A Highly-Efficient Wait-Free Universal Construction

Panagiota Fatourou
Department of Computer Science
University of Crete
& FORTH ICS
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Department of Computer Science
University of Ioannina
nkallima@cs.uoi.gr

TR 01-2011

Abstract

We present a new simple wait-free universal construction, called Sim, that uses just a `Fetch&Add` and an `LL/SC` object and performs a constant number of shared memory accesses. We have implemented Sim in a real shared-memory machine. In theory terms, our practical version of Sim, called P-Sim, has worse complexity than its theoretical analog; in practice though, we experimentally show that P-Sim outperforms several state-of-the-art lock-based and lock-free techniques, and this given that it is *wait-free*, i.e., that it satisfies a stronger progress condition than all the algorithms it outperforms.

We have used P-Sim to get highly-efficient *wait-free* implementations of stacks and queues. Our experiments show that our implementations outperform the currently state-of-the-art shared stack and queue implementations which ensure only weaker progress properties than wait-freedom.

1 Introduction

Designing efficient shared data structures has become ever so urgent due to the proliferation of multicore machines and the strong necessity of exploiting their computational power by developing parallel software; shared data structures, like stacks and queues, are the most widely used inter-thread communication structures, and therefore they are major building blocks of such software. A universal construction is a generic mechanism to implement any shared data structure; it supports an operation, called ApplyOp, that takes as a parameter the sequential implementation of any operation of the simulated object, and simulates its execution in a concurrent environment.

Herlihy [17] introduced the consensus hierarchy which characterizes the power of a shared object to simulate (together with r/w registers) other objects in a wait-free manner; wait-freedom [17] ensures that each process should finish the execution of its operation within a finite number of its own steps independently of the speed of other processes. A shared object with consensus number n can simulate any other object in a system of n processes. The strongest types of objects are `CAS` and `LL/SC` which have infinite consensus number. Although `CAS` (or `LL/SC`) are currently provided by several systems, it is highly desirable to perform as few such operations as possible since their current implementation is much slower than that of simpler types of objects.

A `Fetch&Add` object O is a weaker type of object (with consensus number 2), which supports in addition to `read`, the operation `FA`(R,x) which adds some (positive or negative) value x to O and returns its previous value. `Fetch&Add` has performance advantages [13] compared to other synchronization primitives (like `CAS`, or `LL/SC`); in brief, a `Fetch&Add` requires only one memory access which minimizes serialization delays, it is combinable [12], and excessive contention for `Fetch&Add` objects can be reduced or eliminated by using appropriate software techniques [30]. In some architectures (i.e., the Origin2000) a `Fetch&Add` is implemented in-memory (bypassing the cache and its coherence protocol) which was proved to be much faster under contention than the integrated to the coherence protocol implementation of `LL/SC` [25].

In this paper, we investigate how to use `Fetch&Add` (in addition to `LL/SC`) to design a highly-efficient wait-free universal construction. Jayanti [21] has proved a lower bound of $\Omega(\log n)$ on the shared memory accesses performed by any *oblivious* universal construction (that does now exploit the semantics of the simulated object) using `LL/SC` objects. One of the open problems mentioned in that paper is the following: *"If shared-memory supports all of* `read`*,* `write`*,* `LL/SC`*,* `swap`*,* `CAS`*,* `move`*,* `Fetch&Add`*,* `Fetch&Multiply`*, would the* $\Omega(\log n)$ *lower bound still hold?"* We present a simple oblivious universal construction, called Sim, that performs a constant number of shared memory accesses. It uses a single `Fetch&Add` object in addition to an `LL/SC` object, thus proving that the lower bound in [21] can be beaten if we use just a single `Fetch&Add` object in addition to an `LL/SC` object. To the best of our knowledge, Sim is the first universal construction that performs just a constant number of shared memory accesses; it proves that the common belief that ensuring wait-freedom is possible only with a significant performance cost is in many cases wrong.

Sim exploits the well-known technique [15, 26, 27, 30] of having a thread executing an operation helping other already announced operations. We have implemented and experimentally tested Sim on a real shared-memory machine. Our experiments show that achieving synchronization using Sim outperforms several state-of-the-art synchronization techniques, both lock-based (like local spinning) and lock-free (Figures 2 and 3). We believe that this is very surprising given that Sim is a *wait-free* algorithm whereas all other techniques ensure only weaker progress properties.

Flat-combining introduced by Hendler, Incze, Shavit, and Tzafrir in SPAA'10 [15] also employs the simple idea of having a thread execute sequentially all announced operations; in flat combining,

this is the thread that manages to acquire a global lock protecting the entire data structure. Apparently, the algorithm is blocking and therefore it is not robust (a thread holding the lock could be preempted causing all other threads to wait or it may fail causing the entire system to block). We experimentally prove that Sim exhibits all the performance benefits of flat combining and sometimes outperforms it (Figures 2 and 3) without sacrificing robustness; in fact, it ensures the strongest known progress condition of being wait-free.

We have used Sim to design new highly-efficient *wait-free* implementations of simple shared data structures like queues and stacks. We experimentally prove that our stack implementation, called SimStack, outperforms most well-known previous shared stack algorithms, like the lock-free stack implementation of Treiber [29], the elimination back-off stack [16], a stack implementation based on a CLH spin lock [9, 22], and a linked stack implementation based on flat combining [15]. Similarly, our queue implementation, called SimQueue significantly outperforms the following previous implementations: a lock-based algorithm (using two CLH locks) and the lock-free algorithm presented in [24], as well as the implementation using flat combining provided by Hendler *et. al* [15].

One limitation of Sim is that it cannot efficiently cope with large objects (i.e., objects that need a large amount of storage s to maintain their state) since it copies (the part of) the object's state (that should be updated) locally. To overcome this limitation, we have combined the main techniques of the universal construction presented by Chuong, Ellen and Ramachandran in SPAA'10 [7] with Sim to get a universal construction, called L-Sim, that operates directly in the shared data structure (and not on a local copy of the entire state). The resulted algorithm exhibits all the advantages of the universal construction in [7], and improves upon it by being adaptive; it performs $O(kw)$ shared memory accesses (where w is the maximum number of different memory words accessed by an operation on the sequential data structure, and k is the *interval contention*, i.e., the maximum number of processes that are active during the execution interval of any operation) instead of $O(nw)$ that does the algorithm in [7]. We would like to point out that the algorithm in [7] is transaction friendly. Making a transaction-friendly version of our algorithm is left as future work; however, we believe that this can be easily achieved by applying similar techniques to those in [7]. The experimental analysis of L-Sim is also left as future work.

While designing Sim, we saw that using a single `Fetch&Add` we could get simple implementations of an active set, a collect object and a snapshot object that perform just one cache miss in cache-coherent machines with up to as many threads as the length c of the system's cache line, and $\lceil n/c \rceil$ cache misses in case $c < n$. Using these implementations, one could get improved performance for several previously presented algorithms [2, 5, 20, 28] in case $\lceil n/c \rceil$ is a small constant.

Fatourou and Kallimanis have presented in [10] a family of wait-free, adaptive, universal constructions, called RedBlue. The first algorithm (F-RedBlue) performed $O(min\{k, \log n\})$ shared memory accesses; the second (S-RedBlue) used smaller objects than F-RedBlue and performed $O(k)$ shared memory access. Sim is much simpler than F-RedBlue and S-RedBlue, uses significantly less objects and performs much less shared memory accesses. Two additional adaptive RedBlue universal constructions [10] (LS-RedBlue and BLS-RedBlue) coped with large objects. These algorithms combined some of the techniques described by Anderson and Moir [4] with the techniques of the RedBlue family to get the best of both worlds. Using Sim, we can obtain much simpler versions of these algorithms (which although perform the same number of shared memory accesses as LS-RedBlue and BLS-RedBlue, they employ $\Omega(n)$ less `LL/SC` objects and reduce the number of `LL/SC` instructions performed in any execution by a factor of $\Omega(\log k)$ per operation. A summary of known wait-free universal algorithms is presented in Table 1.

Algorithm	Primitives	Shared Memory Accesses	Space Overhead
Herlihy [17]	consensus objects, r/w regs	$O(n)$	$O(n^3 s)$
GroupUpdate [1]	LL/SC, consensus objects, r/w regs	$O(\min\{n, k \log k\})$	$O(n^2 s \log n)$
IndividualUpdate [1]	$LL/VL/SC$	$O(kw \log w)$	$O(nw + s)$
F-RedBlue [10]	LL/SC	$O(\min\{k, \log n\})$	$O(n^2 + s)$
S-RedBlue [10]	$LL/VL/SC$, r/w regs	$O(k + s)$	$O(n^2 + ns)$
Anderson & Moir [3]	$LL/VL/SC$	$O((n/\min\{k, M/T\})(B+ ML + nw))$	$O(n^2 + n(B + ML))$
LS-RedBlue [10]	$LL/VL/SC$, r/w regs	$O(B + k(w + TL))$	$O(n^2 + n(B + kTL))$
BLS-RedBlue [10]	$LL/VL/SC$, r/w regs	$O((k/\min\{k, M/T\})\ (B + ML + k + \min\{k, M/T\}w))$	$O(n^2 + n(B + ML))$
Chuong, *et. al* [7]	CAS, r/w regs	$O(nw)$	$O(s + n)$
Sim (this paper)	LL/SC or CAS, Fetch&Add	$O(1)$	$O(1)$
L-Sim (this paper)	LL/SC or CAS, Fetch&Add	$O(kw)$	$O(s + n)$

Table 1: Wait-free universal algorithms and their complexities; in [3], B is the number of blocks, each of size L, needed to store the object's state, and each process is allowed to modify at most T blocks and help at most M/T, where $M \geq 2T$ is some integer.

We note that Sim (and L-Sim) has similar applicability limitations to flat combining [15]; efficient implementations of data structures like search trees, where m lookups can be executed in parallel performing just a logarithmic number of shared memory accesses each, are expected to outperform Sim (since Sim applies each operation sequentially like most previous universal constructions [7, 10, 15, 17, 18]). This limitation can possibly be overcome by using multiple instances of Sim (as done in our queue implementation of Section 5); for more complicated data structures this will be part of our future work.

This paper is organized as follows. Section 2 presents a brief model. Sim is presented in Section 3 and the techniques we have applied to get a practical version of it, as well as some experimental results are described in Section 4. The new wait-free stack and queue implementations as well as their experimental analysis are presented in Section 5. Finally, Section 6 presents L-Sim.

2 Model

We consider an *asynchronous* system of n *processes*, $p_1, \ldots, p_n$, each of which may fail by *crashing*. An *active set* implements a set of processes that participate in a computation; it supports the operations GETSET (which returns a set of "participating" processes), JOIN (to request participation to the set), and LEAVE (to request removal from the set). A *collect* object consists of n components $A_1, ..., A_n$, one for each process, each of which stores a value from a set $V = \{0, ..., 2^d - 1\}$; it supports the operations UPDATE(v) (when executing by p_i it stores the value v in A_i) and COLLECT (which returns a vector of n values, one for each component). An implementation of a (high-level) object from base objects provides an algorithm for each process to simulate each operation of the simulated object using the base objects. A *configuration* C is a vector containing the states of the processes and the values of the shared variables at any point in time. At an *initial* configuration, registers contain initial values and processes are at initial states. A process completes the execution of a step, each time it accesses a shared register. An *execution* is a sequence of steps by processes. A process is *active* at some configuration C, if it has executed the invocation of an operation *op* at C but it has not yet executed the response of *op*. The *execution interval* of *op* is the part of the execution that starts with *op*'s invocation and ends with *op*'s response.

Linearizability [19] imposes a total order, called *linearization*, on all operations performed in an execution α. The linearization must respect the partial order imposed by the execution intervals of operations. If α is *linearizable*, each of its operations has the same response as the corresponding operation in the serial execution determined by the linearization; when this hold, we say that the response is *consistent*. An implementation is *linearizable* if all its executions are linearizable. We remark that implementations of active set and collect objects do not have to be linearizable. An implementation of an active set should rather satisfy the following: (1) the returned set by a GETSET GS should contain any process p that has finished the execution of a JOIN J before the invocation of GS and it has not started the execution of a LEAVE in the execution interval between the end of J and the end of GS, and (2) the returned set by GS should not contain any process p that has finished the execution of a LEAVE L before the invocation of GS and it has not invoked a JOIN in the execution interval between the end of L and the end of GS. Similarly, in an implementation of a collect object the returned vector of each COLLECT Col should contain the value written by an UPDATE (executed by a process p) that has finished its execution before the invocation of Col, given that p has not started the execution of a new UPDATE in the execution interval between the end of U and the end of Col. A *snapshot* object is a collect object that satisfies the extra property of being linearizable (we then use the term SCAN instead of COLLECT).

3 A new universal construction

We start with the presentation of a (single-writer) collect object, called SimCollect, which is a major constructing module of Sim. A collect object is comprised of n components, one for each process, each of which is capable of storing a value from a set D. Let d be the number of bits that are needed for the representation of any value in D. The implementation uses a `Fetch&Add` object R of nd bits. R is partitioned into n chunks of d bits each, one for each process. Process p_i owns the i-th chunk of d bits, and stores there the value of the component that has been assigned to it. An UPDATE U with value v by p_i first performs a `FA`() to ensure that v is written into the i-th chunk of R, and then keeps a copy of v into a local variable *prev*; this copy is maintained by p_i to discover the appropriate value that should be added in the i-th chunk of R during its next UPDATE (which will be the new value minus v). Whenever, p_i executes a COLLECT operation, it simply reads the value stored in R and returns for each component the value stored in the corresponding chunk. Apparently, the step complexity of SimCollect is 1.

If the size b of a `Fetch&Add` object is less than nd bits, then we can employ $\lceil nd/b \rceil$ `Fetch&Add` objects, $R_1, \ldots, R_{\lceil nd/b \rceil}$. In this case, the value last written by p_i is represented by the $(id \mod b)$-th chunk of $R_{\lceil id/b \rceil}$. An UPDATE by p_i adds an appropriate value to $R_{\lceil id/b \rceil}$, and COLLECT reads every `Fetch&Add` object once and returns the set of values written in the chunks. This version of the algorithm has step complexity 1 for UPDATE, and $\lceil nd/b \rceil$ for COLLECT. Notice that in this version COLLECT is not linearizable (but recall that linearizability is not necessary for COLLECT). In case $b \geq nd$, COLLECT is linearizable, so then SimCollect can serve as a single-writer snapshot implementation. We remark that similar techniques as in SimCollect can be used to get an implementation of an active set, SimActSet, by a `Fetch&Add` object of n bits (one for each process) with step complexity 1 if $b < n$, or $\lceil n/b \rceil$ if $b > n$.

We continue to present Sim (Algorithm 1). Sim uses an `LL/SC` object S and an instance Col of the collect implementation discussed above. The `LL/SC` object stores the state *st* of the simulated object, a vector *applied* of n bits identifying whether the current operation (if any) of each process

Algorithm 1 Pseudocode for Sim.

```
type Pindex {1, ..., n};
typedef struct State{
   boolean applied[1..n];
   RetVal rvals[1..n];
   state st;
} State;

shared Collect Col;
shared State S = <<F,...,F>, <⊥,...,⊥>, ⊥>;

// Code for process p_i
RetVal ApplyOp(operation op){
1.   update(Col, i, op);
2.   Attempt();
3.   update(Col,i, ⊥);
4.   Attempt();
5.   return S.rvals[i];
 }
```

```
void Attempt(){
     State ls;        Pindex i, j;
     BitVector act;   operation ops[1..n];

6.   for j=1 to 2 do{
7.      ls = LL(S);
8.      ops = collect(Col);
9.      for i=1 to n do { // local loop
10.       if(ops[i] ≠ ⊥ AND ls.applied[i] == false)
11.          apply ops[i] to ls.st and store
               into ls.rvals[i] the return value;
12.       if(ops[i] ≠ ⊥) ls.applied[i] = true;
13.       else ls.applied[i] = false;
        }
14.     SC(S, ls);
     }
  }
```

has been applied to the simulated object, and an array *rvals* of return values, one for each process; notice that the size of S could be reduced to just a single pointer using indirection. (In later sections, we present how we can get a practical version of Sim that outperforms most of the state of the art lock-based and lock-free algorithms.)

Whenever a process p_i wants to apply some operation *op* to the simulated object, it first announces *op* by updating its component in *Col* (line 1). Then, p_i executes a routine, called `Attempt` (line 2), to ensure that its operation has been applied to the object. Next, p_i updates its component with the special value $\perp$ (line 3) to inform the other processes that *op* has been completed. Then, p executes `Attempt` once more to eliminate any evidence of *op* (line 4).

We now discuss the details of `Attempt`. First, p executes an LL to S (line 7), and then a collect to discover other active operations (line 8). Next, p applies all these operations (in addition to its own) to a local copy *ls* of the state of the simulated object, and calculates the return value for each applied operation (lines 9-13). Finally, p tries to update S by executing an SC (line 14). We prove that it is enough for p to execute lines 7-14 twice to guarantee that its operation *op* has been applied to the simulated object (or that the evidence of its last operation has been eliminated).

Theorem 3.1 Sim *is a linearizable, wait-free implementation of a universal object using a* `Fetch&Add` *object of size b equal to nd bits and one* LL/SC *object.* Sim *performs* $O(1)$ *shared memory accesses. In case* $b < nd$, Sim *uses* $\lceil nd/b \rceil$ `Fetch&Add` *objects and one* LL/SC *object; it performs* $O(nd/b)$ *shared memory accesses.*

We finally discuss some implications of our universal construction. Jayanti [21] has proved that any oblivious implementation of a universal object from LL/SC objects has step complexity $\Omega(\log n)$. Sim is oblivious, so the lower bound can be beaten if just one `Fetch&Add` object (or a collect object) is used in addition to an LL/SC object. Thus, our universal construction implies a lower bound of $\Omega(\log n)$ on the step complexity of any implementation of (1) a collect object, (2) a single-writer snapshot object, or (3) a `Fetch&Add` object, from LL/SC objects.

3.1 Correctness proof of Sim

In this section, we prove that Sim is linearizable. We start by introducing some useful notation. Let α be any execution of Sim and assume that some thread p_i, $i \in \{1, ..., n\}$, executes $m_i > 0$

instances[1] of `Attempt` in α. Let π_j^i be the jth instance of `Attempt` executed by p_i in α (see Fig. 1). By the code, it follows that if $j \mod 2 = 1$, π_j^i is an instance of `Attempt` called on line 2; otherwise, it is an instance of `Attempt` called on line 4. Let U_j^i be the last UPDATE executed by p_i before π_j^i and let Q_j^i be the configuration just before the first step of U_j^i; let r_j^i be the value written by U_j^i.

Let req_l^i be the lth request initiated by p_i, where $l \leq \lceil m_i/2 \rceil$; we remark that req_l^i executes π_{2l-1}^i and π_{2l}^i. We say that req_l^i *is applied* when the following hold: (1) COLLECT, executed by some request req' (that might be req_l^i or any other request) returns req_l^i as the value of the ith component, (2) `Attempt` by req' executes line 11 for req, and (3) the execution of the `SC` of line 14 on S by req' succeeds. When these conditions are satisfied, we sometimes also say that req' *applies* req_l^i.

We start with a brief outline of the proof. We first prove that $S.applied[i]$ equals to 1 just after the execution of the first `Attempt` of req_l^i, whereas it is equal to 0 just after the execution of its second `Attempt`. We also prove that $S.applied[i]$ changes from 0 to 1 for the lth time after U_{2l-1}^i has started its execution and before the end of π_{2l-1}^i; similarly, $S.applied[i]$ changes from 1 to 0 for the lth time after U_{2l}^i has started its execution and before the end of π_{2l}^i. Each time $S.applied[i]$ changes from 0 to 1, a request by p_i is appplied, whereas each time it changes back to 0, the evidence of the fact that p_i has an active request is eliminated. This is enough to prove that req_l^i is applied exactly once in a consistent way.

We first present the following observation which is an immediate consequence of the code (lines 6, 7 and 14).

Observation 3.2 *Consider any j, $0 < j \leq m_i$. There are at least two successful* `SC` *instructions in the execution interval of π_j^i.*

We next prove that at the end of the execution of the first `Attempt` of any instance of Sim by thread p_i, $S.applied[i]$ equals to 1, whereas at the end of the second `Attempt`, $S.applied[i]$ equals to 0.

Lemma 3.3 *Consider any j, $0 < j \leq m_i$. It holds that $S.applied[i]$ is equal to $j \mod 2$ at the end of π_j^i.*

Proof: Assume, by the way of contradiction, that $S.applied[i]$ is equal to $1 - (j \mod 2)$ at the end of π_j^i. By Observation 3.2, there are at least two successful `SC` instructions in the execution interval of π_j^i. It follows that the last successful `SC` instruction executed in π_j^i writes $1 - (j \mod 2)$ into $S.applied[i]$. Let $\texttt{SC}_x$ be this `SC` instruction, let $\texttt{LL}_x$ be its matching `LL` instruction, let p_x be the thread that executes $\texttt{LL}_x$ and $\texttt{SC}_x$, and let G_x be the instance of COLLECT executed by p_x between $\texttt{LL}_x$ and $\texttt{SC}_x$. If $j \mod 2 = 0$, then $\texttt{SC}_x$ writes $1 - (j \mod 2) = 1$ to $S.applied[i]$; the code (lines 8 and $12 - 14$) implies that, in this case, G_x returns a value $r \neq \perp$ for the ith component, whereas $r_j^i = \perp$. In the opposite case where $j \mod 2 = 1$, $\texttt{SC}_x$ writes 0 to $S.applied[i]$; the code (lines 8 and $12 - 14$) implies that, in this case, G_x returns a value $r = \perp$ for the ith component, whereas $r_j^i \neq \perp$. Thus in either case, $v \neq r_j^i$.

Since $\texttt{SC}_x$ is executed in the execution interval of π_j^i, G_x (which is executed before $\texttt{SC}_x$) returns before the end of π_j^i. Since no UPDATE occurs on component i after U_j^i and before the end of π_j^i, if the execution of G_x starts after the end of the execution of U_j^i, G_x must return r_j^i for the ith

[1]We remark that m_i may be ∞.

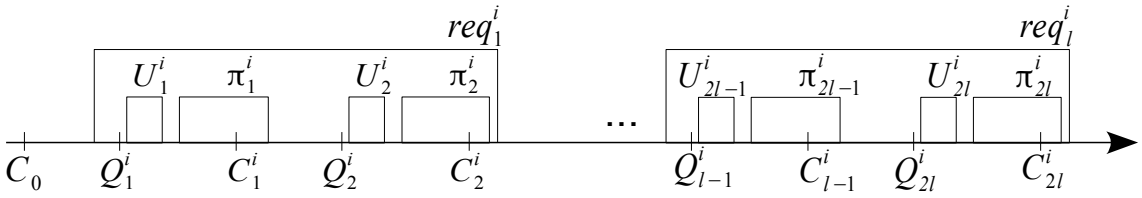

Figure 1: An example execution of Sim algorithm.

component. However, recall that G_x returns $v \neq r_j^i$ for the ith component. Thus, the execution of G_x must start before the end of the execution of U_j^i. Therefore, $\texttt{LL}_x$ which is called by p_x before executing G_x, is performed before the beginning of π_j^i. By Observation 3.2, there are at least two successful SC instructions in the execution interval of π_j^i. Since, $\texttt{SC}_x$ is the last one, it follows that $\texttt{SC}_x$ is an unsuccessful SC instruction, which is a contradiction. ■

For the rest of the proof we use the following notation. Let $C_0^i = C_0$ be the initial configuration. At C_0^i, $S.applied[i]$ is equal to 0. If $m_i > 0$, Lemma 3.3 implies that just after π_1^i, $S.applied[i]$ is equal to 1. Let C_1^i be the first configuration between C_0 and the end of π_1^i at which $S.applied[i]$ is equal to 1. Consider any integer $1 < j \leq m_i$. Lemma 3.3 implies that just after π_{j-1}^i, $S.applied[i]$ is equal to $(j-1) \mod 2$, while just after π_j^i, $S.applied[i]$ is equal to $j \mod 2$. Let C_j^i be the first configuration between the end of π_{j-1}^i and the end of π_j^i such that $S.applied[i]$ is equal to $j \mod 2$. Obviously, C_j^i precedes the end of π_j^i. Fig. 1 illustrates the above notation.

Since the value of $S.applied[i]$ can change only by the execution of an SC instruction on S, it follows that just before C_j^i a successful SC on S is executed. Let $\texttt{SC}_j^i$ be this SC instruction and let $\texttt{LL}_j^i$ be its matching LL instruction. Denote by G_j^i the instance of COLLECT that is executed (line 8) between $\texttt{LL}_j^i$ and $\texttt{SC}_j^i$ by the same thread.

We continue to prove that G_j^i returns the value r_j^i written by U_j^i for thread p_i; moreover, we prove that $\texttt{SC}_j^i$ is executed after Q_j^i (i.e., after the first step of U_j^i has been executed).

Lemma 3.4 *Consider any j, $0 < j \leq m_i$. It holds that: (1) $\texttt{SC}_j^i$ is executed after Q_j^i, and (2) G_j^i returns r_j^i for the ith component.*

Proof: Assume first that $j = 1$. Then, $\texttt{SC}_1^i$ writes 1 to $S.applied[i]$; the code (lines 8 and $12-14$) implies that, in this case, G_1^i returns a value $r \neq \perp$ for the ith component. However, since the initial value of the ith component is $\perp$, and U_1^i is the only UPDATE on the ith component before π_1^i, this can happen only if the execution of G_1^i ends after the beginning of U_1^i, i.e., after Q_1^i. It follows that, G_1^i returns r_1^i, and $\texttt{SC}_1^i$, which occurs after G_1^i, is performed after Q_1^i.

Consider now any $j > 1$. Suppose that G_j^i starts executing before the beginning of π_{j-1}^i. By Observation 3.2, at least two successful SC instructions are executed in the execution interval of π_{j-1}^i. By the code it follows that $\texttt{LL}_j^i$ is executed before G_j^i and, by its definition, $\texttt{SC}_j^i$ is executed after the end of π_{j-1}^i. It follows that $\texttt{SC}_j^i$ is not successful, which is a contradiction. Thus, G_j^i starts its execution after the beginning of π_{j-1}^i.

If $j \mod 2 = 0$, then by definition, $\texttt{SC}_j^i$ writes 0 to $S.applied[i]$. Then, the code (lines 8 and $12-14$) implies that, in this case, G_j^i returns a value $r = \perp$ for the ith component; also, by the

code (lines 1 and 3) and by definition, if $j \mod 2 = 0$, then $r^i_{j-1} \neq \perp$. In the opposite case where $j \mod 2 = 1$, by definition, $\mathtt{SC}^i_j$ writes 1 to $S.applied[i]$. Then, the code (lines 8 and $12-14$) implies that, in this case, G^i_j returns a value $r \neq \perp$ for the ith component; also, by the code (lines 1 and 3) and by definition, $r^i_{j-1} = \perp$ in this case. Thus in either case, $r \neq r^i_{j-1}$.

By the code (lines $1-4$), no UPDATE other than U^i_j is executed on the ith component between U^i_{j-1} and the end of the π^i_j. Since G^i_j starts after the end of U^i_{j-1}, ends before the end of π^i_j, and returns a value not equal to r^i_{j-1}, it follows that G^i_j must return the value r^i_j written by U^i_j. Moreover, the execution intervals of G^i_j and U^i_j should be overlapping. So, the execution of G^i_j ends after the beginning of the execution of U^i_j, and the same is true for $\mathtt{SC}^i_j$ which is executed right after G^i_j. ■

We next prove that no SC on S that is executed between $\mathtt{SC}^i_{j-1}$ and $\mathtt{SC}^i_j$, can change the value of $S.applied[i]$.

Lemma 3.5 *Consider any j, $0 < j \leq m_i$. At each configuration C following C^i_{j-1} and preceding C^i_j, it holds that $S.applied[i] = (j-1) \mod 2$.*

Proof: By definition of C^i_j, no successful SC writes $j \mod 2$ to $S.applied[i]$ between the end of π^i_{j-1} and C^i_j. Assume, by the way of contradiction, that there is some configuration between C^i_{j-1} and the end of π^i_{j-1} such that $S.applied[i]$ is equal to $j \mod 2$. Let C_x be the first of these configurations. Since only SC instructions change the value of S, there is a successful SC instruction, $\mathtt{SC}_x$, which occurs just before C_x and writes $j \mod 2$ to $S.applied[i]$. Let $\mathtt{LL}_x$ be the matching LL instruction to $\mathtt{SC}_x$ and let π_x be the instance of `Attempt` that executes $\mathtt{SC}_x$.

If $j \mod 2 = 0$, then by definition, $\mathtt{SC}_x$ writes 0 to $S.applied[i]$. Then, the code (lines 8 and 14) implies that, in this case, G_x returns a value $r = \perp$ for the ith component; also, by the code (lines 1 and 3) and by definition, it follows that if $j \mod 2 = 0$, then $r^i_{j-1} \neq \perp$. In the opposite case where $j \mod 2 = 1$, $\mathtt{SC}_x$ writes 1 to $S.applied[i]$. Then, the code (lines 8 and 14) implies that, in this case, G_x returns a value $r \neq \perp$ for the ith component; also, by the code (lines 1 and 3) and by definition, $r^i_{j-1} = \perp$ in this case. Thus in either case, $r \neq r^i_{j-1}$.

Since $\mathtt{SC}_x$ is successful, $\mathtt{LL}_x$ must have occurred after C^i_{j-1}. Since G_x occurs between $\mathtt{LL}_x$ and $\mathtt{SC}_x$, and U^i_{j-1} is executed before π^i_{j-1}, G_x occurs after the end of the execution of U^i_{j-1}. Since no other UPDATE occurs on component i between U^i_{j-1} and the end of π^i_j, it follows that G_x returns r^i_{j-1} for the ith component, which contradicts our argument above that G_x returns $r \neq r^i_{j-1}$. ■

Consider any l, $l \leq \lceil m_i/2 \rceil$. By the pseudocode (lines 10-12), when req^i_l is applied, $S.applied[i]$ changes from 0 to 1. This, Lemma 3.5, and the definitions of C^i_{2l-1} and C^i_{2l}, imply the following corollary.

Corollary 3.6 *For each l, $0 < l \leq \lceil m_i/2 \rceil$, req^i_l is applied at most once.*

We next prove that req^i_l is applied exactly once.

Lemma 3.7 *For each l, $0 < l \leq \lceil m_i/2 \rceil$, req^i_l executed by p_i is applied just before C^i_{2l-1}.*

Proof: Recall that a request by p_i is applied each time $S.applied[i]$ changes from 0 to 1. By the definition of C^i_{2l-1}, it follows that some of the requests of p_i is applied before C^i_{2l-1}.

By Lemma 3.4, G^i_{2l-1} returns r^i_{2l-1} for the ith component. By the code, $r^i_{2l-1} = req^i_l$. It follows that req^i_l is applied just before C^i_{2l-1}, as needed. ■

We are now ready to assign linearization points. Let α be any execution. For each $i \in \{1, ..., n\}$ and for each l, $0 < l \leq \lfloor m_i/2 \rfloor$, we place the linearization point of req^i_l at C^i_{2l-1}; ties are broken by the order imposed by thread identifiers.

Lemma 3.8 *Each request req^i_l, $0 < l \leq \lfloor m_i/2 \rfloor$, is linearized within its execution interval.*

Proof: Lemma 3.4 implies that $\texttt{SC}^i_{2l-1}$ follows Q^i_{2l-1}. By its definition, $\texttt{SC}^i_{2l-1}$ occurs before the end of π^i_{2l-1}. Thus, C^i_{2l-1} is in the execution interval of req^i_l, as needed. ■

In order to prove consistency, we use the following notation. Denote by $\texttt{SC}_i$ the i-th successful SC instruction on S and let $\texttt{LL}_i$ be its matching LL. Obviously, between $\texttt{SC}_i$ and $\texttt{SC}_{i+1}$, S (and therefore also its *st* field) is not modified.

Let α be any execution of the algorithm. Denote by α_i, the prefix of α which ends at $\texttt{SC}_i$ and let C_i be the first configuration following $\texttt{SC}_i$. Let α_0 be the empty execution. Denote by l_i the linearization order of the requests in α_i. We remark that $S.st$ stores a copy of the simulated state at each point in time. Moreover, each process applies each request on its local copy of the simulated state sequentially, the one after the other. We say that $S.st$ is *consistent* at C_i if it is the same as the state resulted by executing the requests of α_i sequentially in the order specified by l_i.

Lemma 3.9 *For each $i \geq 0$, (1) $S.st$ is consistent at C_i, and (2) α_i is consistent.*

Proof: We prove the claim by induction on i.
Base case (i=0): The claim holds trivially; we remark that α_i is empty in this case.
Induction hypothesis: Fix any $i > 0$ and assume that the claim holds for $i - 1$.
Induction step: We prove that the claim holds for i. By the induction hypothesis, it holds that: (1) $S.st$ is consistent at C_{i-1}, and (2) α_{i-1} is consistent with linearization l_{i-1}. Let req be the request that executes $\texttt{SC}_i$. If req applies no request on the simulated object, the claim holds by induction hypothesis. Thus, assume that req applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered with respect to the identifiers of the threads that initiate them.

Notice that req performs $\texttt{LL}_i$ after C_{i-1} since otherwise $\texttt{SC}_i$ would not be successful. By definition, $S.st$ does not change between C_{i-1} and C_i. Thus, $\texttt{LL}_i$ returns the value written in $S.st$ at C_{i-1}. By the induction hypothesis, this value is consistent at C_{i-1}. Lemma 3.7 and Corollary 3.6 imply that $\texttt{SC}_i$ is the only SC that applies $req_1, \ldots, req_j$. Thus, none of these requests have been applied in past.

Given that $req_1, ..., req_j$ are executed by req sequentially, the one after the other in the order mentioned above, it is a straightforward induction to prove that (1) for each l, $0 \leq l \leq j$, request req_l returns a consistent response; moreover, $ls.st$ is consistent once line 11 has been executed by req for all these requests. Therefore, $S.st$ is consistent after the execution of req's successful SC. This concludes the proof of the claim. ■

4 From theory to practice

Implementation. We describe the major techniques applied to Sim to port it to a real-world machine architecture, like x86_64. This gives a practical variation of Sim, called P-Sim.

A shared bit vector *Act* of size n is employed, containing one bit for each process; each process toggles its bit, by performing a `Fetch&Add`, when it initiates a new operation. An operation by p_i is applied only if the i-th bit of *Act* differs from the i-th bit of the *applied* array of struct *State*; before attempting to write the new state of the simulated object, p_i changes *S.applied* to be equal to *Act*. In this way, there is no need for eliminating the evidence of an executed operation. This technique reduces the total number of cache misses to almost half giving a noticeable speed gain.

The collect object is replaced by a set of n single-writer r/w registers (*Announce* array). When p_i wants to apply an operation *op*, announces it by writing *op* (and its parameters) in *Announce*[i]. Process p_i discovers the operations that other active processes want to perform (to help them) by reading the appropriate entries of *Announce*. This increases the time complexity of Sim to $O(k)$ (where k is the interval contention) but it decreases the size of the `Fetch&Add` object.

The information stored in struct *State* is now maintained using indirection. Each process p_i maintains a pool of a constant number C of structs of (an enhanced version of) type *State*. These pools are implemented by allocating an array *Pool* of nC structs of type *State*. Process p_i's pool is comprised by the $Pool[(i-1)C..iC-1]$ part of the array. Variable S has now been replaced by a shared variable P which is an index in *Pool*, i.e., P is a "reference" to a struct of type *State* which stores the current state of the simulated object (in addition to other useful information).

Several modern shared memory machines (like those using the x86_64 architecture) support a `Fetch&Add` instruction on up to 64 bit words. In order to cope efficiently with more than 64 threads, the multi-word bit vector *Act* is implemented by storing its words to the minimum possible number of cache lines. Notice that a typical cache line is usually of 64 bytes; thus, it can be used to store one bit for each of up to 512 processes (so, more than one cache line may be needed only if the number of processes is more than 512; otherwise, we read *Act* with just one cache miss).

The majority of the commercially available shared memory machines support `CAS` rather than `LL/SC`. We simulate an `LL` on P with a `read`(P), and an `SC` with a `CAS` on a timestamped version of P to avoid ABA. Since P stores just an index to the pool of blocks (and not a full 64 bit pointer), there are enough bits (in our experiments 48) in a word to store the timestamp (we remark that in systems with more processes, we could use 128 bit words, supported e.g., in x86_64).

We remark that the performance of P-Sim gets enhanced when processes manage to help a large number of other processes while performing their operations. For exploring this property, we use an adaptive exponential backoff scheme which has some similarities to that used in [16]. A process p_i backoffs after it has announced its operation and has indicated in *Act* that it is active. This results in increasing the number of operations that p_i will help while executing the current instance of its operation, as is desirable. It is worth-pointing out that P-Sim achieves very good performance even if no backoff is employed.

A simplified version of P-Sim is presented in Algorithms 2, 3. The full code is provided at http://code.google.com/p/sim-universal-construction/.

Performance Evaluation. We run our experiments on a 32-core machine consisting of four `AMD` opteron 6134 processors (Magny Cours). Each processor consists of two dies and each of them contains four processing cores and an L3 cache shared by its cores. Dies and thus processors are connected to each other with Hyper Transport Links creating a topology with an average diameter

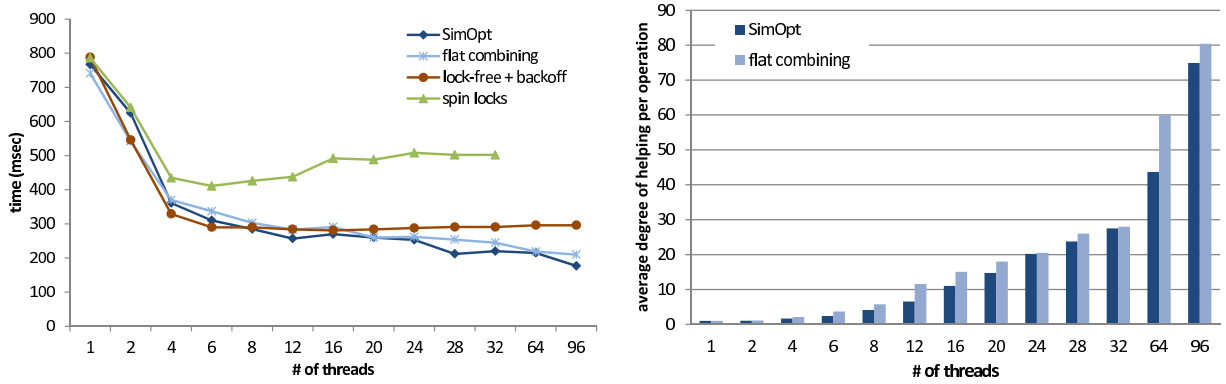

Figure 2: Performance and degree of helping of Sim.

of 1.25 [8]. All codes were compiled with *gcc* 4.3.4, and the Hoard memory allocator [6] was used to eliminate bottlenecks in memory allocation.

We first focus on a synthetic benchmark which shows the performance advantages of Sim over well-known blocking and lock-free techniques. More specifically, we used Sim to implement a simple `Fetch&Multiply` instruction; we measure the time needed to complete the execution of 10^6 `Fetch&Multiply` instructions (with each thread executing $10^6/n$ such instructions) for different values of n. For each value of n, the experiment has been performed 10 times and averages have been taken. A random number (up to 512) of dummy loop iterations have been inserted between the execution of two `Fetch&Multiply` by the same process; in this way, we simulate a random work load large enough to avoid unrealistically low cache miss ratios (but not too big to reduce contention). A similar technique was employed by Michael and Scott in [24] for the same reasons.

We have performed the same experiment using the following mechanisms: CLH spin locks [9, 22][2], a simple lock-free algorithm with exponential back-off using a single `CAS` object, and flat combining [14, 15]. We carefully optimized these algorithms to achieve best performance in our computing environment. For those that use backoff schemes, we performed a large number of experiments to select the best backoff parameters in each case. CLH spin locks have been evaluated for only up to 32 threads (so that each thread runs on a distinct core), since otherwise they result in very poor performance. We used the flat combining implementation provided by its inventors [14, 15] but we carefully chose its parameters (i.e., polling level, number of combining rounds) to optimize its performance in our computing environment; we performed a big number of experiments and we observed that choosing the flat combining's parameters differently to get other degrees of helping leads to performance degradation. The simple lock-free algorithm uses a single `CAS` object O, and executes a `CAS` instruction on O repeatedly, until it successfully stores the new value there; the algorithm employs an exponential back-off scheme to reduce contention in accessing O. Given that this seems to be the simplest lock-free implementation, we expect that it performs well.

In our experiment, Sim has been proved to be up to 2.36 times faster than spin-locks, and up to 1.67 times faster than the lock-free algorithm (Figure 2). Since both Sim and flat-combining are based on the simple idea of having each process that performs an operation helping other active operations, we would be happy to see Sim (which is a wait-free algorithm) to perform the same

[2]As expected for cache-coherent NUMA architectures, we experimentally saw that MCS spin locks [23] have slightly worse (or similar) performance than (to) CLH locks, so we present our results only for CLH locks.

well as flat-combining (which is blocking). As Figure 2 shows, this is indeed the case. Even more, for all values of $n > 4$, we got better numbers for Sim than for flat-combining. We found it very surprising that there exists a *wait-free* universal construction with the performance advantages of Sim.

As illustrated in Figure 2, all algorithms scale well for up to 4 cores. For larger values of n, the performance of spin-locks and the lock free algorithm degrade as the number of threads increases. A major reason for this is that the (intra)communication cost between dies is much higher than the (inter)communication cost between the four cores of the same die; additionally, the lock-free algorithm causes more contention as n increases. It is worth-pointing out that the performance of Sim and flat combining is being enhanced as the number of n increases (even for values of $n > 4$). This is so since the average degree of helping (where Sim and flat-combing owe their good performance) increases with the number of active operations in the system, and therefore also with n, as shown in the second part of Figure 2. We remark that this enhancement in performance is noticed even in case of $n > 32$ where the processing cores are oversubscribed.

5 A stack and a queue implementation based on Sim

A New Wait-Free Implementation of a Shared Stack. Implementing a stack object based on Sim is not a difficult task. The only encountered subtlety is that instead of maintaining the entire state of the stack, Sim is employed to atomically manipulate just the top of the stack.

We compare the experimental performance of SimStack with that of state-of-the-art concurrent stack implementations, like the lock free stack implementation presented by Treiber in [29], the elimination back-off stack [16], a stack implementation based on CLH spin lock [9, 22], and a linked stack implementation based on flat combining [14, 15].

Algorithm 2 Data structures used in P-Sim algorithm.

```
typedef struct State {
    int seq1, seq2;
    BitVector applied;
    state st;
    RetVal rvals[1..n];
} State;
typedef struct TimedPoolIndex {
    int index;  // 16 bit array index
    int tm;     // 48 bit timestamp
} TimedPoolIndex;

// Each element of pool is initialized
// as follows < 0, 0, < 0, ..., 0 >, ⊥, < ⊥, ..., ⊥ >>
shared State Pool[0..n*C];  // C > 1 is a small constant
shared TimedPoolIndex P = {n*C, 0};
shared BitVector Act = 0;
shared OpType Announce[1..n];

// private persistent variables of process p_i
// operator << implements a left bit shift
BitVector mask_i = 1 << i;
BitVector offset_i = -mask_i;
int pool_index_i = 0;
```

Algorithm 3 Pseudocode for P-Sim algorithm.

```
RetVal ApplyOp(function sfunc, ArgVal arg) { // Code for process p_i
   TimedPoolIndex lp, mp;
   ObjectState *lst;
   ArgVal tmp_arg;
   int j, k;
   BitVector Act;

1.  Announce[i] = arg;                          // announce the operation
2.  offset_i = -offset_i;                       // offset_i is added to Act to toggle p_i's bit
3.  FAA(Act, offset_i);                         // toggle p_i's bit in Act, Fetch&Add acts as a full write-barrier
4.  backoff();
5.  for j=0 to 1 do {                           // code of Attempt
6.     lp = P;                                  // read reference to struct State
7.     lst = &Pool[i * n + pool_index_i];
8.     *lst = Pool[lp.index];                   // read struct State in a local variable lst
9.     active = Act;                            // read Act
10.    diffs = lst → applied XOR active;                    // determine the set of active processes
11.    if (lst → seq1 != lst → seq2) continue;              // consistency check
12.    if (diffs AND mask_i == 0) return lst → rvals[p_i];  // if the operation has already been applied return
13.    if (j == 0) compute_backoff();
14.    lst → seq1 = lst → seq1 + 1;
15.    while (diffs != 0) {                     // as long as there are still processes to help
16.       k = bitSearchFirst(diffs);            // find the next such process
17.       tmp_arg = Announce[k];                // discover its operation
18.       lst → rvals[k] = sfunc(lst, tmp_arg);          // apply the operation to a local copy of the object's state
19.       diffs = diffs XOR (1L << k);          // extract this process from the set
       }
20.    lst → applied = active;                  // change applied to be equal to what was read in Act
21.    lst → seq2 = lst → seq2 + 1;
       // compute a new reference mp to store in P
22.    mp.tm = lp.tm + 1;                       // increase P's timestamp
23.    mp.index = i * n + pool_index_i;         // store in mp.index the index in Pool where lst will be stored
24.    Pool[i * n + pool_index_i] = lst;        // store the new state in position mp.index of Pool
25.    if (CAS(P, lp, mp)) {                    // try to change P to the value mp
26.       pool_index_i = (pool_index_i + 1) mod C; //if this happens successfully,use next item in p_i's pool next time
27.       return lst → rvals[i];                // return;
       }
   }
28. lp = P;                                     // after two unsuccessful efforts, read current value of P
29. lst = &Pool[lp.index];                      // read the element of Pool indicated by the index field of P
30. return lst → rvals[i];                      // return the value found in the record stored there
}
```

Our experiment is similar to that performed by Michael and Scott for queues in [24]. More specifically, we measure the time needed to complete the execution of 10^6 pairs of a PUSH and a POP as the number of threads increases (Figure 3). Again, for each value of n, the experiments have been performed 10 times and averages have been taken; we have also simulated a random workload by executing a random number of iterations of a dummy loop after each operation.

As shown in Figure 3, all algorithms scale well up to 4 threads but SimStack outperforms all other implementations for $n > 4$. More specifically, SimStack is up to 2.94 times faster than the lock-free stack, up to 2.58 times faster than the spin-lock based stack, up to 2.57 times faster than the elimination back-off stack, and up to 1.17 times faster than flat-combining.

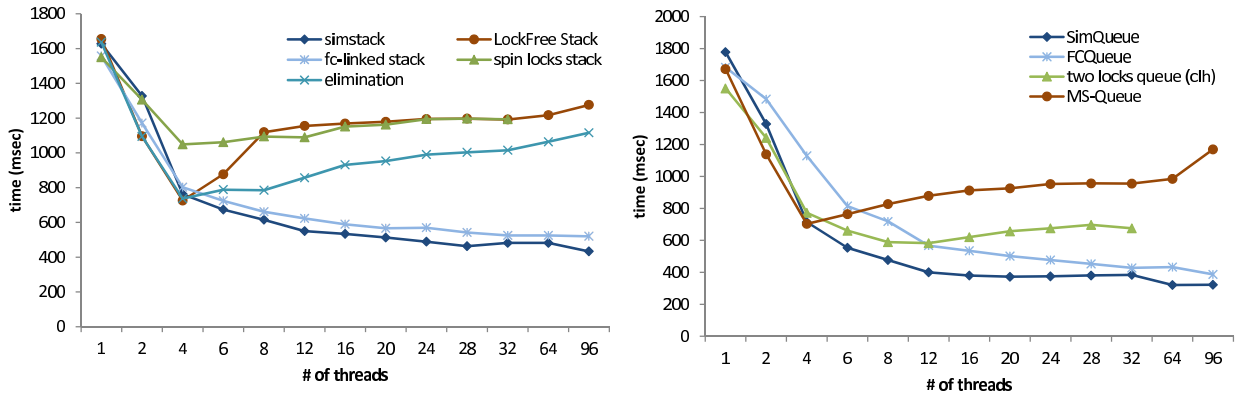

Figure 3: Performance of SimStack (left) and SimQueue (right).

As expected the elimination backoff stack achieves better performance than the lock-free and the spin-locks based implementations in almost all experiments. Again, the performance of the spin-lock based and the lock free implementations, as well as that of the elimination back-off stack degrade as the number of threads increases once n becomes more than four, in contrast to flat-combining and Sim that enjoy further performance enhancement as n increases. SimStack and flat combining significantly outperform the other stack implementations. A possible reason that Sim exhibits better performance than flat-combining could be that executing the algorithm instead of performing local spinning may result in better cache locality.

A New Wait-Free Implementation of a Shared Queue. To allow the enqueuers and dequeuers to run independently, we employed two instances of P-Sim. Whenever a process p performs an ENQUEUE, it helps only other enqueuers (ignoring currently active dequeuers). Process p creates a local list of nodes, one for each enqueuer it helps. These nodes are eventually inserted to the shared queue by changing the next field of the queue's tail node to point to the first node of the list, and the queue's tail to point to the last node of this list. To do so, the tail of the queue, and pointers to the first and last nodes of this list are stored in the *EnqState* struct of the enqueuer's instance of Sim. If process p manages to successfully update *EnqP* (i.e., the pointer pointed to the *EnqState* struct), it also tries to update (using `CAS`) the next field of the last node of the queue to point to the first node of its local list. To avoid situations where p crashes before doing this change but after it has written a new value in *EnqP*, any subsequent ENQUEUE also tries to connect (using `CAS`) the tail of the shared queue with the first node of the list (recorded in the *EnqP*). Similarly, a DEQUEUE helps only active dequeuers. The *DeqState* struct stores a pointer to the front element of the queue. To ensure consistency, each DEQUEUE also executes a `CAS` to connect the two parts of the queue in a similar way that ENQUEUE operations do.

Each process maintains three pools of structs, one containing structs of type *EnqState*, one containing structs of type *DeqState* and one containing nodes of the queue. Each time a process wants to allocate a new struct, it simply uses one of the structs in the appropriate local pool. The pseudocode of the implementation is provided in Algorithms 2, 3.

We compare the experimental performance of SimQueue with that of state-of-the-art concurrent queue implementations, like the lock-based implementation (using two CLH locks) and the lock-free algorithm presented in [24], and the implementation using flat combining [14, 15]. Similarly to the experiment performed in [24], we measure the time needed to complete the execution of 10^6 pairs of an ENQUEUE and a DEQUEUE operation as the number of threads increases (Figure 3). As in

previous experiments, we simulate a random workload after each operation.

As shown in Figure 3, SimQueue significantly outperforms all other implementations for $n > 4$. More specifically, SimQueue is up to 3.06 times faster than the lock-free implementation, up to 1.82 times faster than the spin-lock based implementation, and up to 1.5 times faster than flat combining. As expected, flat combining outperforms all queue implementations other than SimQueue. However, SimQueue achieves much better performance than flat combining for almost any number of threads. This performance advantage of SimQueue over flat-combining is basically due to the fact that we used two instances of Sim for our queue implementation, thus achieving increased parallelism by having enqueuers and dequeuers run concurrently.

6 Sim for large objects

The algorithm (Algorithms 7 and 8) uses an instance of SimActSet and a set of n single-writer registers (instead of a collect object). Each process starts the execution of an operation *op* by announcing *op* in its single-writer register and joining the active set. The main difficulty in designing L-Sim was to ensure that at each point in time, all "up-to-date" processes (i.e., those that have read the current version of *State*) will help the same set of operations. This is achieved by storing in *State* (S) two versions of the *applied* bit vector (called *applied* and *papplied*). Each time an instance A of `Attempt` is executed, *papplied* is updated to store the values found in *applied* at the beginning of A (line 14); *applied* is updated based on the processes that are recorded in the active set (lines $15 - 16$). Whether an operation by a process p_i should be applied or not is determined based on the values read in the i-th entry of the arrays *applied* and *papplied* of S; if they contain `FALSE` and `TRUE`, respectively, then the operation has not been applied yet and it should be simulated (lines $18 - 35$); otherwise, the operation (if any) has already been applied.

The simulated data structure is now shared and it can be updated directly by any process. For each data item x, L-Sim maintains a struct of type $ItemSV$. This struct stores the old and the current value of the data, a toggle bit that identifies the position in the *val* array of the struct where the current data for x should be read from, and a sequence number. All these fields are required to achieve synchronization between the processes that help the same set of operations.

Each process p_i uses a local directory D containing structs of type $DirectoryNode$, where it stores information about each item it accesses during the execution of its current instance of `Attempt` (lines 25, 31 and 32), and performs all its updates first on these copies (lines 29 and 34). Only after it has finished the simulation of the set of operations described in the arrays of S, it applies the changes listed in the elements of its directory to the shared data structure (lines $37-41$).

Some additional synchronization that should be achieved between different helpers of the same set of operations is when new data items are allocated by these operations; Then, all helpers should use the same allocated ItemSV struct for each of these data items. To solve this problem, S stores a pointer (called *var_list*) to a list of newly created data items shared by all processes that read this instance of S. Each time a process p_i needs to allocate the k-th, $k \geq 1$, such data item, it tries to add a struct of type $NewVar$ as the k-th element of the list (lines $21 - 23$). If it does not succeed, some other process has already done so, so p uses this struct (by moving pointer *ltop* to this element on line 24, and by inserting $ltop \rightarrow var$ in its dictionary on line 25).

Theorem 6.1 L-Sim *is a linearizable, wait-free implementation of a universal object. The number of shared memory accesses performed by* L-Sim *is* $O(kW)$.

Algorithm 4 Data structures of SimQueue algorithm.

```
// Two copies of Sim are used
// the first is used to achieve synchronization between the enqueuers,
// and the second to achieve synchronization between the dequeuers

typedef struct Node {                    // node of the queue
   Value v; // value stored in each node
   struct Node *next;
} Node;

typedef struct EnqState {                // struct of type State for the enqueuers' copy of Sim
   int seq1;
   BitVector applied;
   Node *new_tail;
   Node *lfirst;
   Node *old_tail;
   int seq2;
} EnqState;

typedef struct DeqState {                // struct of type State for the dequeuers' copy of Sim
   int seq1;
   BitVector applied;
   Node *head;
   Node rvals[1..n];
   int seq2;
} DeqState;

typedef struct TimedPoolIndex {
   int index;
   int tm;
} TimedPoolIndex;

shared EnqState EnqPool[0..nC];          // where C is a small constant greater than 1
shared DeqState DeqPool[0..nC];
shared TimedPoolIndex EnqP = {nC, 0};
shared TimedPoolIndex DeqP = {nC, 0};
shared BitVector EnqAct = 0;
shared BitVector DeqAct = 0;
shared OpType EnqAnnounce[1..n];
// EnqAnnounce[i] stores the argument of the last executed (or the currently active)
// enqueue operation of process p_i
// there is no need for having a DeqAnnounce array since dequeues do not have any arguments

// private, persistent variables of process p_i
BitVector mask_i = 1 << i;
BitVector enq_offset_i = -mask_i, deq_offset_i = -mask_i;
int enq_pool_index_i = 0, deq_pool_index_i = 0;
```

Algorithm 5 Pseudocode of ENQUEUE operation.

```
void Enqueue(ArgVal arg) {                    //Code for process p_i
   BitVector lactive, diffs;
   TimedPoolIndex lp, mp;
   EnqState ms, ls;
   int k, j;
   Node *node, *lfirst;

1. EnqAnnounce[i] = arg;                      // announce the operation
2. enq_offset_i = -enq_offset_i;              // enq_offset_i is added to EnqAct to toggle p_i's bit
3. Fetch&Add(EnqAct, enq_offset_i);           // toggle the bit in EnqAct
4. backoff();

5. for j=0 to 1 do {                          // code of Attempt
6.     lp = EnqP;                             // read reference to struct State for enqueuers
7.     ms = EnqPool[lp.index];                // read struct State for enqueuers in a local variable
8.     lactive = EnqAct;
9.     diffs = ms.applied XOR lactive;        // determine the set of active enqueuers
10.    if (ms.seq1 != ms.seq2) continue;      // consistency check
11.    if (diffs AND mask_i == 0) return;     // if the operation has already been applied return
12.    ms.seq1 = ms.seq1 + 1;
13.    node = new Node();                     // allocate a new node for the item to be enqueued
14.    node → next = nil;                     // and initiate its fields
15.    node → obj = arg;
16.    lfirst = node;                         // make a new list which will eventually contain one node
                                              //for each of the the enqueuers that p_i will help

17.    diffs = diffs XOR mask_i;              // exclude p_i from the set of active enqueuers
18.    CAS(ms.old_tail → next, nil, ms.lfirst);   // Connect the two parts of the queue
                                                  // if not already done by other enqueuers
19.    while (diffs != 0) {                   // as long as there are still processes to help
20.        k = bitSearchFirst(diffs);         // find the next such enqueuer p_k
21.        node → next = new Node();          // assign a new node for the item that p_k wants to enqueue.
22.        node = node → next;
23.        node → next = nil;
24.        node → obj = EnqAnnounce[k];       // initialize appropriately the fields of this node
25.        diffs = diffs XOR (1 << k);        // exclude p_k from the set of active processes
       }
26.    ms.old_tail = ms.new_tail;             // store tail in ms.old_tail
27.    ms.lfirst = lfirst;                    // store a pointer to the first node of the list of newly created nodes
28.    ms.new_tail = node;                    // keep a pointer to the last node of the list of newly created nodes
                                              // in ms.new_tail; this will be the new tail if the CAS by p_i succeeds
29.    ms.applied = lactive;                  // change applied to be equal to what was read in EnqAct
30.    ms.seq2 = ms.seq2 + 1;
31.    mp.tm = lp.tm + 1;                     // increase timestamp
32.    mp.index = n*i + enq_pool_index_i;     // store in mp.index the index in EnqPool where ms will be stored
33.    EnqPool[mp.index] = ms;                // store the new state in position mp.index of EnqPool
34.    if (CAS(EnqP, lp, mp)) {               // try to change EnqP to point to mp
35.        CAS(ms.old_tail → next, nil, ms.lfirst);  // try to change the last pointer to point to the first node
                                                      // of the local list
36.        enq_pool_index_i = (enq_pool_index_i + 1) mod C;   // use next item in p_i's enqueue pool next time
37.        return;
       }
   }
38. return;
}
```

Algorithm 6 Pseudocode of DEQUEUE operation.

```
Node Dequeue(void) {                                    //Code for process p_i
   BitVector lactive, diffs;
   TimedPoolIndex lp, mp;
   DeqState ms, ls;
   EnqState lenq_s;
   int k,j;

39. deq_offset_i = -deq_offset_i;                       // deq_offset_i is added to DeqAct to toggle p_i's bit
40. Fetch&Add(DeqAct, deq_offset_i);                    // toggle the bit in DeqAct
41. backoff();

42. for j=0 to 1 do {                                   // code of Attempt
43.     lp = DeqP;                                      // read reference to struct State for dequeuers
44.     ms = DeqPool[lp.index];                         // read struct State for dequeuers in a local variable ms
45.     lactive = DeqAct;                               // read DeqAct
46.     diffs = ms.applied XOR lactive;                 // determine the set of active dequeuers
47.     if (ms.seq1 != ms.seq2) continue;               // consistency check
48.     if (diffs AND mask_i == 0) return ms.rvals[p_i];    // if the operation has already been applied return
49.     lenq_s = EnqPool[EnqP.index];                   // read the current state of the enqueuers' copy of Sim to help
                                                        // enqueuers connect the queue which might be split in two parts
50.     if(lenq_s.seq1 == lenq_s.seq2)                  // consistency check
51.         CAS(lenq_s.old_tail → next, nil, lenq_s.lfirst);    // try to connect the two parts of the queue
52.     ms.seq1 = ms.seq1 + 1;
53.     while (diffs != 0) {                            // as long as there are dequeuers to help
54.         k = bitSearchFirst(diffs);                  // find the next such dequeuer p_k
55.         next = ms.head → next;          // calculate the return value for p_k; ms.head holds the head of the queue
56.         if (next == nil)
57.             ms.rvals[k] = nil;
58.         else {
59.             ms.rvals[k] = next;
60.             ms.head = next;
            }
61.         diffs = diffs XOR (1 << k);                 // exclude p_k from the set of active dequeuers
        }
62.     ms.applied = lactive;                           // change applied to be equal to the value read in DeqAct
63.     ms.seq2 = ms.seq2 + 1;
64.     mp.tm = lp.tm + 1;                              // increase timestamp
65.     mp.index = n*i + deq_pool_index_i;              // store in mp.index the index in DeqPool where ms will be stored
66.     DeqPool[n * i + deq_pool_index_i] = ms;         // store the new state in position mp.index of DeqPool
67.     if (CAS(DeqP, lp, mp)) {                        // try to change DeqP to point to mp
68.         deq_pool_index_i = (deq_pool_index_i + 1) mod C;    // use next item in p_i's dequeue pool next time
69.         return ms.rvals[pid];
        }
    }
70. lp = DeqP;                                          // after two unsuccessful efforts, read current value of DeqP
71. ms = DeqPool[lp.index];                             //read the element of DeqPool indicated by the index field of DeqP,
72. return ms.rvals[p_i];                               // and return the value found in the record stored there
}
```

Algorithm 7 Data structures used in L-Sim algorithm.

```
typedef struct NewVar {
   ItemSV *var;
   NewVar *next;
} NewVar;

typedef struct NewList {
   ItemSV *first;
} NewList;

typedef struct State {
   boolean applied[1..n], papplied[1..n];
   RetVal rvals[1..n];
   int seq;
   NewList *var_list;
} State;

typedef struct DirectoryNode {
    Name name;
    ItemSV *sv;
    Value val;
} DirectoryNode;

typedef struct ItemSV {
   Value val[0..1];
   int toggle;
   int seq;
} ItemSV;

shared ActiveSet Act = ⊥;
shared State S = << F, ..., F >, < F, ..., F >, < ⊥, ..., ⊥ >, 0, < ⊥ >>;
shared OpType Announce[1..n] = {⊥, ..., ⊥};

RetVal ApplyOp(operation op){  // Pseudocode for process p_i
1. Anounce[i] = op;            // Announce the operation
2. JOIN(Act);                  // Join the active set
3. Attempt();                  // Execute Attempt twice
4. Attempt();
5. LEAVE(Act);                 // Leave the active set
6. Attempt();                  // Eliminate any evidence of op
7. return S.rvals[i];
}
```

Acknowledgments. We would like to thank Dimitris Nikolopoulos, Angelos Bilas and Manolis Katevenis for several useful discussions. We especially thank Dimitris Nikolopoulos for arranging the provision of access to some of the multi-core machines of the Department of Computer Science at Virginia Tech where we run our experiments. Many thanks to Nir Shavit and Danny Hendler for providing the code of flat combining. Thanks also to Faith Ellen for several useful comments on a preliminary version of this paper [11], and for suggesting to combine techniques from Sim and the algorithm in [7] to get the best of both worlds.

Algorithm 8 Pseudocode of L-Sim algorithm.

```
void Attempt()(operation op){          // pseudocode for process p_i
   Pindex q, j;   State ls, tmp;   Set lact;   DirectoryNode D;
   NewVar *pvar = new NewVar(), *ltop;   ItemSV sv, *psv = new ItemSV();
8.  psv → ⟨val, toggle, seq⟩ = << ⊥, ⊥ >, 0, 0 >;  pvar → ⟨var, next, ⟩ = < psv, nill >;
9.  for j = 1 to 2 do {
10.    D = ∅;
11.    ls = LL(S);                                          // read State struct
12.    lact = GETSET(Act);                                  // read active set
13.    ltop = ls.var_list → first;                          // read pointer to the current variable list
14.    tmp.seq = ls.seq + 1;
15.    tmp.papplied[1..n] = ls.applied[1..n];       // p will attempt to update S with tmp
16.    for q = 1 to n do                                    // local loop
17.       if(q ∈ lact) tmp.applied[q] = TRUE;
18.       else tmp.applied[q] = FALSE;
19.    for q = 1 to n do {                                  // local loop
20.       if (ls.applied[q] == TRUE AND ls.papplied[q] == FALSE) {   // if appropriate conditions hold,
21.          foreach access of a variable x while applying operation Announce[q]{ //apply operation of process q
22.             if (x is a newly allocated variable) {
23.                if(CAS(ltop → next, nil, pvar)){ // try to insert a new node for the new variable in list
24.                   psv = new ItemSV();  psv → ⟨val, toggle, seq⟩ = << ⊥, ⊥ >, 0, 0 >;  // in case of success,
25.                   pvar = new NewVar();  pvar → ⟨var, next, ⟩ = < psv, nill >;          // allocate new pvar
                   }
26.                ltop = ltop → next;              // in any case, use ltop → next as the new variable's metadata
27.                add < x, ltop → var, ltop → var.val[0] > to D;          // add variable to local dictionary
28.             }else {                                      // if x is not a newly allocated variable
29.                let svp be a pointer to the ItemSV struct for x;
30.                if this access is a read instruction {
31.                  if (x exists in D) read x from D;          // perform the operation on the local copy of x (if any)
32.                    else {sv = LL(*svp);
33.                       if(tmp.seq == sv.seq) add < x, svp, sv.val[1 − sv.toggle] to D;
34.                       else if(tmp.seq > sv.seq) add < x, svp, sv.val[sv.toggle] to D;
35.                       else goto Line 38;                 // the State read by p is obsolete, start from scratch
                     }
36.                } else if (this access is a write instruction) update x in D;  // perform operation on local copy
             }
          }
37.          store into tmp.rvals[q] the return value;
       }
    }
38.    if (ls != S) continue;                       // the State read by p is obsolete, start from scratch
39.    foreach record < x, svp, v > in D {
40.       if(svp → seq > tmp.seq) return;           // if all operations have been applied, return
41.       else if(svp → seq == tmp.seq) continue;  // if variable R_x has already been modified, continue
42.       else if(svp → toggle == 0) SC(*svp, << svp → val[0], v >, 1, tmp.seq >);   // make update visible
43.       else SC(*svp, << v, svp → val[1] >, 0, tmp.seq >);          // make update visible
    }
44.    tmp.var_list = new List();  tmp.var_list → first = nil;          // re-initiate tmp.var_list to point to nill
45.    SC(S, tmp);                                          // try to modify S
  }
}
```

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[2] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, pages 262–272, 1999.

[3] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[4] James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, dec 1999.

[5] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 336–343, 2008.

[6] 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.

[7] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.

[8] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Blade computing with the amd opteron processor (magny-cours). *Hot chips 21*, August 2009.

[9] 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, February 1993.

[10] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue adaptive universal constractions. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 127–141, 2009.

[11] Panagiota Fatourou and Nikolaos D. Kallimanis. Fast implementations of shared objects using fetch&add. Technical Report TR 02-2010, Department of Computer Science, University of Ioannina, February 2010.

[12] D. George S. Harvey W. Kleinfelder K. McAuliffe E. Melton V. Norton G. Pfister, W. Brantley and J. Weiss. The ibm research parallel processor prototype (rp3): Introduction and architecture. pages 764–771, 1985.

[13] P. Heidelberger, A. Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers.*, 39(1):133–138, 1990.

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. The code for flat combining. http://github.com/mit-carbon/flat-combining.

[15] 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.

[16] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th ACM Symposium on Parallel Algorithms and Architectures*, pages 206–215, 2004.

[17] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, jan 1991.

[18] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, nov 1993.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[20] Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 142–156. Springer, 2009.

[21] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[22] 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.

[23] 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.

[24] 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.

[25] Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. A quantitative architectural evaluation of synchronization algorithms and disciplines on ccnuma systems: the case of the sgi origin2000. In *Proceedings of the 13th international conference on Supercomputing (ICS '99)*, pages 319–328, New York, NY, USA, 1999. ACM.

[26] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *EuroSys*, pages 305–315, 2006.

[27] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[28] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

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

[30] Pen-Chung Yew, Nian-Feng Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388 –395, April 1987.