SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools

Elad Gidron
CS Department
Technion, Haifa, Israel
eladgi@cs.technion.ac.il

Idit Keidar
EE Department
Technion, Haifa, Israel
idish@ee.technion.ac.il

Dmitri Perelman*
EE Department
Technion, Haifa, Israel
dima39@tx.technion.ac.il

Yonathan Perez
EE Department
Technion, Haifa, Israel
yonathan0210@gmail.com

ABSTRACT

We present a highly-scalable non-blocking producer-consumer task pool, designed with a special emphasis on lightweight synchronization and data locality. The core building block of our pool is *SALSA, Scalable And Low Synchronization Algorithm* for a single-consumer container with task stealing support. Each consumer operates on its own SALSA container, stealing tasks from other containers if necessary. We implement an elegant self-tuning policy for task insertion, which does not push tasks to overloaded SALSA containers, thus decreasing the likelihood of stealing.

SALSA manages large chunks of tasks, which improves locality and facilitates stealing. SALSA uses a novel approach for coordination among consumers, without strong atomic operations or memory barriers in the fast path. It invokes only two CAS operations during a chunk steal.

Our evaluation demonstrates that a pool built using SALSA containers scales *linearly* with the number of threads and significantly outperforms other FIFO and non-FIFO alternatives.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Multi-core, concurrent data structures

*This work was partially supported by Hasso Plattner Institute.

SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.

1. INTRODUCTION

Emerging computer architectures pose many new challenges for software development. First, as the number of computing elements constantly increases, the importance of *scalability* of parallel programs becomes paramount. Second, accessing memory has become the principal bottleneck, while multi-CPU systems are based on NUMA architectures, where memory access from different chips is asymmetric. Therefore, it is instrumental to design software with *local data access*, *cache-friendliness*, and *reduced contention* on shared memory locations, especially across chips. Furthermore, as systems get larger, their behavior becomes less predictable, underscoring the importance of *robust* programs that can overcome unexpected thread stalls.

Our overarching goal is to devise a methodology for developing parallel algorithms addressing these challenges. In this paper, we focus on one of the fundamental building blocks of highly parallel software, namely a producer-consumer task pool. Specifically, we present a scalable and highly-efficient non-blocking pool, with lightweight synchronization-free operations in the common case. Its data allocation scheme is cache-friendly and highly suitable for NUMA environments. Moreover, our pool is robust in the face of imbalanced loads and unexpected thread stalls.

Our system is composed of two independent logical entities: 1) *SALSA, Scalable and Low Synchronization Algorithm*, a single-consumer pool that exports a stealing operation, and 2) a work stealing framework implementing a management policy that operates multiple SALSA pools.

In order to improve locality and facilitate stealing, SALSA keeps tasks in chunks, organized in per-producer chunk lists. Only the producer mapped to a given list can insert tasks to chunks in this list, which eliminates the need for synchronization among producers.

Though each consumer has its own task pool, inter-consumer synchronization is required in order to allow stealing. The challenge is to do so without resorting to costly atomic operations (such as CAS or memory fences) upon each task retrieval. We address this challenge via a novel chunk-based stealing algorithm that allows consume operations to be synchronization-free in the common case, when no stealing occurs, which we call the *fast path*. Moreover, SALSA reduces the stealing rate by moving entire chunks of tasks in one steal operation, which requires only two CAS operations.

In order to achieve locality of memory access on a NUMA architecture, SALSA chunks are kept in the consumer's local memory. The management policy matches producers and consumers according to their proximity, which allows most task transfers to occur within a NUMA node.

In many-core machines running multiple applications, system behavior becomes less predictable. Unexpected thread stalls may lead to an asymmetric load on consumers, which may in turn lead to high stealing rates, hampering performance. SALSA employs a novel auto-balancing mechanism that has producers insert tasks to less loaded consumers, and is thus robust to spurious load fluctuations.

We have implemented SALSA in C++, and tested its performance on a 32-core NUMA machine. Our experiments show that the SALSA-based work stealing pool *scales linearly* with the number of threads; it is 20 times faster than other work-stealing alternatives, and shows a significant improvement over state-of-the-art non-FIFO alternatives. SALSA-based pools scale well even in unbalanced scenarios.

This paper proceeds as follows. Section 2 describes related work. We give the system overview in Section 3. The SALSA single-consumer algorithm is described in Section 4 and its correctness is discussed in Section 5. We discuss our implementation and experimental results in Section 6, and finally conclude in Section 7.

2. RELATED WORK

Task pools.

There is a large body of work on lock-free unbounded FIFO queues and LIFO stacks [12, 16, 17, 21, 22]. However, due to the inherent need for ordering all operations, such algorithms generally have high contention and do not scale, and are therefore less appealing for use as task pools.

A number of previous works have recognized this limitation, and observed that strict FIFO order is seldom needed in multi-core systems [3, 4, 7, 24]. To the best of our knowledge, all previous solutions use strong atomic operations (like CAS), at least in every consume operation. Moreover, most of them [3, 4, 7] do not partition the pool among processors, and therefore do not achieve good locality and cache-friendliness, which has been shown to limit their scalability on NUMA systems [6].

The closest non-FIFO pool to our work is the Concurrent Bags of Sundell et al. [24], which, like SALSA, uses per-producer chunk lists. This work is optimized for the case that the same threads are both consumers and producers, and typically consume from themselves, while SALSA improves the performance of such a task pool in NUMA environments where producers and consumers are separate threads. Unlike our pool, the Concurrent Bags algorithm uses strong atomic operations upon each consume. In addition, steals are performed in the granularity of single tasks and not whole chunks as in SALSA. Overall, their throughput does not scale linearly with the number of participating threads, as shown in [24] and in Section 6 of this paper.

Techniques.

Variations of techniques we employ were previously used in various contexts. Work-stealing [9] is a standard way to reduce contention by using individual per-consumer pools, where tasks may be stolen from one pool to another. We improve the efficiency of stealing by transferring a chunk of tasks upon every steal operation. Hendler et al. [15] have proposed stealing of multiple items by copying a range of tasks from one dequeue to another, but this approach requires costly CAS operations on the fast-path and introduces non-negligible overhead for item copying. In contrast, our approach of chunk-based stealing coincides with our synchronization-free fast-path, and steals whole chunks in O(1) steps. Furthermore, our use of page-size chunks allows for data migration in NUMA architectures to improve locality, as done in [8].

The principle of keeping NUMA-local data structures was previously used by Dice et al. for constructing scalable NUMA locks [11]. Similarly to their work, our algorithm's data allocation scheme is designed to reduce inter-chip communication.

The concept of a synchronization-free fast-path previously appeared in works on scheduling queues, e.g., [5, 14]. However, these works assume that the same process is both the producer and the consumer, and hence the synchronization-free fast-path is actually used only when a process transfers data to *itself*. On the other hand, our pool is synchronization-free even when tasks are transfered among multiple threads; our synchronization-free fast-path is used also when multiple producers produce data for a single consumer. We do not know of any other work that supports synchronization-free data transfer among different threads.

The idea of organizing data in chunks to preserve locality in dynamically-sized data structures was previously used in [10, 12, 14, 24]. SALSA extends on the idea of chunk-based data structures by using chunks also for efficient stealing.

3. SYSTEM OVERVIEW

In the current section we present our framework for scalable and NUMA-aware producer-consumer data exchange. Our system follows the principle of separating mechanism and policy. We therefore consider two independent logical entities:

1. *A single consumer pool (SCPool)* mechanism manages the tasks arriving to a given consumer and allows tasks stealing by other consumers.

2. A management policy operates SCPools: it routes producer requests to the appropriate consumers and initiates stealing between the pools. This way, the policy controls the system's behavior according to considerations of load-distribution, throughput, fairness, locality, etc. We are especially interested in a management policy suitable for NUMA architectures (see Figure 1), where each CPU has its own memory, and memories of other CPUs are accessed over an interconnect. As a high rate of remote memory accesses can decrease the performance, it is desirable for the SCPool of a consumer to reside close to its own CPU.

SCPool abstraction.

The SCPool API provides the abstraction of a single-consumer task pool with stealing support, see Algorithm 1. A producer invokes two operations: **produce()**, which attempts to insert a task to the given pool and fails if the pool

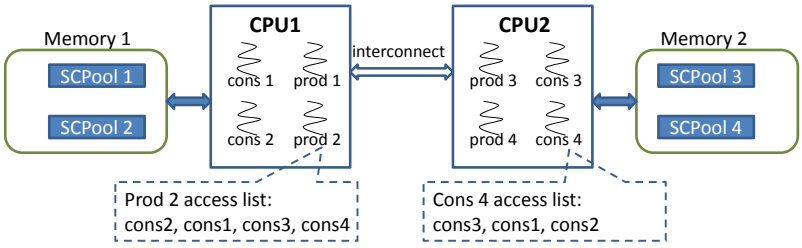


Figure 1: Producer-consumer framework overview. In this example, there are two processors connected to two memory banks (NUMA architecture). Two producers and two consumers running on each processor, and the data of each consumer is allocated at the closest physical memory. A producer (consumer) has a sorted access list of consumers for task insertion (respectively stealing).

Algorithm 1 API for a Single Consumer Pool with stealing support.

```
1: boolean: produce(Task, SCPool)   ▷ Tries to insert the task to the pool, returns false if no space is available.
2: void: produceForce(Task, SCPool) ▷ Insert the task to the pool, expanding the pool if necessary.
3: {Task ∪⊥}: consume()             ▷ Retrieve a task from the pool, returns ⊥ if no tasks in the pool are detected.
4: {Task ∪⊥}: steal(SCPool from)    ▷ Try to steal a number of tasks from the given pool and move them to the current pool. Return some stolen task or ⊥.
```

is full, and **produceForce()**, which always succeeds by expanding the pool on demand. There are also two ways to retrieve a task from the pool: the owner of the pool (only) can call the **consume()** function; while any other thread can invoke **steal()**, which tries to transfer a number of tasks between two pools and return one of the stolen tasks.

A straightforward way to implement the above API is using dynamic-size multi-producer multi-consumer FIFO queue (e.g., Michael-Scott queue [21]). In this case, **produce()** enqueues a new task, while **consume()** and **steal()** dequeue a task. In the next section we present SALSA, a much more efficient SCPool.

Algorithm 2 Work stealing framework pseudo-code.

```
5:  Local variables:
6:    SCPool myPool ▷ The consumer's pool
7:    SCPool[] accessList
8:  Function get():
9:    while(true)
10:     ▷ First try to get a task from the local pool
11:     t ← myPool.consume()
12:     if (t ≠ ⊥) return t
13:     ▷ Failed to get a task from the local pool – steal
14:     foreach SCPool p in accessList in order do:
15:       t ← p.steal()
16:       if (t ≠ ⊥) return t
17:     ▷ No tasks found – validate emptiness
18:     if (checkEmpty()) return ⊥
19: Function put(Task t):
20:   ▷ Produce to the pools by the order of the access list
21:   foreach SCPool p in accessList in order do:
22:     if (p.produce(t)) return
23:   firstp ← the first entry in accessList
24:   ▷ If all pools are full, expand the closest pool
25:   produceForce(t,firstp)
26:   return
```

Management policy.

A management policy defines the way in which: 1) a producer chooses an SCPool for task insertion; and 2) a consumer decides when to retrieve a task from its own pool or steal from other pools. Note that the policy is independent of the underlying SCPool implementation. We believe that the policy is a subject for engineering optimizations, based on specific workloads and demands.

In the current work, we present a NUMA-aware policy. If the individual SCPools themselves are lock-free, then our policy preserves lock-freedom at the system level. Our policy is as follows:

- **Access lists.** Each process in the system (producer or consumer) is provided with an *access list*, an ordered list of all the consumers in the system, sorted according to their distance from that process (see Figure 1). Intuitively, our intention is to have a producer mostly interact with the closest consumer, while stealing mainly happens inside the same processor node.

- **Producer's policy.** The producer policy is implemented in the **put**() function in Algorithm 2. The operation first calls the **produce()** of the first SCPool in its access list. Note that this operation might fail if the pool is full, (which can be seen as evidence of that the corresponding consumer is overloaded). In this case, the producer tries to insert the task into other pools, in the order defined by its access list. If all insertions fail, the producer invokes **produceForce()** on the closest SCPool, which always succeeds (expanding the pool if needed).

- **Consumer's policy.** The consumer policy is implemented in the **get**() function in Algorithm 2. A consumer takes tasks from its own SCPool. If its SCPool is empty, then the consumer tries to steal tasks from other pools in the order defined by its access list. The **checkEmpty**() operation handles the issue of when a consumer gives up and returns ⊥. This is subtle issue, and we discuss it in Section 5. Stealing serves two purposes: first, it is important for distributing the load among all available consumers. Second, it ensures that tasks are not lost in case they are inserted into the SCPool of a crashed (or very slow) consumer.

4. ALGORITHM DESCRIPTION

In the current section we present the SALSA SCPool. We first show the data structures of SALSA in Section 4.1, and then present the basic algorithm without stealing support in Section 4.2. The stealing procedure is described in Section 4.3, finally, the role of chunk pools is presented in Section 4.4. For the simplicity of presentation, in this section we assume that the the memory accesses satisfy sequential consistency [19], we describe the ways to solve memory reordering issues in Section 6.1.

4.1 SALSA Structure

Algorithm 3 SALSA implementation of SCPool: Data Structures.

```
27: Chunk type
28:     Task[CHUNK_SIZE] tasks
29:     int owner ▷ owner's consumer id
30: Node type
31:     Chunk c; initially ⊥
32:     int idx; initially -1
33:     Node next;
34: SALSA per consumer data structure:
35:     int consumerId
36:     List⟨Node⟩[] chunkLists ▷ one list per producer + extra list for stealing (every list is single-writer multi-reader)
37:     Queue⟨Chunk⟩ chunkPool ▷ pool of spare chunks
38:     Node currentNode, initially ⊥ ▷ current node to work with
```

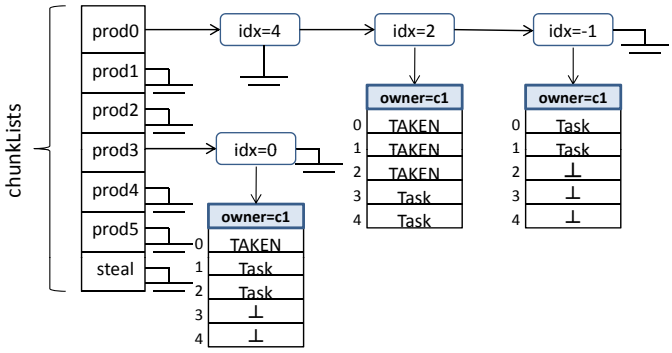


Figure 2: Chunk lists in SALSA single consumer pool implementation. Tasks are kept in chunks, which are organized in per-producer lists; an additional list is reserved for stealing. Each list can be modified by the corresponding producer only. The only process that is allowed to retrieve tasks from a chunk is the owner of that chunk (defined by the ownership flag). A Node's index corresponds to the latest task taken from the chunk or the task that is about to be taken by the current chunk owner.

The SALSA data structure of a consumer c_i is described in Algorithm 3 and partially depicted in Figure 2. The tasks inserted to SALSA are kept in chunks, which are organized in per-producer chunk lists. Only the producer mapped to a given list can insert a task to any chunk in that list. Every chunk is owned by a single consumer whose id is kept in the *owner* field of the chunk. The owner is the only process that is allowed to take tasks from the chunk; if another process wants to take a task from the chunk, it should first steal the chunk and change its ownership. A task entry in a chunk is used at most once. Its value is $\perp$ before the task is inserted, and TAKEN after it has been consumed.

The per-producer chunk lists are kept in the array *chunkLists* (see Figure 2), where *chunkLists[j]* keeps a list of chunks with tasks inserted by producer p_j. In addition, the array has a special entry *chunkLists[steal]*, holding chunks stolen by c_i. Every list has a single writer who can modify the list structure (add or remove nodes): *chunkLists[j]*'s modifier is the producer p_j, while *chunkLists[steal]*'s modifer is the SCPool's owner. The nodes of the used chunks are lazily reclaimed and removed by the list's owner. For brevity, we omit the linked list manipulation functions from the pseudo-code bellow. Our single-writer lists can be implemented without synchronization primitives, similarly to the single-writer linked-list in [20]. In addition to holding the chunk, a node keeps the index of the latest taken task in that chunk, this index is then used for chunk stealing as we show in Section 4.3.

Safe memory reclamation is provided by using hazard pointers [20] both for nodes and for chunks. The free (reclaimed) chunks in SALSA are kept at per-consumer *chunkPools* implemented by lock-free Michael-Scott queues [21]. As we show in Section 4.4, the chunk pools serve two purposes: 1) efficient memory reuse and 2) producer-based load balancing.

4.2 Basic Algorithm

4.2.1 SALSA producer

The description of SALSA producer functions is presented in Algorithm 4. The insertion of a new task consists of two stages: 1) finding a chunk for task insertion (if necessary), and 2) adding a task to the chunk.

Finding a chunk.

The chunk for task insertions is kept in the local producer variable *chunk* (line 41 in Algorithm 4). Once a producer starts working with a chunk c, it continues inserting tasks to c until c is full – the producer is oblivious to chunk stealing. If the *chunk*'s value is $\perp$, then the producer should start a new chunk (function *getChunk*). In this case, it tries to retrieve a chunk from the chunk pool and to append it to the appropriate chunk list. If the chunk pool is empty then the producer either returns $\perp$ (if force=false), or allocates a new chunk by itself (otherwise) (lines 56–58).

Inserting a task to the chunk.

As previously described in Section 4.1, different producers insert tasks to different chunks, which removes the need for synchronization among producers. The producer local variable *prodIdx* indicates the next free slot in the chunk. All that is left for the insertion function to do, is to put a task in that slot and to increment *prodIdx* (line 48). Once the index reaches the maximal value, the *chunk* variable is set to $\perp$, indicating that the next insertion operation should start a new chunk.

4.2.2 SALSA consumer without stealing

The consumer's algorithm without stealing is given in the

Algorithm 4 SALSA implementation of SCPool: Producer Functions.

```
39: Producer local variables:
40:   int producerId
41:   Chunk chunk; initially ⊥ ▷ the chunk to insert to
42:   int prodIdx; initially 0 ▷ the prefix of inserted tasks

43: Function produce(Task t, SCPool scPool):
44:   return insert(t, scPool, false)

45: Function insert(Task t, SCPool scPool, bool force):
46:   if (chunk = ⊥) then ▷ allocate new chunk
47:     if (getChunk(scPool, force) = false) then return
        false
48:   chunk.tasks[prodIdx] ← t; prodIdx++
49:   if(prodIdx = CHUNK_SIZE) then
50:     chunk ← ⊥ ▷ the chunk is full
51:   return true

52: Function produceForce(Task t, SCPool scPool):
53:   insert(t, scPool, true)

54: Function getChunk(SALSA scPool, bool force)
55:   newChunk ← dequeue chunk from scPool.chunkPool
56:   if (chunk = ⊥) ▷ no available chunks in this pool
57:     if (force = false) then return false
58:     newChunk ← allocate a new chunk
59:   newChunk.owner ← scPool.consumerId
60:   node ← new node with idx = −1 and c = newChunk
61:   scPool.chunkLists[producerId].append(node)
62:   chunk ← newChunk; prodIdx ← 0
63:   return true
```

left column of Algorithm 5. The consumer first finds a nonempty chunk it owns and then invokes **takeTask()** to retrieve a task.

Unlike producers, which have exclusive access to insertions in a given chunk, a consumer must take into account the possibility of stealing. Therefore, it should notify other processes which task it is about to take.

To this end, each node in the chunk list keeps an index of the taken prefix of its chunk in the *idx* variable, which is initiated to -1. A consumer that wants to take a task T, first increments the index, then checks the chunk's ownership, and finally changes the chunk entry from T to *TAKEN* (lines 78–80). By doing so, a consumer guarantees that *idx* always points to the last taken task or to a task that is about to be taken. Hence, a process that is stealing a chunk from a node with $idx = i$ can assume that the tasks in the range $[0 \dots i)$ have already been taken. The logic for dealing with stolen chunks is described in the next section.

4.3 Stealing

The stealing algorithm is given in the function **steal()** in Algorithm 5. We refer to the stealing consumer as c_s, the victim process whose chunk is being stolen as c_v, and the stolen chunk as *ch*.

The idea is to turn c_s to the exclusive owner of *ch*, such that c_s will be able to take tasks from the chunk without synchronization. In order to do that, c_s changes the ownership of *ch* from c_v to c_s using CAS (line 96) and removes the chunk from c_v's list (line 104). Once c_v notices the change in the ownership it can take at most one more task from *ch* (lines 83–86).

When the **steal()** operation of c_s occurs simultaneously with the **takeTask()** operation of c_v, both c_s and c_v might try to retrieve the same task. We now explain why this might happen. Recall that c_v notifies potential stealers of the task it is about to take by incrementing the *idx* value in *ch*'s node (line 78). This value is copied by c_s in line 99 when creating a copy of *ch*'s node for its steal list.

Consider, for example, a scenario in which the *idx* is incremented by c_v from 10 to 11. If c_v checks *ch*'s ownership before it is changed by c_s, then c_v takes the task at index 11 *without synchronization* (line 80). Therefore, c_s cannot be allowed to take the task pointed by *idx*. Hence, c_v has to take the task at index 11 even if it does observe the ownership change. After stealing the chunk, c_s will eventually try to take the task pointed by $idx+1$. However, if c_s copies the node before *idx* is incremented by c_v, c_s might think that the value of $idx + 1$ is 11. In this case, both c_s and c_v will try to retrieve the task at index 11. To ensure that the task is not retrieved twice, both invoke CAS in order to retrieve this task (line 108 for c_s, line 83 for c_v).

The above algorithm works correctly as long as the stealing consumer can observe the node with the updated index value. This might not be the case if the same chunk is concurrently stolen by another consumer rendering the *idx* of the original node obsolete. In order to prevent this situation, stealing a chunk from the pool of consumer c_v is allowed only if c_v is the owner of this chunk (line 96). This approach is prone to the ABA problem: consider a scenario where consumer c_a is trying to steal from c_b, but before the execution of the CAS in line 96, the chunk is stolen by c_c and then stolen back by c_b. In this case, c_a's CAS succeeds but c_a has an old value of *idx*. To prevent this ABA problem, the owner field contains a "tag", which is incremented on every CAS operation. For brevity, tags are omitted from the pseudo-code.

A naïve way for c_s to steal the chunk from c_v would be first to change the ownership and then to move the chunk to the steal list. However, this approach may cause the chunk to "disappear" if c_s is stalled, because the chunk becomes inaccessible via the lists of c_s and yet c_s is its owner. Therefore, SALSA first adds the original node to the steal list of c_s, then change the ownership, and only then replaces the original node with a new one (lines 95–104).

4.4 Chunk Pools

As described in Section 4.1, each consumer keeps a pool of free chunks. When a producer needs a new chunk for adding a task to consumer c_i, it tries to get a chunk from c_i's chunk pool – if no free chunks are available, the **produce()** operation fails.

As described in Section 3, our system-wide policy defines that if an insertion operation fails, then the producer tries to insert a task to other pools. Thus, the producer avoids adding tasks to overloaded consumers, which in turn decreases the amount of costly steal operations. We further refer to this technique as producer-based balancing.

Another SALSA property is that a chunk is returned to the pool of a consumer that retrieves the latest task of this chunk. Therefore, the size of the chunk pool of consumer c_i

Algorithm 5 SALSA implementation of SCPool: Consumer Functions.

```
64: Function consume():
65:   if (currentNode ≠ ⊥) then ▷ common case
66:     t ← takeTask(currentNode)
67:     if (t ≠ ⊥) then return t
68:   foreach Node n in ChunkLists do: ▷ fair traversal
69:     if (n.c ≠ ⊥ ∧ n.c.owner = consumerId) then
70:       t ← takeTask(n)
71:       if (t ≠ ⊥) then currentNode ← n; return t
72:   currentNode ← ⊥; return ⊥

73: Function takeTask(Node n):
74:   chunk ← n.c
75:   if (chunk = ⊥) then return ⊥ ▷ stolen chunk
76:   task ← chunk.tasks[n.idx + 1]
77:   if (task = ⊥) then return ⊥ ▷ no inserted tasks
      ▷ tell the world you're going to take a task from idx
78:   n.idx++
79:   if (chunk.owner = consumerId) then ▷ common case
80:     chunk.tasks[n.idx] ← TAKEN
81:     checkLast(n)
82:     return task
      ▷ the chunk has been stolen, CAS the last task and
      go away
83:   success ← (task ≠ TAKEN ∧
          CAS(chunk.tasks[n.idx], task, TAKEN))
84:   if(success) then checkLast(n)
85:   currentNode ← ⊥
86:   return (success) ? task : ⊥

87: Function checkLast(Node n):
88:   if(n.idx + 1 = CHUNK_SIZE) then
89:     n.c ← ⊥; return chunk to chunkPool
90:     currentNode ← ⊥

91: Function steal(SCPool p):
92:   prevNode ← a node holding tasks, whose owner is p,
      from some list in p's pool ▷ different policies possible
93:   if (prevNode = ⊥) return ⊥ ▷ No Chunk found
94:   c ← prevNode.c; if (c = ⊥) then return ⊥
      ▷ make it restealable
95:   chunkLists[steal].append(prevNode)
96:   if (CAS(c.owner,p.consumerId,consumerId)=false)
97:     chunkLists[steal].remove(prevNode)
98:     return ⊥ ▷ failed to steal
99:   newNode ← copy of prevNode
100:  if (newNode.idx+1 = CHUNK_SIZE)
101:    chunkLists[steal].remove(prevNode)
102:    return ⊥
103:  replace prevNode with newNode in chunkLists[steal]
104:  prevNode.c ← ⊥
      ▷ done stealing the chunk, take one task from it
105:  idx ← newNode.idx
106:  task ← c.tasks[idx+1]
107:  if (task = ⊥) then return ⊥ ▷ still no task at idx+1
108:  if (task = TAKEN ∨
          !CAS(c.tasks[idx+1], task, TAKEN)) then
109:    task ← ⊥
110:  if (task ≠ ⊥) then checkLast(newNode)
111:  newNode.idx ← newNode.idx+1
112:  if (c.owner = consumerId) currentNode ← newNode
113:  return task
```

is proportional to the rate of c_i's task consumption. This property is especially appealing for heterogeneous systems – a faster consumer c_i, (e.g., one running on a stronger or less loaded core), will have a larger chunk pool, and so more **produce()** operations will insert tasks to c_i, automatically balancing the overall system load.

5. CORRECTNESS

Linearizability.

In the full version of the paper [13], we prove that SALSA does not return the same task twice. However, for our system to be linearizable, we must ensure that SALSA's **get()** operation returns $\perp$ only if the pool contains no tasks at some point during the consume operation. We describe a policy for doing so in a lock-free manner.

Let us examine why a naïve approach, of simply traversing all task pools and returning $\perp$ if no task is found, violates correctness. First, a consumer might "miss" one task added during its traversal, and another removed during the same traversal, as illustrated in Figure 3. In this case, a single traversal would have returned $\perp$ although the pool was not empty at any point during the consume operation. Second, a consumer may miss a task that is moved from one pool to another due to stealing. In order to identify these two cases, we add to each pool a special *emptyIndicator*, a bit array with a bit per-consumer, which is cleared every time the pool *may* become empty. In SALSA, this occurs when the last task in a chunk is taken or when a chunk is stolen. In addition, we implement a new function, **checkEmpty()** (full pseudo-code shown in [13]), which is called by the framework whenever a consumer fails to retrieve tasks from its pool and all other pools. This function return true only if there is a time during its execution when there are no tasks in the system. If **checkEmpty()** returns false, the consumer simply restarts its operation.

Denote by c the number of consumers in the system. The **checkEmpty()** function works as follows: the consumer traverses all SCPools, to make sure that no tasks are present. After checking a pool, the consumer sets its bit in *emptyIndicator* using CAS. The consumer repeats this traversal c times, where in all traversals except the first, it checks that its bit in *emptyIndicator* is set, i.e., that no chunks were emptied or removed during the traversal. The c traversals are needed in order to account for the case that other consumers have already stolen or removed tasks, but did not yet update *emptyIndicator*, and thus their operations were not detected by the consumer. Since up to $c-1$ pending operations by other consumers may empty pools before any *emptyIndicator* changes, it is guaranteed that among c traversals in which no chunks were seen and the *emptyIndicator* did not change, there is one during which the system indeed contains no tasks, and therefore it is safe to return

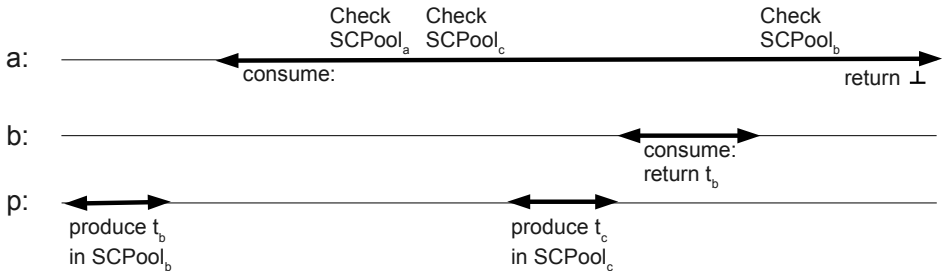


Figure 3: An example where a single traversal may violate linearizability: consumer a is trying to get a task. It fails to take a task from its own pool, and starts looking for chunks to steal in other pools. At this time there is a single non-empty chunk in the system, which is in b's pool; a checks c's pool and finds it empty. At this point, a producer adds a task to c's pool and then b takes the last task from its pool before a checks it. Thus, a finds b's pool empty, and returns $\perp$. There is no way to linearize this execution, because throughout the execution of a's operation, the system contains at least one task.

$\perp$. This method is similar to the one used in Concurrent Bags [24].

Lock-freedom.

The operations of every individual SALSA SCPool are trivially wait-free, since they always return. However, a **get**() operation is restarted whenever **checkEmpty()** returns false, and therefore the algorithm does not guarantee that a consumer will finish every operation. Nevertheless, as shown in the full version of the paper [13], the system is lock-free, i.e., there always exists some consumer that makes progress.

6. IMPLEMENTATION AND EVALUATION

In this section we evaluate the performance of our work-stealing framework built on SALSA pools. We first present the implementation details on dealing with memory reordering issues in Section 6.1. The experiment setup is described in Section 6.2, we show the overall system performance in Section 6.3, study the influence of various SALSA techniques in Section 6.4 and check the impact of memory placement and thread scheduling in Section 6.5.

6.1 Dealing with Memory Reordering

The presentation of the SALSA algorithm in Section 4 assumes sequential consistency [19] as the memory model. However, most existing systems relax sequential consistency to achieve better performance. Specifically, according to x86-TSO [23], memory loads can be reordered with respect to older stores to different locations. In SALSA, this reordering can cause an index increment to occur after the ownership validation (lines 78, 79 in Algorithm 5), which violates correctness as it may cause the same task to be taken twice, by both the original consumer and the stealing thread.

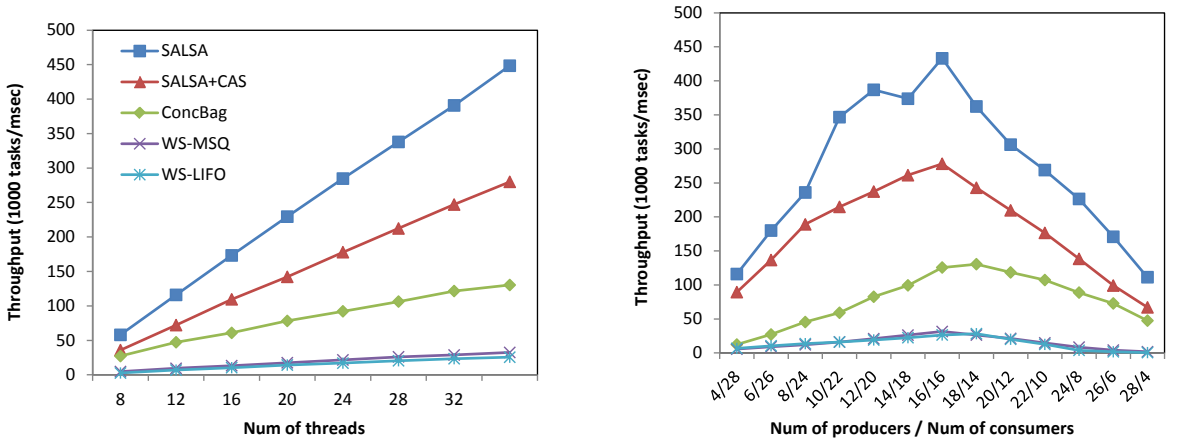
The conventional way to ensure a correct execution in such cases is to use memory fences to force a specific memory ordering. For example, adding an `mfence` instruction between lines 78 and 79 guarantees SALSA's correctness. However, memory fences are costly and their use in the common path degrades performance. Therefore, we prefer to employ a synchronization technique that does not add substantial overhead to the frequently used **takeTask()** operation. One example for such a technique is location-based memory fences, recently proposed by Ladan-Mozes et al. [18], which is unfortunately not implemented in current hardware.

In our implementation, we adopt the synchronization technique described by Dice et al. [1], where the slow thread (namely, the stealer) binds directly to the processor on which the fast thread (namely, the consumer) is currently running, preempting it from the processor, and then returns to run on its own processor. Thread displacement serves as a full memory fence, hence, a stealer that invokes the displacement binding right after updating the ownership (before the line 99 in Algorithm 5) observes the updated consumer's index. On the other hand, the steal-free fast path is not affected by this change.

6.2 Experiment Setup

We compare the following task pool implementations:

- **SALSA** – our work-stealing framework with SCPools implemented by SALSA.
- **SALSA+CAS** – our work-stealing framework with SCPools implemented by a simplistic SALSA variation, in which every **consume()** and **steal()** operation tries to take a single task using CAS. In essence, SALSA+CAS removes the effects of SALSA's low synchronization fast-path and per-chunk stealing. Note that disabling per-chunk stealing in SALSA annuls the idea of chunk ownership, hence, disables its low synchronization fast-path as well.
- **ConcBag** – an algorithm similar to the lock-free Concurrent Bags algorithm [24]. It is worth noting that the original algorithm was optimized for the scenario where the same process is both a producer and a consumer (in essence producing tasks to itself), which we do not consider in this paper; in our system no thread acts as both a producer and a consumer, therefore every consume operation steals a task from some producer. We did not have access to the original code, and therefore reimplemented the algorithm in our framework. Our implementation is faithful to the algorithm in the paper, except in using a simpler and faster underlined linked list algorithm. All engineering decisions were made to maximize performance.
- **WS-MSQ** – our work-stealing framework with SCPools implemented by Michael-Scott non-blocking queue [21]. Both **consume()** and **steal()** operations invoke the **dequeue()** function.
- **WS-LIFO** – our work-stealing framework with SCPool implemented by Michael's LIFO stack [20].



(a) System throughput – N producers, N consumers.

(b) System throughput – variable producers-consumers ratio.

Figure 4: System throughput for various ratios of producers and consumers. SALSA scales linearly with the number of threads – in the 16/16 workload, it is ×20 faster than WS-MSQ and WS-LIFO, and ×3.5 faster than Concurrent Bags. In tests with equal numbers of producers and consumers, the differences among work-stealing alternatives are mainly explained by the consume operation efficiency, since stealing rate is low and hardly influences performance.

We did not experiment with additional FIFO and LIFO queue implementations, because, as shown in [24], their performance is of the same order of magnitude as the Michael-Scott queue. Similarly, we did not evaluate CAFÉ [7] pools because their performance is similar to that of WS-MSQ [6], or ED-Pools [3], which have been shown to scale poorly in multi-processor architectures [6, 24].

All the pools are implemented in C++ and compiled with `-O2` optimization level. In order to minimize scalability issues related to allocations, we use `jemalloc` allocator, which has been shown to be highly scalable in multi-threaded environments [2]. Chunks of SALSA and SALSA+CAS contain 1000 tasks, and chunks of ConcBag contain 128 tasks, which were the respective optimal values for each algorithm (see the full version of the paper [13]).

We use a synthetic benchmark where 1) each producer works in a loop of inserting dummy items; 2) each consumer works in a loop of retrieving dummy items. Each data point shown is an average of 5 runs, each with a duration of 20 seconds. The tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor.

6.3 System Throughput

Figure 4(a) shows system throughput for workloads with equal number of producers and consumers. SALSA *scales linearly* as the number of threads grows to 32 (the number of physical cores in the system), and it clearly outperforms all other competitors. In the 16/16 workload, SALSA is ×20 faster than WS-MSQ and WS-LIFO, and more than ×3.5 faster than Concurrent Bags.

We note that the performance trend of ConcBags in our measurements differs from the results presented by Sundell et al. [24]. While in the original paper, their throughput *drops* by a factor of 3 when the number of threads increases from 4 to 24, in our tests, the performance of ConcBags *increases* with the number of threads. The reasons for the better scalability of our implementation can be related to the use of different memory allocators, hardware architectures, and engineering optimizations.

All systems implemented by our work-stealing framework scale linearly because of the low contention between consumers. Their performance differences are therefore due to the efficiency of the **consume()** operation – for example, SALSA is ×1.7 faster than SALSA+CAS thanks to its fast-path consumption technique.

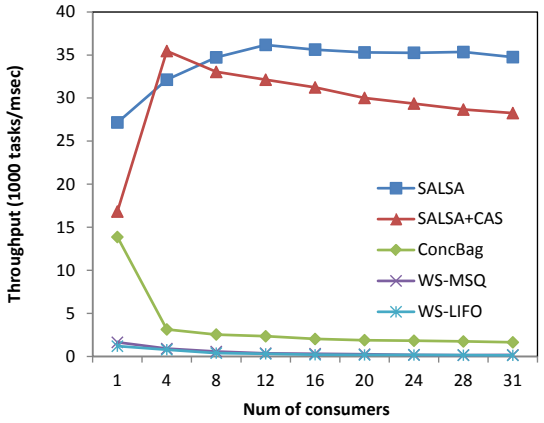
In ConcBags, which is not based on per-consumer pools, every **consume()** operation implies stealing, which causes contention among consumers, leading to sub-linear scalability. The stealing policy of ConcBags algorithm plays an important role. The stealing policy described in the original paper [24] proposes to iterate over the lists using round robin. We found out that the approach in which each stealer initiates stealing attempts from the predefined consumer improves ConcBags' results by 53% in a balanced workload.

Figure 4(b) shows system throughput of the algorithms for various ratios of producers and consumers. SALSA outperforms other alternatives in all scenarios, achieving its maximal throughput with equal number of producers and consumers, because neither of them is a system bottleneck.

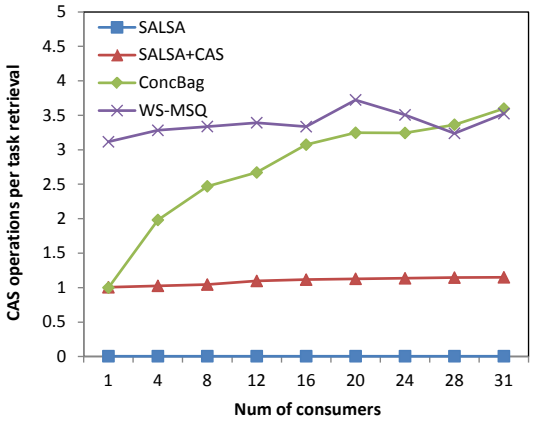
We next evaluate the behavior of the pools in scenarios with a single producer and multiple consumers. Figure 5(a) shows that the performance of both SALSA and SALSA+CAS does not drop as more consumers are added, while the throughput of other algorithms degrades by the factor of 10. The degradation can be explained by high contention among stealing consumers, as evident from Figure 5(b), which shows the average number of CAS operations per task transfer.

6.4 Evaluating SALSA techniques

In this section we study the influence of two of the techniques used in SALSA: 1) chunk-based-stealing with a low-synchronization fast path (Section 4.3), and 2) producer-based balancing (Section 4.4). To this end, we compare SALSA and SALSA+CAS both with and without producer-



(a) System throughput – 1 Producer, N consumers.



(b) CAS operations per task retrieval – 1 Producer, N consumers.

Figure 5: System behavior in workloads with a single producer and multiple consumers. Both SALSA and SALSA+CAS efficiency balance the load in this scenario. The throughput of other algorithms drops by a factor of 10 due to increased contention among consumers trying to steal tasks from the same pool.

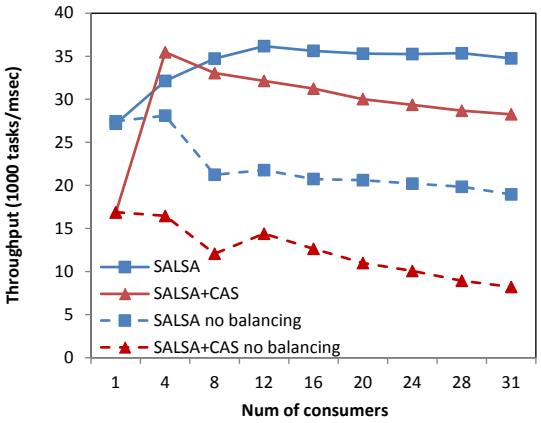


Figure 6: System throughput – 1 Producer, N consumers. Producer-based balancing contributes to the robustness of the framework by reducing stealing. With no balancing, chunk-based stealing becomes important.

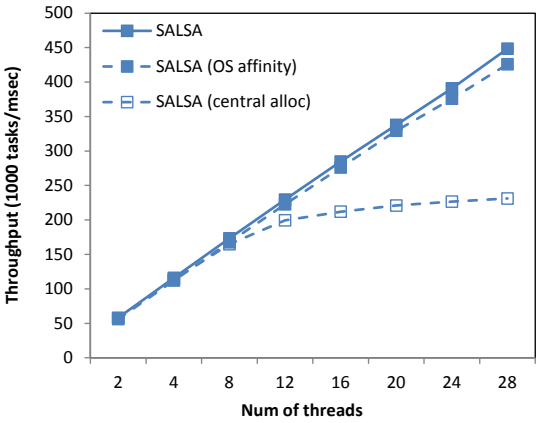


Figure 7: Impact of scheduling and allocation (equal number of producers and consumers). Performance decreases once the interconnect becomes saturated.

based balancing (in the latter a producer always inserts tasks to the same consumer's pool).

Figure 6 depicts the behavior of the four alternatives in single producer / multiple consumers workloads. We see that producer-based balancing is instrumental in redistributing the load: neither SALSA nor SALSA+CAS suffers any degradation as the load increases. When producer-based balancing is disabled, stealing becomes prevalent, and hence the stealing granularity becomes more important: SALSA's chunk based stealing clearly outperforms the naïve task-based approach of SALSA+CAS.

6.5 Impact of Scheduling and Allocation

We now evaluate the impact of scheduling and allocation in our NUMA system. To this end, we compare the following three alternatives: 1) the original SALSA algorithm; 2) SALSA with no affinity enforcement for the threads s.t. producers do not necessarily work with the closest consumers; 3) SALSA with all the memory pools preallocated on a single NUMA node.

Figure 7 depicts the behavior of all the variants in the balanced workload. The performance of SALSA with no predefined affinities is almost identical to the performance of the standard SALSA, while the central allocation alternative looses its scalability after 12 threads.

The main reason for performance degradation in NUMA systems is bandwidth saturation of the interconnect. If all chunks are placed on a single node, every remote memory access is transfered via the interconnect of that node, which causes severe performance degradation. In case of random affinities, remote memory accesses are distributed among different memory nodes, hence their rate remains below the maximum available bandwidth of each individual channel, and the program does not reach the scalability limit.

7. CONCLUSIONS

We presented a highly-scalable task pool framework, built upon our novel SALSA single-consumer pools and work stealing. Our work has employed a number of novel techniques

for improving performance: 1) lightweight and synchronization-free produce and consume operations in the common case; 2) NUMA-aware memory management, which keeps most data accesses inside NUMA nodes; 3) a chunk-based stealing approach that decreases the stealing cost and suits NUMA migration schemes; and 4) elegant producer-based balancing for decreasing the likelihood of stealing.

We have shown that our solution scales linearly with the number of threads. It outperforms other work-stealing techniques by a factor of 20, and state-of-the art non-FIFO pools by a factor of 3.5. We have further shown that it is highly robust to imbalances and unexpected thread stalls.

We believe that our general approach of partitioning data structures among threads, along with chunk-based migration and an efficient synchronization-free fast-path, can be of benefit in building additional scalable high-performance services in the future.

8. REFERENCES

[1] `http://home.comcast.net/~pjbishop/Dave/Asymmetric-Dekker-Synchronization.txt`.

[2] `www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`.

[3] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 151–162, 2010.

[4] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 395–410.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, 1998.

[6] D. Basin. Café: Scalable task pools with adjustable fairness and contention. Master's thesis, Technion, 2011.

[7] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. Café: scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 475–488, 2011.

[8] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, 2011.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[10] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 107–118, 2011.

[11] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, 2011.

[12] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, 2010.

[13] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools. Technical report, Technion, 2012.

[14] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput.*, 18:189–207, February 2006.

[15] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, 2002.

[16] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 206–215, 2004.

[17] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th international conference on Principles of distributed systems*, OPODIS'07, pages 401–414, 2007.

[18] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 75–84, 2011.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, pages 690–691, 1979.

[20] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[22] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.

[23] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, pages 89–97, 2010.

[24] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 335–344, 2011.