a combiner from traversing a continuously growing list, an upper bound h on the requests that p may serve is set; experiments show that h does not significantly affect performance; for our experiments, we have chosen $h = 2n$.

† `CAS` implementations on real machines usually return `true`/`false`. For simplicity of the presentation, we assume that `CAS` returns the old value of the memory location on which it is applied. We can trivially make the algorithm work with `CAS` instructions that return `true`/`false`.

Algorithm 1 Pseudocode for Osci.

```
constant t = ⌈n/m⌉, LOCKED= 0, OPEN= 1, CLOSE= 2;     // m is the number of cores, n is the number of threads
struct ThreadReq {                                    struct ListNode {
    ArgVal req;                                           ThreadReq reqs[0..t-1];
    RetVal ret;                                           int door;
    boolean completed, locked;    }                       struct ListNode *next;    }
shared ListNode *Tail = NULL, *Announce[0..m-1] = {NULL};
private ListNode node_i[0..1] = {⟨ {⟨ ⊥, ⊥, false, true⟩}, LOCKED, NULL⟩};
private boolean toggle_i = 0;

RetVal Osci(ArgVal arg) {                             // pseudocode for thread p_i, i ∈ {1, ..., n}
    ListNode *myNode, *tmpNode, *cur = NULL;
    int offset = i mod t, counter = 0, j;
1   myNode = &node_i[toggle_i];                       // choose one of the nodes of p_i to use
2   cur = CAS(Announce[⌊i/t⌋], NULL, myNode);
3   if (cur == NULL) {                                // p_i is the current director on core ⌊i/t⌋
4       toggle_i = 1 - toggle_i;
5       myNode→reqs[offset].req = arg;                // the director announces its request
6       myNode→reqs[offset].locked = true;
7       myNode→reqs[offset].completed = false;
8       myNode→next = NULL;
9       myNode→door = OPEN;
10      YIELD();                                      // pass control to some other thread
11      Announce[⌊i/t⌋] = NULL;
12      while (CAS(myNode→door, OPEN, CLOSE) != OPEN) YIELD();// close the door
13      tmpNode = SWAP(Tail, myNode);                 // append the node in the request list
14      if (tmpNode ≠ NULL) {                         // p_i is not the combiner
15          tmpNode→next = myNode;                    // set the next field of the previous node
16          while (myNode→reqs[offset].locked==true) YIELD();// p_i calls Yield() instead of spinning
17          if (myNode→reqs[offset].completed==true)  // check if p_i's request has already been applied
18              return myNode→reqs[offset].ret;
        }
19  } else {                                          // p_i is not the director
20      while (CAS(cur→door, OPEN, LOCKED) != OPEN)   // try to acquire the lock
21          if (cur→door == CLOSE) goto line 2;       // if door is closed start from scratch
22          else YIELD();
23      myNode = cur;
24      myNode→reqs[offset] = <arg, ⊥, false, true>; // p_i announces its request
25      myNode→door = OPEN;
26      while (myNode→reqs[offset].locked==true) YIELD();// yield to other threads of same core
27      if (myNode→reqs[offset].completed==true)      // check if p_i's request has already been applied
28          return myNode→reqs[offset].ret;
    }
29  tmpNode = myNode;                                 // p_i is the combiner
30  while(true) {
31      for (j = 0; j < t; j++) {                     // p_i applies the requests of threads executing on core j
32          if (tmpNode→reqs[j].completed == false) {
33              apply tmpNode→reqs[j].req and store the return value to tmpNode→reqs[j].ret;
34              tmpNode→reqs[j].completed = true;     // announce that tmpNode→ req[j] is applied
35              tmpNode→reqs[j].locked = false;       // unlock the corresponding spinning thread
36              counter = counter + 1;
            }
        }
37      if (tmpNode→next == NULL or counter ≥ h) break;// h is an upper bound of the combined requests
38      tmpNode = tmpNode→next;
    }
39  if (tmpNode→next == NULL) {                       // check if p_i's req is the only record in the list
40      if(CAS(Tail, tmpNode, NULL) == tmpNode)       // attempt to set Tail to NULL
41          return myNode→reqs[offset].ret;
42      while (tmpNode→next == NULL) YIELD();         // some thread has in the mean time appended a node
    }
43  for (j = 0; tmpNode→next→reqs[j].completed≠false; j++) noop;
44  tmpNode→next→reqs[j].locked = false;              // identify the new combiner
45  return myNode→reqs[offset].ret;
} // Osci
```

operations). In case that the director becomes a combiner, it also executes lines 29-45. To apply a request, a thread $p_j \neq p_i$ that is also executed on c, first checks whether the *door* of the node nd pointed to by Announce[c] is `OPEN`, and if this is so, it tries to *acquire the door* by setting the value of *door* to `LOCKED` using `CAS` (line 20). If the `CAS` succeeds, p_j records its request in nd (lines 23, 24), unlocks the *door* (line 25), and repeatedly calls `Yield` (line 26) until some combiner either serves its request (lines 32-36) or informs p_j to become the new combiner (line 44).

Since we assume fair scheduling of threads on each core, it is guaranteed that at some later point, p_i is re-activated and executes from line 11 on. Then, p_i changes the *door* field of nd to `CLOSE` (line 12) using `CAS` to avoid synchronization problems with other threads running on c that may simultaneously try to record their requests in nd. Next, p_i appends nd in the shared list by executing `Swap` (line 13). If p_i has appended its node in an empty list (i.e. if the condition of line 14 is `false`), or both the *locked* and *completed* fields in the entry of $nd \rightarrow reqs$ corresponding to p_i, are equal to `false` (lines 16, 17), p_i becomes a combiner.

If p_i does not become a combiner, it updates the *next* field of the previous node to point to its node (line 15), and repeatedly calls `Yield` (line 16) until a combiner either serves its request or informs p_i that it is the new combiner. In the first case, p_i returns on line 18, whereas in the second it executes the combiner code (lines 29-45).

If a thread p_i evaluates the if condition of line 39 to `true` but unsuccessfully executes the `CAS` on line 40, then some other thread p_j has succeeded in updating $Tail$ to point to a node nd' but it has not yet changed the *next* field of the node to which $Tail$ was previously pointed to point to nd'. Then, p_i has to wait until p_j sets *next* to point to nd' (line 42), to ensure that it will correctly choose the thread to become the next combiner (lines 43-44). Notice that on lines 43-44, p_i indicates the node with the smallest identifier among those that have recorded their requests in nd' as the new combiner. A combiner is the director of a core and owns the first node of the list that contains requests unserved by previous combiners.

Osci is linearizable. If a thread p executes the `CAS` of line 2 successfully (i.e. it becomes a director), and until the next time p_i is scheduled and executes line 12, no other thread on this core can become the director (since these threads will execute the `CAS` of line 2 unsuccessfully). Let α be any execution and consider a thread p_i that executes an instance A of Osci at some configuration C. Let $Tail(C)$ be the value of $Tail$ at C. For each i, $1 \leq i \leq n$, denote by $mynode_i(C)$ the value of variable $mynode$ at C. If A executes the `Swap` of line 13 and gets `NULL` as the response, denote by C_f the configuration resulting from the execution of line 13; in case there is at least one configuration C' in A such that $mynode_i(C') \rightarrow reqs[i \mod t].locked =$ `false` and $mynode_i(C') \rightarrow reqs[i \mod t].completed =$ `false`, denote by C_f the first of these configurations. We say that p_i is a *combiner* at C if C_f exists and C is a configuration that follows C_f in which A is active. We prove that in each configuration C, either $Tail(C)$ equals `NULL` and there is no combiner at C, or $Tail(C)$ points to a node nd and there is exactly one combiner at C. We also prove that only a combiner executes lines 30-45 implying that each request recorded in a list node is applied exactly once, as needed to prove linearizability.

Description of PSim and PSim-x. PSim uses (1) an array *Announce* of n entries, where each if the n threads announces its requests, (2) a bit vector *Toggles* which records the threads that have active requests, and (3) an `LL/SC` object $Tail$, which stores a pointer to the state of the object. Whenever a thread p_i wants to apply a request, it announces its request in $Announce[i]$. After that, it toggles $Toggles[i]$ (by executing a `Fetch&Add` instruction) to indicate that it has an active request. Thread p_i discovers which requests are active using

this vector and $Toggles$; p_i serves the active requests by executing their code on a local copy of the simulated state. Then, p_i executes an SC in an effort to change $Tail$ to point to this copy. These actions may have to be applied by p_i twice to ensure that its request has been served. Together with the simulated state, PSim stores a response value for each thread.

In PSim-x, each thread p calls `Yield` after announcing its request. This increases the combining degree of the algorithm. In most cases, when p is scheduled again, it finds out that its request has been completed, so it does not pay the overhead of executing the rest of the algorithm. These are the reasons for the better performance of PSim-x.

5 Performance Evaluation

We evaluated Osci and PSim-x in two different multiprocessor architectures. The first is a 64-core machine consisting of 4 AMD Opteron 6272 processors. Each of these processors consists of 2 dies and each die contains 4 modules, each of which contains 2 cores (64 logical cores). The second one is an Intel 40-core machine equipped with 4 Intel Xeon E5-4610 processors. Each processor consists of 10 cores, and each core executes two threads concurrently (80 logical cores). We used the gcc 4.8.5 compiler and the Hoard memory allocator [4] for all experiments. The operating system was Linux with kernel version 3.4. We performed a lot of experiments for different numbers of user level threads to achieve the best performance for each algorithm. All algorithms were carefully optimized and for those that use backoff schemes, we performed a lot of experiments to choose the best backoff parameters. To prohibit the linux scheduler from doing unnecessary kernel thread migrations, threads were bound in all experiments: the i-th thread was bound on core $\lfloor i/t \rfloor$, where $t = \lceil n/m \rceil$. Most of the commercially available shared memory machines provide `CAS` instructions rather than supporting `LL/SC`. For our experiments, we simulate an `LL` on some object O with a simple `read`, and an `SC` with a `CAS` on a timestamped version of O to avoid the ABA problem[‡].

For our experiments, we developed a library that provides basic support for user level threads. We note that our goal is not to present a new user-level threads library but to ensure that we use a library which is simple and provides fast context switching. By replacing our user-level threads library with any other library that ensures fast context switching, Osci and PSim-x would exhibit similar performance gains. The thread on library we used supports the operations `CreateThread`, `Join`, and `Yield`. A POSIX thread (*kernel-level* thread) runs on each core. This thread calls `CreateThread` to spawn the other *user-level* threads running on the same core. It calls `Join` to wait its children threads to complete. `Yield` activates a user-level scheduler which passes control to some other thread executing on the same core. `Yield` is implemented with a FIFO queue that stores the running threads on each processing core. Moreover, `Yield` makes the appropriate calls to standard *_setjmp/_longjmp* functions to context switch between user-level threads. For performance reasons, it is important that the implementation of `Yield` is as fast as possible and the scheduler is fair.

For our experiments, we first consider a synthetic benchmark, called the `Fetch&Multiply` benchmark, which is similar to that presented in [8, 9]. In this benchmark, a `Fetch&Multiply` object is simulated using state-of-the-art synchronization techniques such as CC-Synch [9], PSim [8], flat-combining [13, 14], a blocking implementation of a `Fetch&Multiply` object based on CLH queue spin-locks [7, 19], OyamaAlg [26], and a simple lock-free implementation. A `Fetch&Multiply` object supports the operation `Fetch&Multiply`(O, k) which returns

[‡] The ABA problem occurs when a thread p reads a value A from a shared variable O and then a thread p' modifies O to the value B and back to A; when p executes again, it thinks that the value of O has never changed which is incorrect.

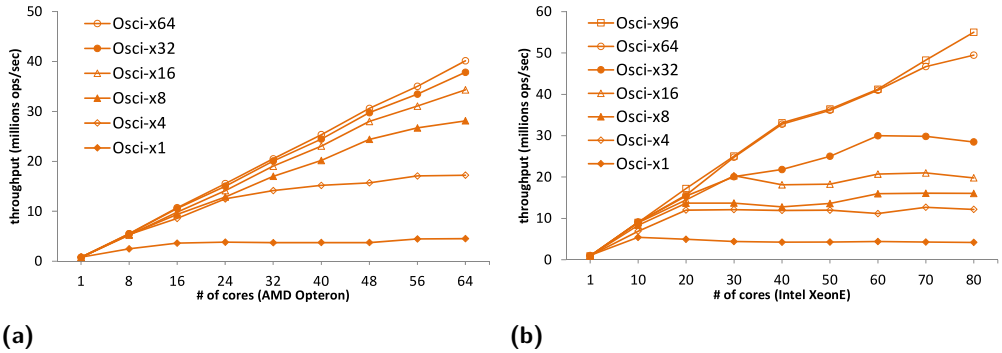
Table 2 Speedup of state-of-the-art synchronization algorithms when employing user level threads (AMD Opteron).

Algorithm	throughput	Variant	throughput	speedup
CC-Synch	4.18	CC-Synch-x8	4.60	1.10
DSM-Synch	4.10	DSM-Synch-x8	4.58	1.12
H-Synch	4.23	H-Synch-x32	10.1	2.39
PSim	3.90	PSim-x64	23.2	5.94
Lock-Free	2.00	Lock-Free-x2	1.87	0.94
CLH	1.58	CLH-x4	1.70	1.08
flat-combining	2.99	flat-combining-x24	5.51	1.84
OyamaAlg	1.72	OyamaAlg-x16	2.80	1.63

the current value v of memory location O and updates the value of O to be $v * k$. The considered lock free implementation uses a single `CAS` object. When a thread wants to apply a `Fetch&Multiply`, it repeatedly executes `CAS` until it succeeds; to increase the scalability, a backoff scheme is employed. The `Fetch&Multiply` object is simple enough to exhibit any overheads that a synchronization technique may induce while simulating a simple, small shared object. To avoid long runs and unrealistically low number of cache misses [22, 8], which may make the experiment unrealistic, we added some small local workload between two consecutive executions of `Fetch&Multiply` in a similar way as in [9, 22]. This local workload is implemented as a loop of dummy iterations whose number is chosen randomly (to be up to 512). For our machine configuration, this workload is large enough to avoid long runs and unrealistically low number of cache misses; still, it is small enough to allow large contention in the simulated object (see Figure 4b for more details). Each instance of the benchmark simulates 10^8 `Fetch&Multiply`, in total, with each of the n threads simulating $10^8/n$ `Fetch&Multiply` out of them. The average throughput is measured. Each experiment is executed 10 times and averages are taken.

In Table 2, we present the throughput for (1) the original versions of the evaluated algorithms (i.e. only one thread per core) and (2) their variants where many user level threads per core are created (for executing the `Fetch&Multiply` benchmark). The x$\langle yy \rangle$ suffix in their names indicates the number of user level threads that are executed per core, so CLH-x4 indicates that CLH spin locks are evaluated with 4 user level threads per core. We performed a lot of experiments for different numbers of user level threads in order to achieve the best performance for each algorithm. We also report the additional speedup we gain for each algorithm by using more than one user level thread per core. The performance of these variants appears in column 4 of the table. The performance presented in Table 2 was measured for the best number of user-level threads for each algorithm and it was performed on the AMD machine. For blocking algorithms that perform spinning on some shared variable, namely CC-Synch, flat-combining and the implementation based on CLH locks, we (repeatedly) call `Yield` instead of spinning. For implementations that repeatedly perform `CAS`, like OyamaAlg and the lock-free implementation, we call `Yield` between any two consecutive attempts to execute `CAS`.

Table 2 shows that all algorithms other than PSim, H-Synch, flat-combining and OyamaAlg do not exhibit any performance gain when employing user level threads. The main reason for this is that the total number of atomic primitives (i.e. `CAS`, `Swap` and `Fetch&Add`) that are executed by each of these algorithms is the same independently of how many user level threads are employed. We note that the performance of the simple lock-free implementation deteriorates for $n > m$, since then this variant (1) behaves similarly to the original algorithm



(a) (b)

Figure 2 Average throughput of Osci while simulating a `Fetch&Multiply` object on (a) the AMD machine (b) the Intel machine, for different numbers of user-level threads per core. The legends are listed top to bottom in the order of the curve they refer to.

(the identity of the thread that repeatedly attempts to execute `CAS` at each point in time is immaterial), and (2) has the additional cost of executing `Yield`.

It is noticeable that the performance of PSim improves by a factor of up to 5.9 when using user-level threads. This performance gain is due to the fact that each thread in PSim-x first announces its request, then calls `Yield` to allow to other threads on the same core to announce their requests, and finally checks if its request has been applied. Only if this is not so, PSim-x tries to serve all the pending requests. However, it turns out that in most of the cases, this is not needed, so the request is completed without paying the overhead of executing the combining part. This results in a big performance advantage of PSim-x. The version of flat combining that uses user level threads increases its performance by a factor of up to 1.84, while the performance of OyamaAlg increases by a factor of up to 1.63. However, PSim-x is 4.21 times faster than flat-combing, 8.2 times faster than OyamaAlg, and 3.8 times faster than H-Synch. We remark that the maximum performance for flat-combining is achieved for 24 user-level threads per core. Notice that H-Synch performs better than that of CC-Synch and PSim but much lower than that of Osci and PSim-x.

Figures 2a and 2b show Osci's performance for different numbers of user level threads per core when executing the `Fetch&Multiply` benchmark on the AMD and the Intel architectures, respectively. The x-axis of the diagram represent the number of cores, and the y-axis represents the average throughput of Osci (over the 10 runs performed). Each line of the diagrams corresponds to a different number of user-level threads per core. The local work between two consecutive `Fetch&Multiply` is up to 512 dummy iterations. The first figure shows that the performance of Osci increases as the number of user level threads increases. For small numbers of user level threads per core (up to 8), the performance gain is significant. Specifically, by using 4 user level threads per core, the performance of Osci is increased almost 4x, and when 8 user level threads per core are used, the performance increases by a factor of more than 6. However, smaller performance gains are illustrated in case of 16 or more user level threads since then the dominant performance factor becomes the switching overhead that `Yield` induces. Moreover, no significant improvement on Osci's performance is achieved when more than 32 threads are employed per core. Our experiments show that for big numbers of threads per core (e.g. more than 64), the performance of Osci slightly deteriorates. In the case of the Intel architecture (Figure 2b), the performance behavior of Osci is very similar, since its performance increases as the number of user level threads increases. The best performance for Osci is achieved for either 64 or 96 threads per core.

Figures 3a, 3b compare the performance of Osci and PSim-x with the other synchronization

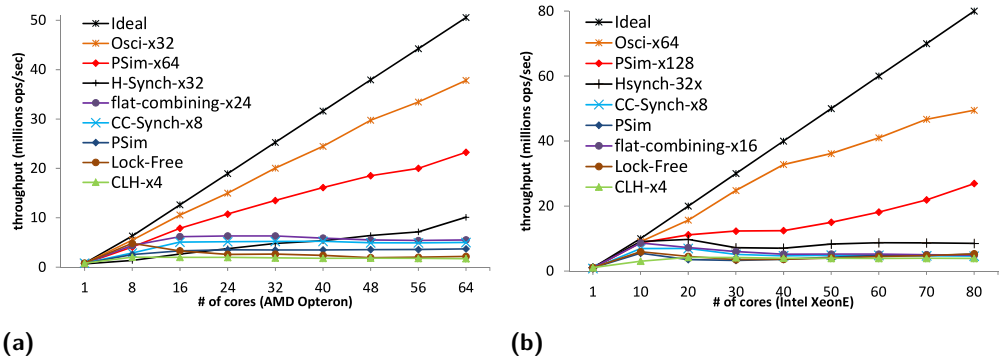


Figure 3 Average throughput of Osci and PSim-x against other synchronization techniques on (a) the AMD machine, and (b) the Intel machine.

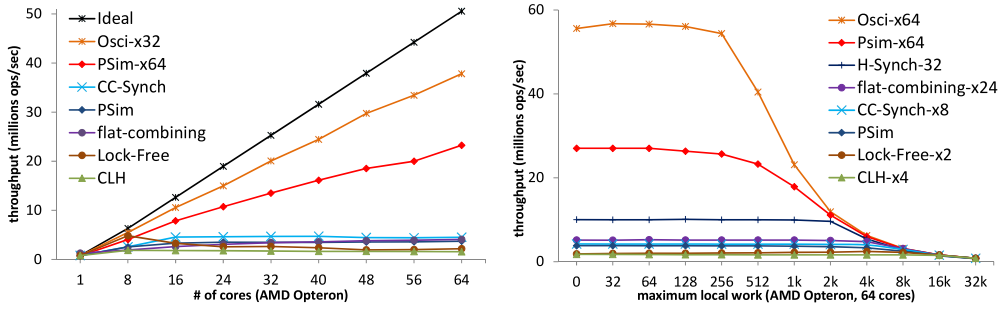
techniques (the versions that use more than one thread per core) running the `Fetch&Multiply` benchmark on the AMD and Intel machines, respectively. Driven by the results presented on Figures 2a, 2b, we configured Osci to use 32 user-level threads per core on the AMD machine and 64 on the Intel machine. Experiments showed that PSim-x scales well with 64 threads per core on the AMD machine and with 128 threads per core on Intel. Figures 3a, 3b also show the performance of an ideal implementation of the `Fetch&Multiply`. The throughput of the ideal implementation is calculated by multiplying n with the throughput of the sequential execution. This is higher than what the ideal algorithm would achieve (so this comparison is not in favor of our algorithms). For $n = 1$, the ideal performance is almost the same as the performance of Osci and CC-Synch [9] (and close to all other algorithms), since (1) the dominant performance factor is the local work, (2) there is no contention, and (3) all algorithms perform just 1 or 2 atomic primitives per request without causing cache misses. Notice that the performance of the ideal algorithm scales linearly to the number of cores.

Figure 3a shows that Osci and PSim-x outperform all other synchronization techniques (the versions that do not use more than one thread per core) by far. On the AMD machine, Osci outperforms CC-Synch [9] by a factor up to 7.5 and PSim-x outperforms CC-Synch by a factor up to 4.6. Recall that, Table 2 shows that the performance of CC-Synch, as well as of all other algorithms, do not improve much when using more than one user-level thread per core. This proves that the batching technique impacts performance significantly. Figure 3a also shows that Osci is up to 6.8 times faster than the variant of flat-combining that employs user-level threads and it is only up to 25% slower than the ideal. As expected [9], CC-Synch is faster than PSim and that the simple lock-free version of `Fetch&Multiply` performs similarly to CLH locks (on this benchmark). Figure 3b shows that the performance advantages of Osci and PSim-x on the Intel machine are very similar to those on the AMD machine. Due to lack of space, we focus our performance study of Osci and PSim-x on the AMD machine, from now on. The performance behavior of Osci and PSim-x on the Intel architecture is similar.

Figure 4a provides a comparison of the performance of Osci and PSim-x to that of the original algorithms (that employ just one thread per core). Similarly to Figure 3a, the performance advantages of Osci and PSim-x are significant. Table 3 shows that Osci does not significantly impact the average latency per operation. With 4 threads per core, the average latency is slightly increased (by less than 5% whereas its throughput is 3.81 times higher than that of CC-Synch. With 8 threads per core, the latency increases just 4 nsec (less than 29%) whereas the throughput is 6.23 times higher. Notice that for local work that is greater than $1k$, the dominant factor in system's performance is the local work and not the overhead of

Table 3 The impact of Osci in latency performance (random local work = 512).

	CC-Synch	Osci-x4	Osci-x8	Osci-x16	Osci-x32	H-Synch-x32
Average Latency (usec)	0.0142	0.0148	0.0182	0.0298	0.0541	0.0205
Throughput (millions ops/sec)	4.51	17.22	28.10	34.32	37.83	10.12



(a) (b)

Figure 4 (a) Average throughput of Osci and PSim-x against other synchronization techniques while simulating a `Fetch&Multiply` object on the AMD machine. (b) Average throughput of Osci and PSim-x with different values of random work for 64 cores.

the synchronization protocol (see Figure 4b). In this case the optimum performance should be achieved with low numbers of user-level threads per core leading to also low latency.

In Figure 4b, we evaluate the performance of Osci and PSim-x for different values of local random work on the AMD machine. In this benchmark, we use 64 user-level threads per core for Osci in order to achieve the best performance. It is shown that Osci and PSim-x outperform all the other synchronization, techniques (Osci outperforms CC-Synch and flat-combining by a factor of up to 11, respectively). Even in cases where the contention is low (local random work is equal to $4k$), Osci and PSim-x perform better (more than 1.5 times faster) than all other synchronization techniques. Smaller values of local work (and therefore higher contention) are in favor of Osci and PSim-x. For relatively large amounts of random local work, the experiment shows that the throughput starts to decline. For local work greater than $16k$, all synchronization techniques have similar performance, since then the local work becomes the dominant performance factor. This experiment shows that Osci and PSim-x would behave efficiently for a large collection of applications, since their performance is tolerant for different amounts of local work that the application may execute. Additionally, even with low contention, Osci and PSim-x perform better than all the other algorithms.

Table 4 sheds light on the reasons that Osci achieves good performance, providing the average number of cache misses (all levels) per request, the average number of cpu cycles spent in backend stalls per request, and the average combining degree achieved by each algorithm. Given that all the memory footprints of our benchmarks were small comparing to the processors' cache size, the great majority of cache misses are cache-line invalidations due to the coherency protocol. The number of cache misses and backend stalls was recorded with *perf* linux tool. As shown in Table 4, Osci exhibits the lowest amount of cache misses among all the other algorithms. Moreover, Osci causes the smallest number of cpu cycles spent in back-end stalls. Therefore, Osci not only executes the smallest amount of cache misses but also its cache-misses are the cheapest, since it spends the fewest cpu cycles in backend-stalls. This is due to the fact that Osci is able to announce an entire batch of active requests by executing just a single `Swap`. Similarly, the combiner reads (and applies) a batch of

Table 4 Last level cache misses and cpu cycles spent in backend stalls per request (64 cores, maximum random work = 64).

Algorithm	cache-misses (all levels)	cpu cycles spent in backend stalls	combining degree
Osci-x64	0.20	247.9	1404
PSim-x64	0.24	2306	1307
H-Synch-x32	0.47	666.1	32
CC-Synch	0.47	4210	1079
PSim	0.40	14300	22

requests without causing a cache miss for each request. We remark that all the other atomic primitives executed by a thread, performing an instance of Osci, access memory locations cached in the local core where the thread resides, avoiding invalidations on the caches of remote cores. PSim-x achieves the second lowest number of cache misses. Even though PSim-x's cache misses are more expensive than H-Synch-x32's (back-end stalls are more in this case), H-Synch-x32 spends a significant amount of cpu cycles spinning on a central lock. Also, the very low combining degree of H-Synch-x32 results in worse performance, comparing to PSim-x; for a bigger number of threads per core, H-Synch's performance deteriorates. PSim-x operates in a more complicated way than Osci in order to achieve wait-freedom. The main overhead of PSim-x compared to Osci is that, to execute a request, each thread locally copies the state of the simulated object and an array of return values of size $\Omega(n)$. Due to the over-subscription, the array of return values is pretty large in PSim-x. This results in a larger amount of cache-misses and backend stalls comparing to Osci. As PSim-x is a simple variant of PSim, the performance gains of PSim-x over PSim originate from the much better combing degree of PSim-x. This results in much less cache misses and cpu stalls.

6 Queue and Stack Implementations

We implement and experimentally analyze a shared queue, called OsciQueue, based on the two lock queue implementation by Michael and Scott [22]. In OsciQueue, the two locks of the lock queue [22] are replaced by two instances of Osci. We also study a version of SimQueue [8] that uses user level threads. This version is called PSimQueue-x.

In Figure 5a, we compare these queue implementations with state-of-the-art queue implementations, like the lock-free LCRQ implementation recently presented by Morrison and Afek in [25], the blocking CC-Queue implementation presented in [9], the wait-free SimQueue [8] implementation, and the lock free queue implementation presented by Michael and Scott in [22]. The experiment is performed on the AMD machine and it is similar to that presented in [22, 8, 9]. Specifically, each of the n threads executes $10^8/n$ *pairs* of ENQUEUE and DEQUEUE requests, starting from an empty data structure. This experiment is performed for different values of n. Similarly to the experiment of Figure 3a, a random local work (up to 512 dummy loop iterations) is simulated between the execution of two consecutive requests by the same thread. In the experiment of Figure 5a, the queue was initially empty. In our environment, OsciQueue achieves its best performance for 64 user-level threads per core. We use the LCRQ implementation that is provided in [24] and we performed a lot of experiments to determine the appropriate ring size of LCRQ that achieves the best performance.

Figure 5a shows that OsciQueue outperforms all other queue implementations by far. Specifically, OsciQueue is more than 4 times faster than LCRQ and more than 5 times faster

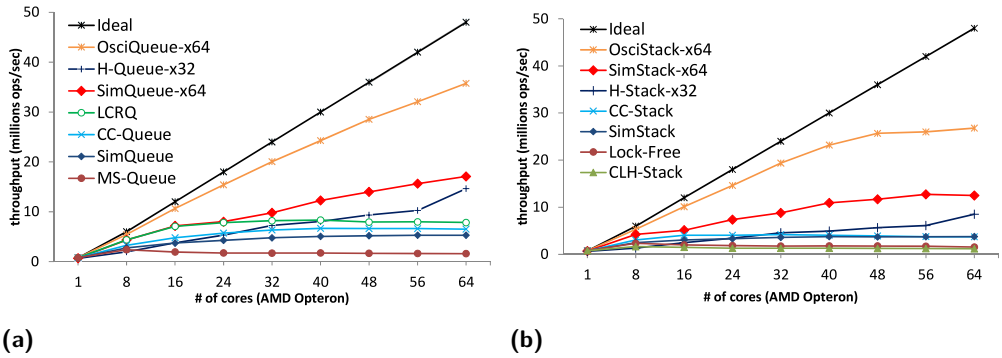


Figure 5 Average throughput of (a) OsciQueue, and (c) OsciStack.

than CC-Queue. It is also shown that PSimQueue-x outperforms LCRQ by a factor of 2. Notice that OsciQueue's performance is far from ideal by a factor of only 25%. Additionally, it ensures wait-freedom which is stronger than lock-freedom ensured by LCRQ. As it is expected [25], LCRQ has the best performance among all the other queue implementations. Recall that the queue is initially empty in this experiment. We also performed a similar experiment where the queue was initially containing 8192 elements. In this case, the performance results were very similar to those of Figure 5a.

Based on Osci and PSim-x, we derive implementations for concurrent stacks, called OsciStack and PSimStack-x, respectively. In Figure 5b, we compare their performance with the state-of-the-art shared stack implementations. More specifically, OsciStack and PSimStack-x were evaluated against CC-Stack [9], SimStack [8], the lock-free stack implementation presented by Treiber in [27], a stack implementation based on CLH spin locks [7, 19], where elimination has been applied when possible. The stack implementation recently presented in [6] is designed for a client-server model and thus it is not evaluated in this paper.

Similarly to the experiment of Figure 5a, we measure the average throughput that each algorithm achieves (every thread executes $10^8/n$ pairs of PUSH and POP requests) for different values of n. The random local work is set to 512. Figure 5b illustrates that OsciStack outperforms by far all other stack implementations. Specifically, OsciStack is up to 7.1 times faster than CC-Stack. It is noticeable that PSimStack-x, which is a wait-free stack implementation outperforms CC-Stack by a factor of up to 3.3.

7 Conclusions

In this paper a new combining technique, which is called Osci is presented. Osci shows remarkable performance when paired with cheap context switching. We have experimentally shown that Osci significantly outperforms all previous combining algorithms. Specifically, the throughput of Osci is higher than that of previously presented combining techniques by more than an order of magnitude. Notably, Osci's throughput is much closer to the ideal than all previous algorithms, while maintaining the average latency in serving each request low. Osci is evaluated in two different multiprocessor architectures, namely AMD and Intel.

Based on Osci, we have implemented and experimentally evaluated implementations of concurrent queues and stacks. These implementations outperform by far all current state-of-the-art concurrent queue and stack implementations. Although the current version of Osci has been evaluated in an environment supporting user-level threads, it would run correctly on any threading library, preemptive or not (including kernel threads).

References

1. Java threads in the solaris environment - earlier releases. URL: `http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/`.
2. User-mode scheduling in windows operating synstem. URL: `http://msdn.microsoft.com/en-us/library/windows/desktop/dd627187`.
3. Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.
4. Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.
5. Neil CC Brown. C++ csp2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007: WoTUG-30*, pages 183–205, 2007.
6. Irina Calciu, Justin E Gottschlich, and Maurice Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *Usenix Workshop on Hot Topics in Parallelism (HotPar)*, 2013.
7. T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, 1993.
8. Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, 2011.
9. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266. ACM, 2012.
10. Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.
11. J. R. Goodman, M.K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.
12. Panagiotis E Hadjidoukas and Vassilios V Dimakopoulos. Nested parallelism in the ompi openmp/c compiler. In *Euro-Par 2007 Parallel Processing*, pages 662–671. Springer, 2007.
13. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. The source code for flat-combining. URL: `http://github.com/mit-carbon/Flat-Combining`.
14. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 355–364, 2010.
15. Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
16. Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*, 2013.
17. Randy H Katz, Susan J Eggers, David A Wood, CL Perkins, and Robert G Sheldon. *Implementing a cache consistency protocol*, volume 13. ACM, 1985.
18. David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Queue delegation locking. 2014.

19 Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, 1994.

20 Simon Marlow, S Peyton Jones, et al. The glasgow haskell compiler, 2004.

21 John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

22 Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

23 Seung-Hyun Min, Kwang-Ho Chun, Young-Rok Yang, and Myoung-Jun Kim. A soft real-time guaranteed java m: N thread mapping method. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1075–1080. Springer, 2005.

24 Adam Morrison and Yehuda Afek. The source code for LCRQ. . URL: `http://mcg.cs.tau.ac.il/projects/lcrq`.

25 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 103–112. ACM, 2013.

26 Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 182–204, 1999.

27 R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

28 Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA*, pages 134–143, 2001. `doi:10.1145/378580.378611`.

29 Philippas Tsigas, Yi Zhang, Daniel Cederman, and Tord Dellsen. Wait-free queue algorithms for the real-time java specification. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 373–383. IEEE Computer Society, 2006. `doi:10.1109/RTAS.2006.45`.

30 Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

31 Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *Computers, IEEE Transactions on*, 100(4):388–395, 1987.