Nonblocking k-compare-single-swap

Victor Luchangco
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
victor.luchangco@sun.com

Mark Moir
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
mark.moir@sun.com

Nir Shavit
Tel Aviv University
Tel Aviv 69978, Israel
shanir@cs.tau.ac.il

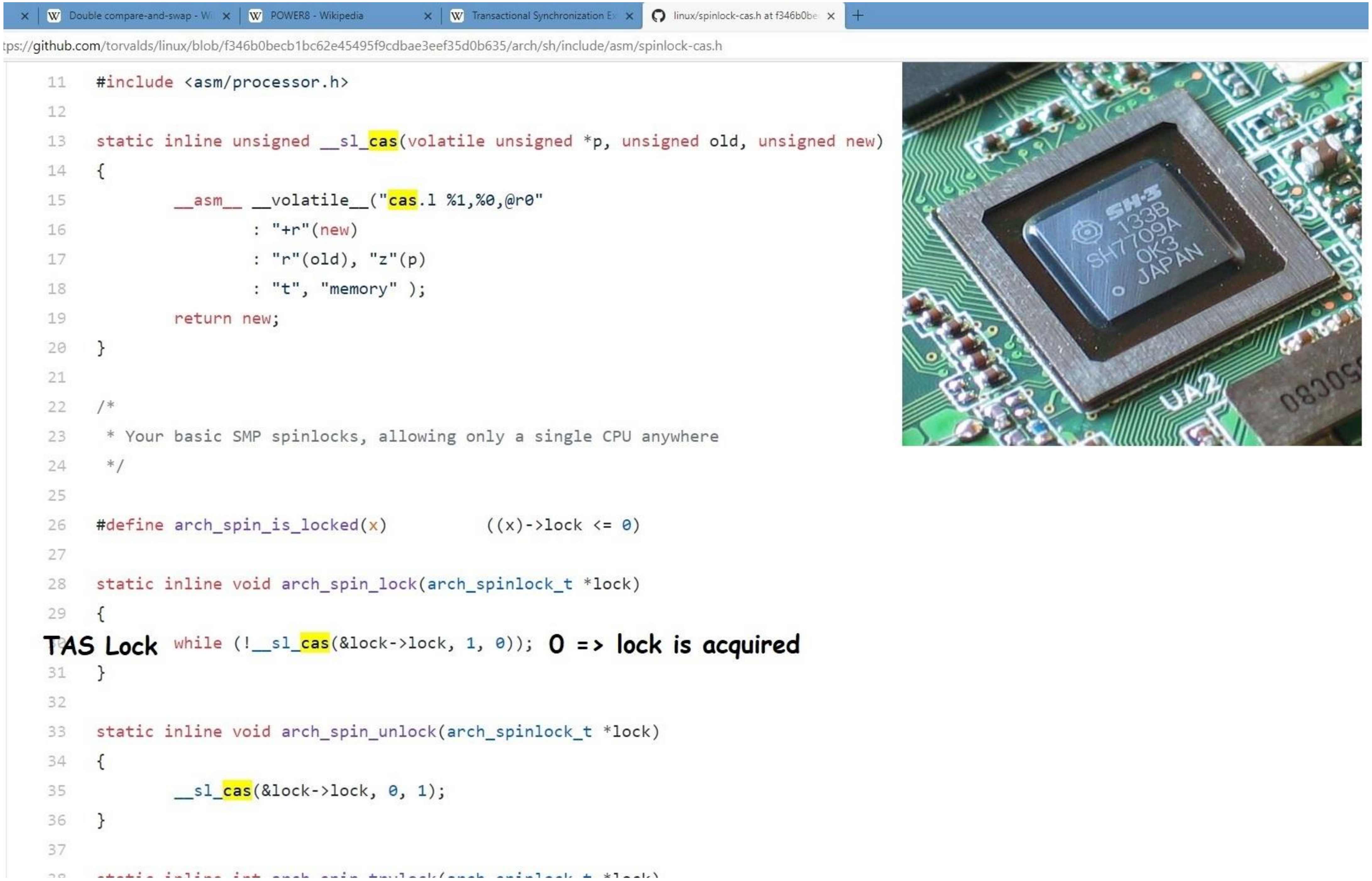
ABSTRACT

The current litera extremes of no software synchroni or concurrent d ture design: intricate designs of specific structures based on single-location operations such as compare-and-swap (CAS), and general-purpose multilocation transactional memory implementations. While the former are sometimes efficient, they are invariably hard to extend and generalize. The latter are flexible and general, but costly. This paper aims at a middle ground: reasonably efficient multilocation operations that are general enough to reduce the design difficulties of algorithms based on CAS alone.

TRODUCT

plementation o uctures is much ne can apply at ultiple memory locations [7]. Current a r, support atomic operations only on small, contiguous regions of memory (such as a single or double word) [6, 18, 27, 29].

The question of supporting multilocation operations in hardware has in recent years been a subject of debate in both industrial and academic circles. Until this question is resolved, and to a large extent to aid in its resolution, there is an urgent need to develop efficient software implementations of atomic multilocation operations.



Double compare-and-swap - Wi | POWER8 - Wikipedia | Transactional Synchronization E | linux/spinlock-cas.h at f346b0be

tps://github.com/torvalds/linux/blob/f346b0becb1bc62e45495f9cdbae3eef35d0b635/arch/sh/include/asm/spinlock-cas.h

```
#include <asm/processor.h>

static inline unsigned __sl_cas(volatile unsigned *p, unsigned old, unsigned new)
{
        __asm__ __volatile__("cas.l %1,%0,@r0"
                : "+r"(new)
                : "r"(old), "z"(p)
                : "t", "memory" );
        return new;
}

/*
 * Your basic SMP spinlocks, allowing only a single CPU anywhere
 */

#define arch_spin_is_locked(x)          ((x)->lock <= 0)

static inline void arch_spin_lock(arch_spinlock_t *lock)
{
```

TAS Lock

```
        while (!__sl_cas(&lock->lock, 1, 0));
```

0 => lock is acquired

```
}

static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
        __sl_cas(&lock->lock, 0, 1);
}

static inline int arch_spin_trylock(arch_spinlock_t *lock)
```

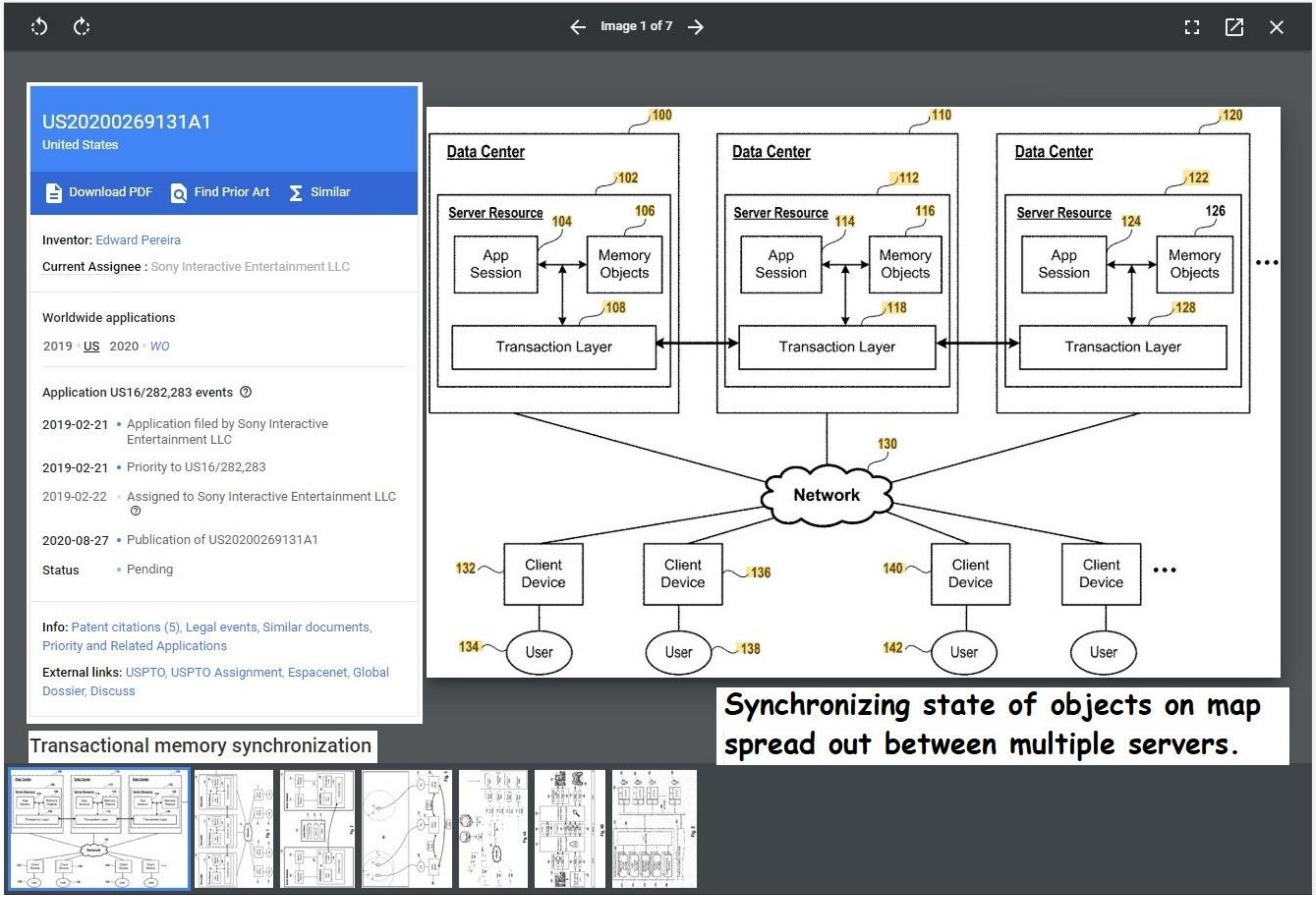


Image 1 of 7

US20200269131A1
United States

Download PDF | Find Prior Art | Similar

Inventor: Edward Pereira
Current Assignee : Sony Interactive Entertainment LLC

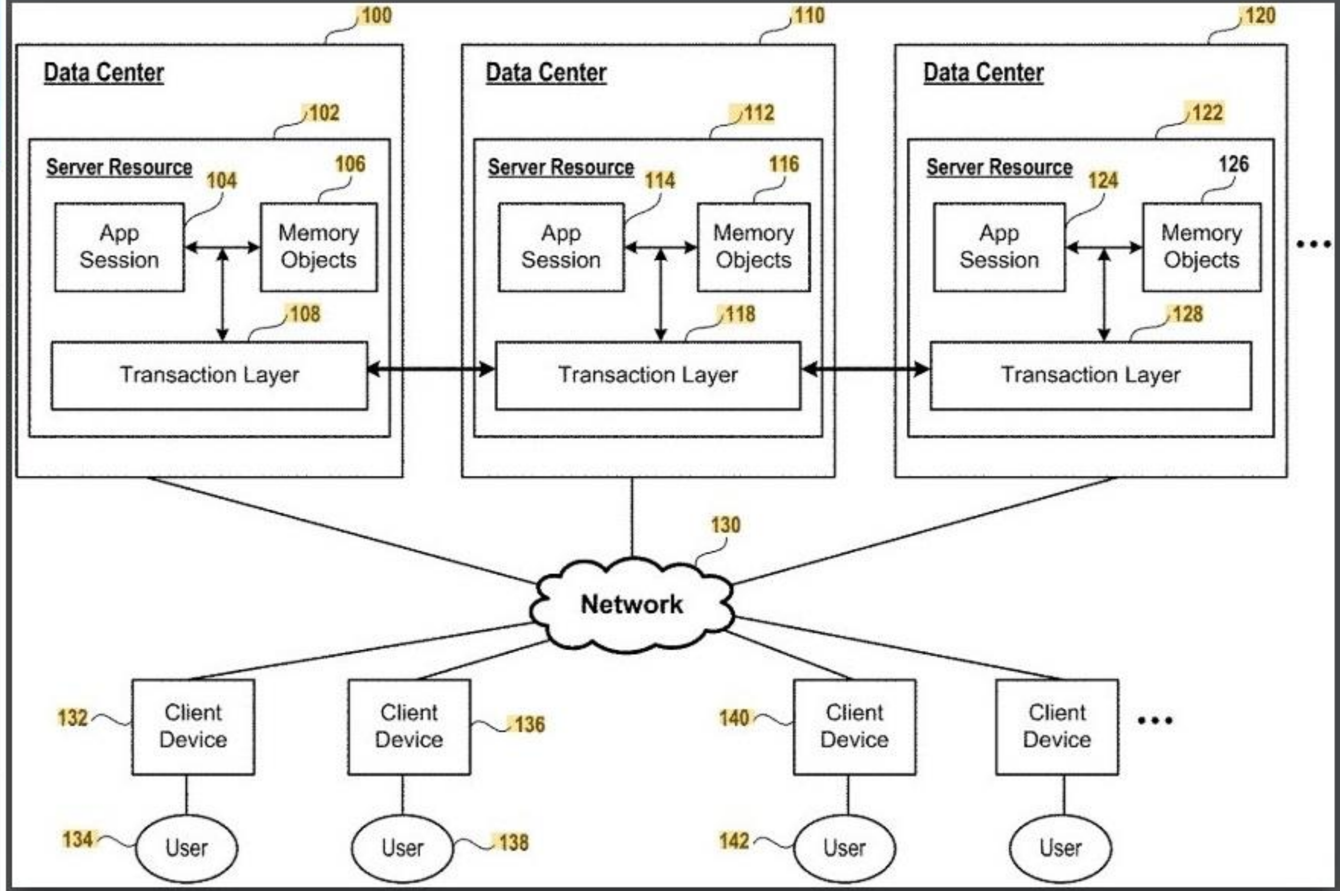
Worldwide applications
2019 · US 2020 · WO

Application US16/282,283 events

2019-02-21	Application filed by Sony Interactive Entertainment LLC
2019-02-21	Priority to US16/282,283
2019-02-22	Assigned to Sony Interactive Entertainment LLC
2020-08-27	Publication of US20200269131A1
Status	Pending

Info: Patent citations (5), Legal events, Similar documents, Priority and Related Applications

External links: USPTO, USPTO Assignment, Espacenet, Global Dossier, Discuss



Transactional memory synchronization

Synchronizing state of objects on map spread out between multiple servers.

:/Downloads/Paper.pdf

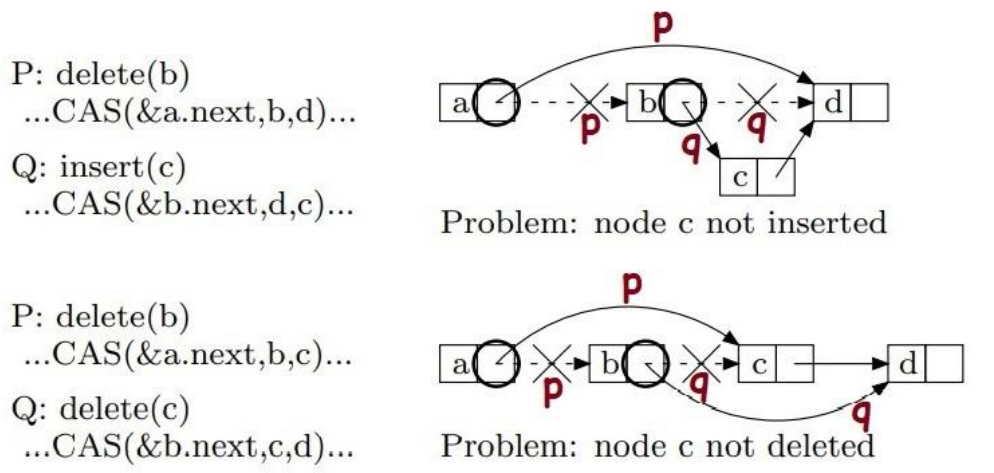


Figure 1: CAS-based list manipulation is hard (the examples slightly abuse CAS notation). In both examples, P is deleting b from the list. In the upper example, Q is trying to insert c into the list, and in the lower example, Q is trying to delete c from the list. Circled locations indicate the target addresses of the CAS operations; crossed out pointers are the values before the CAS succeeds.

the uncontended case, handling contended cases through orthogonal contention management mechanisms. (Contention management mechanisms similar in spirit to the ones dis-

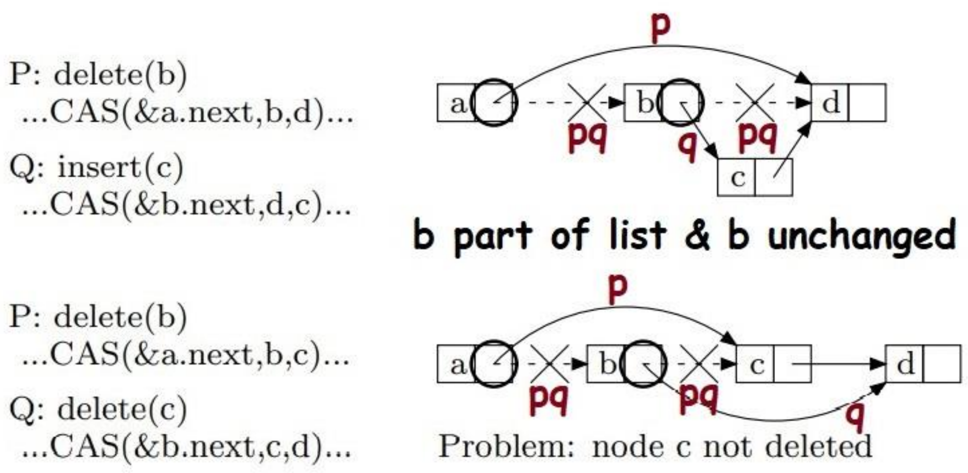


Figure 1: C DCAS can be useful here. **hard (the examples sli n both examples, P is the upper example, Q is trying to insert c into the list, and in the lower example, Q is trying to delete c from the list. Circled locations indicate the target addresses of the CAS operations; crossed out pointers are the values before the CAS succeeds.**

the uncontended case, handling contended cases through orthogonal contention management mechanisms. (Contention management mechanisms similar in spirit to the ones dis-

:/Downloads/Paper.pdf

KCSS is a natural tool for linked data structure manipulation; it allows a thread, while modifying a pointer, to check atomically that related nodes and pointers have not changed. An application of immediate importance is the implementation of nonblocking linked data structures with arbitrary insertions and deletions. As Figure 1 shows, naive approaches to implementing nonblocking insertion and deletion operations for a linked list using single-location CAS do not work. Although there exist effective (and rather ingenious) nonblocking algorithms for ordered list-based sets [8, 21], these algorithms do not generalize easily to arbitrary linked data structures. For example, it is not clear how to modify these algorithms to implement multisets.

By employing KCSS instead of CAS, we can simplify the design of arbitrary nonblocking linked-list operations. For example, Figure 2 illustrates how the use of KCSS can significantly simplify the design of a linked-list construct to support multiset operations (details of the illustrated algorithms are beyond the scope of this paper). For simplicity, the illustrated implementation uses a 4CSS operation to make sure the adjacent nodes have not changed during node removal; we can achieve the same purpose using KCSS operations that access only two locations at the cost of a slightly more intricate algorithm. However, adding a small number of additional locations to a KCSS operation is not prohibitive because the cost of verifying each additional location is only two noncached loads, a worthwhile tradeoff in many cases.

Toward building our KCSS algorithm, we provide a simple

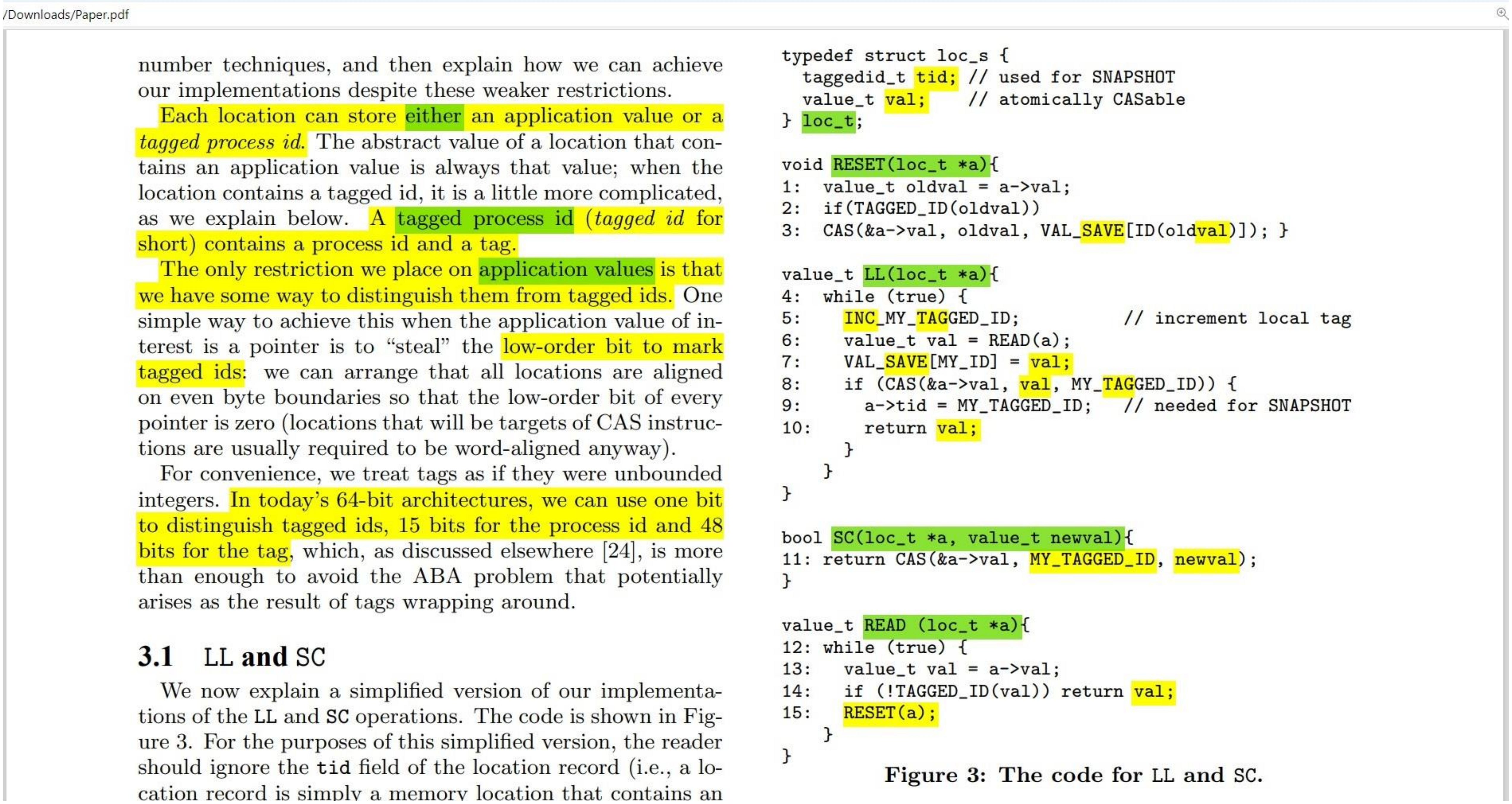
discuss various optimizations, generalizations, and extensions that are possible. In Section 5, we survey previous work on multilocation synchronization. We conclude in Section 6. A correctness proof appears in the appendix.

2. PRELIMINARIES

A k-location-compare single-swap (KCSS) operation takes k locations $a_1..a_k$, k expected values $e_1..e_k$, and a new value n_1. If the locations all contain the expected values, the KCSS operation atomically changes the first location a_1 from e_1 to n_1 and returns *true*; in this case, we say that the KCSS *succeeds*. Otherwise, the KCSS returns *false* and does not modify any memory location; in this case we say that it *fails*. In the next section, we present an implementation for KCSS using special read, load-linked (LL), store-conditional (SC) and snapshot operations that we have also implemented. In this section, we describe more precisely the interface and semantics of the various operations, the correctness requirements, and our assumptions about the system.

2.1 Semantics of operations

We now describe the semantics of the operations for which we provide implementations in the next section. We consider a collection of *locations*. At any point in time, each location has an *abstract value* from a set of *application values*. (As explained in the next section, our implementation requires some mild restrictions on this set.)



/Downloads/Paper.pdf

number techniques, and then explain how we can achieve our implementations despite these weaker restrictions.

Each location can store either an application value or a *tagged process id*. The abstract value of a location that contains an application value is always that value; when the location contains a tagged id, it is a little more complicated, as we explain below. A tagged process id (*tagged id* for short) contains a process id and a tag.

The only restriction we place on application values is that we have some way to distinguish them from tagged ids. One simple way to achieve this when the application value of interest is a pointer is to "steal" the low-order bit to mark tagged ids: we can arrange that all locations are aligned on even byte boundaries so that the low-order bit of every pointer is zero (locations that will be targets of CAS instructions are usually required to be word-aligned anyway).

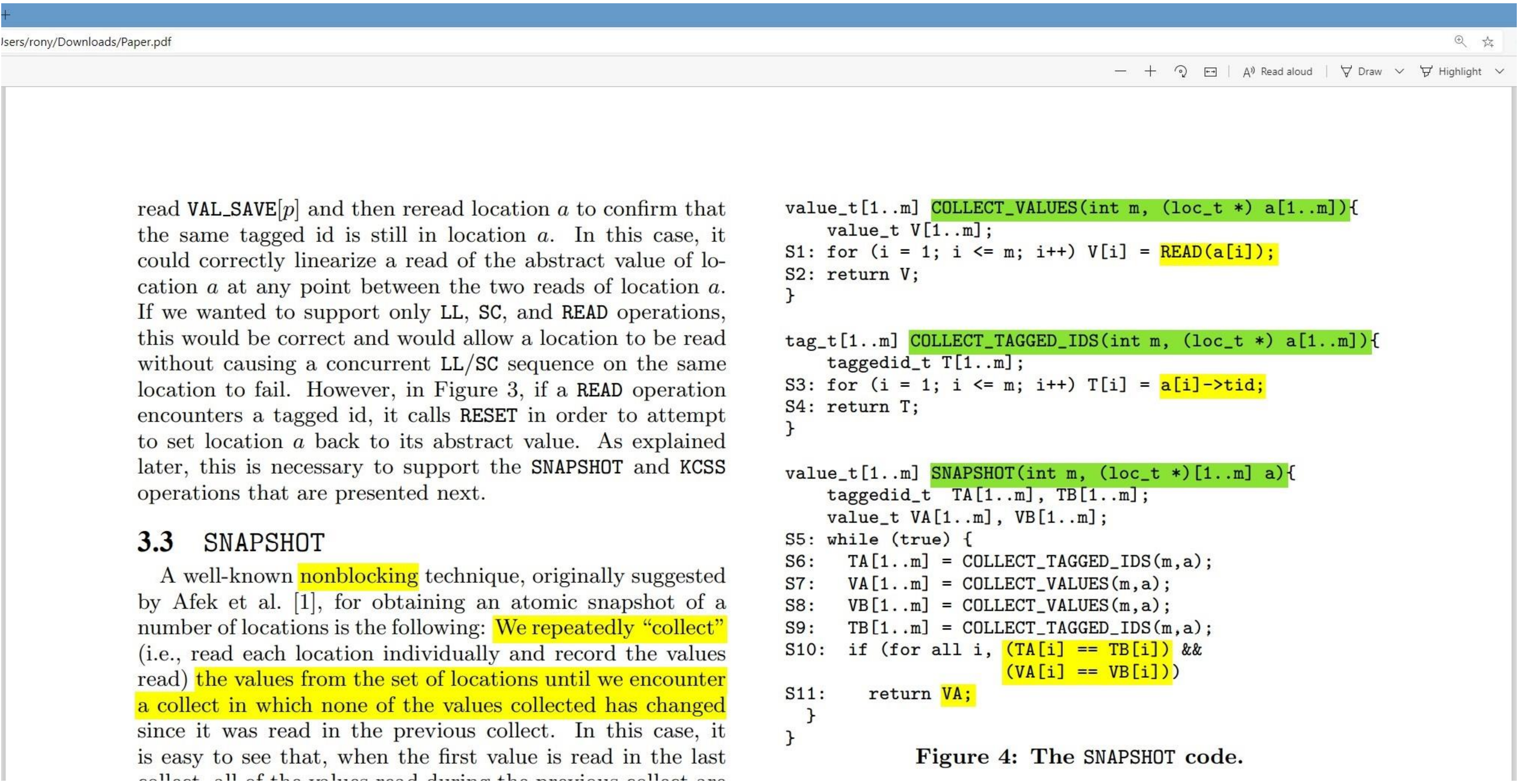
For convenience, we treat tags as if they were unbounded integers. In today's 64-bit architectures, we can use one bit to distinguish tagged ids, 15 bits for the process id and 48 bits for the tag, which, as discussed elsewhere [24], is more than enough to avoid the ABA problem that potentially arises as the result of tags wrapping around.

3.1 LL and SC

We now explain a simplified version of our implementations of the LL and SC operations. The code is shown in Figure 3. For the purposes of this simplified version, the reader should ignore the tid field of the location record (i.e., a location record is simply a memory location that contains an

```
typedef struct loc_s {
  taggedid_t tid; // used for SNAPSHOT
  value_t val;    // atomically CASable
} loc_t;

void RESET(loc_t *a){
    value_t oldval = a->val;
    if(TAGGED_ID(oldval))
    CAS(&a->val, oldval, VAL_SAVE[ID(oldval)]); }

value_t LL(loc_t *a){
    while (true) {
      INC_MY_TAGGED_ID;            // increment local tag
      value_t val = READ(a);
      VAL_SAVE[MY_ID] = val;
      if (CAS(&a->val, val, MY_TAGGED_ID)) {
        a->tid = MY_TAGGED_ID;    // needed for SNAPSHOT
        return val;
      }
    }
}

bool SC(loc_t *a, value_t newval){
    return CAS(&a->val, MY_TAGGED_ID, newval);
}

value_t READ (loc_t *a){
    while (true) {
      value_t val = a->val;
      if (!TAGGED_ID(val)) return val;
      RESET(a);
    }
}
```

Figure 3: The code for LL and SC.



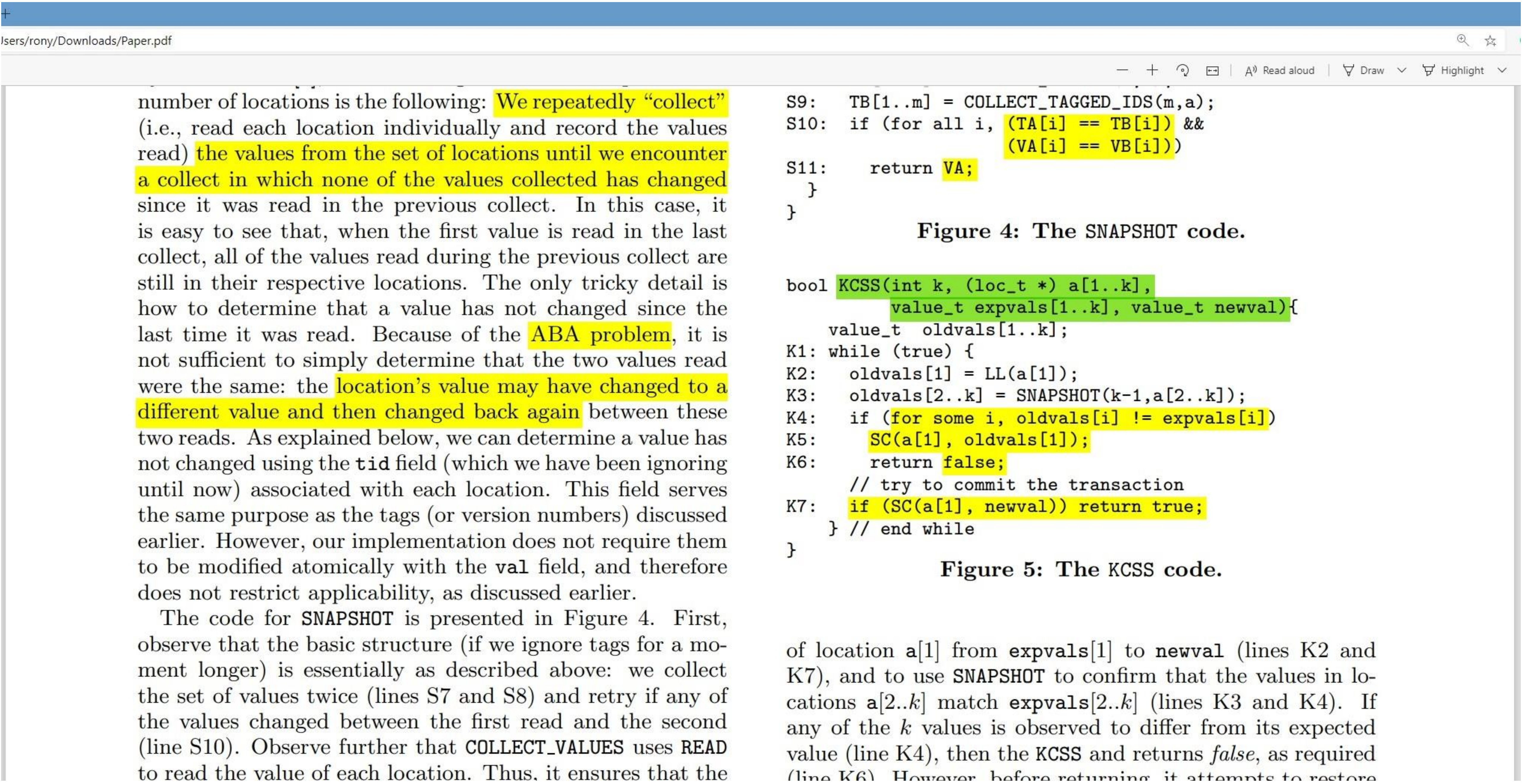
read VAL_SAVE$[p]$ and then reread location a to confirm that the same tagged id is still in location a. In this case, it could correctly linearize a read of the abstract value of location a at any point between the two reads of location a. If we wanted to support only LL, SC, and READ operations, this would be correct and would allow a location to be read without causing a concurrent LL/SC sequence on the same location to fail. However, in Figure 3, if a READ operation encounters a tagged id, it calls RESET in order to attempt to set location a back to its abstract value. As explained later, this is necessary to support the SNAPSHOT and KCSS operations that are presented next.

3.3 SNAPSHOT

A well-known nonblocking technique, originally suggested by Afek et al. [1], for obtaining an atomic snapshot of a number of locations is the following: We repeatedly "collect" (i.e., read each location individually and record the values read) the values from the set of locations until we encounter a collect in which none of the values collected has changed since it was read in the previous collect. In this case, it is easy to see that, when the first value is read in the last

```
value_t[1..m] COLLECT_VALUES(int m, (loc_t *) a[1..m]){
    value_t V[1..m];
S1: for (i = 1; i <= m; i++) V[i] = READ(a[i]);
S2: return V;
}

tag_t[1..m] COLLECT_TAGGED_IDS(int m, (loc_t *) a[1..m]){
    taggedid_t T[1..m];
S3: for (i = 1; i <= m; i++) T[i] = a[i]->tid;
S4: return T;
}

value_t[1..m] SNAPSHOT(int m, (loc_t *)[1..m] a){
    taggedid_t  TA[1..m], TB[1..m];
    value_t VA[1..m], VB[1..m];
S5: while (true) {
S6:   TA[1..m] = COLLECT_TAGGED_IDS(m,a);
S7:   VA[1..m] = COLLECT_VALUES(m,a);
S8:   VB[1..m] = COLLECT_VALUES(m,a);
S9:   TB[1..m] = COLLECT_TAGGED_IDS(m,a);
S10:  if (for all i, (TA[i] == TB[i]) &&
                     (VA[i] == VB[i]))
S11:    return VA;
  }
}
```

Figure 4: The SNAPSHOT code.



number of locations is the following: We repeatedly "collect" (i.e., read each location individually and record the values read) the values from the set of locations until we encounter a collect in which none of the values collected has changed since it was read in the previous collect. In this case, it is easy to see that, when the first value is read in the last collect, all of the values read during the previous collect are still in their respective locations. The only tricky detail is how to determine that a value has not changed since the last time it was read. Because of the ABA problem, it is not sufficient to simply determine that the two values read were the same: the location's value may have changed to a different value and then changed back again between these two reads. As explained below, we can determine a value has not changed using the `tid` field (which we have been ignoring until now) associated with each location. This field serves the same purpose as the tags (or version numbers) discussed earlier. However, our implementation does not require them to be modified atomically with the `val` field, and therefore does not restrict applicability, as discussed earlier.

The code for `SNAPSHOT` is presented in Figure 4. First, observe that the basic structure (if we ignore tags for a moment longer) is essentially as described above: we collect the set of values twice (lines S7 and S8) and retry if any of the values changed between the first read and the second (line S10). Observe further that `COLLECT_VALUES` uses `READ` to read the value of each location. Thus, it ensures that the

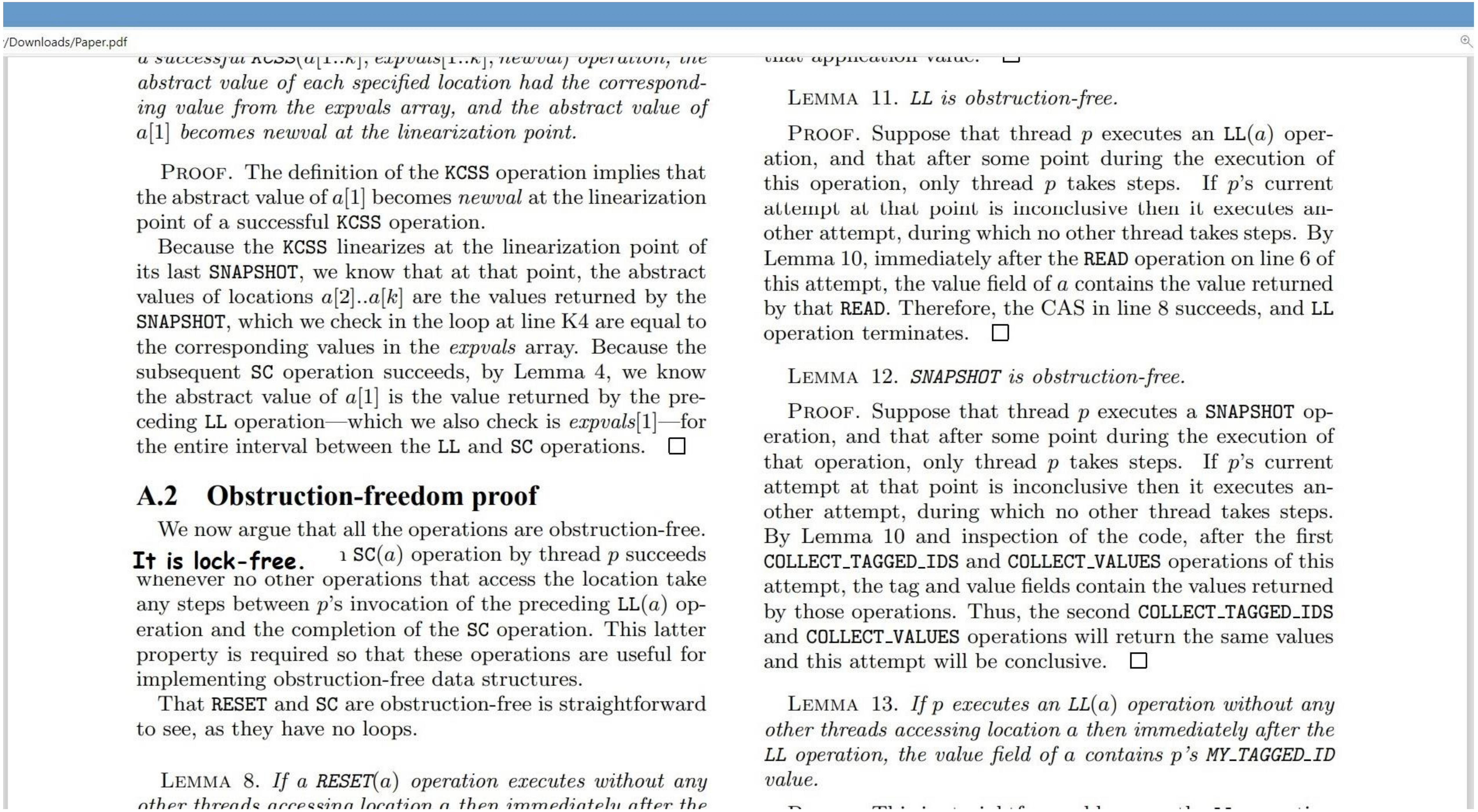
```
S9:   TB[1..m] = COLLECT_TAGGED_IDS(m,a);
S10:  if (for all i, (TA[i] == TB[i]) &&
                     (VA[i] == VB[i]))
S11:    return VA;
  }
}
```

Figure 4: The `SNAPSHOT` code.

```
bool KCSS(int k, (loc_t *) a[1..k],
          value_t expvals[1..k], value_t newval){
    value_t  oldvals[1..k];
K1: while (true) {
K2:   oldvals[1] = LL(a[1]);
K3:   oldvals[2..k] = SNAPSHOT(k-1,a[2..k]);
K4:   if (for some i, oldvals[i] != expvals[i])
K5:     SC(a[1], oldvals[1]);
K6:     return false;
      // try to commit the transaction
K7:   if (SC(a[1], newval)) return true;
    } // end while
}
```

Figure 5: The `KCSS` code.

of location `a`[1] from `expvals`[1] to `newval` (lines K2 and K7), and to use `SNAPSHOT` to confirm that the values in locations `a`[2..k] match `expvals`[2..k] (lines K3 and K4). If any of the k values is observed to differ from its expected value (line K4), then the `KCSS` and returns *false*, as required (line K6). However, before returning, it attempts to restore



/Downloads/Paper.pdf

a successful KCSS(a[1..k], expvals[1..k], newval) operation, the abstract value of each specified location had the corresponding value from the expvals array, and the abstract value of a[1] becomes newval at the linearization point.

Proof. The definition of the KCSS operation implies that the abstract value of a[1] becomes *newval* at the linearization point of a successful KCSS operation.

Because the KCSS linearizes at the linearization point of its last SNAPSHOT, we know that at that point, the abstract values of locations a[2]..a[k] are the values returned by the SNAPSHOT, which we check in the loop at line K4 are equal to the corresponding values in the *expvals* array. Because the subsequent SC operation succeeds, by Lemma 4, we know the abstract value of a[1] is the value returned by the preceding LL operation—which we also check is *expvals*[1]—for the entire interval between the LL and SC operations. □

A.2 Obstruction-freedom proof

We now argue that all the operations are obstruction-free.

It is lock-free. ı SC(a) operation by thread p succeeds whenever no other operations that access the location take any steps between p's invocation of the preceding LL(a) operation and the completion of the SC operation. This latter property is required so that these operations are useful for implementing obstruction-free data structures.

That RESET and SC are obstruction-free is straightforward to see, as they have no loops.

Lemma 8. *If a RESET(a) operation executes without any other threads accessing location a then immediately after the*

that application value. □

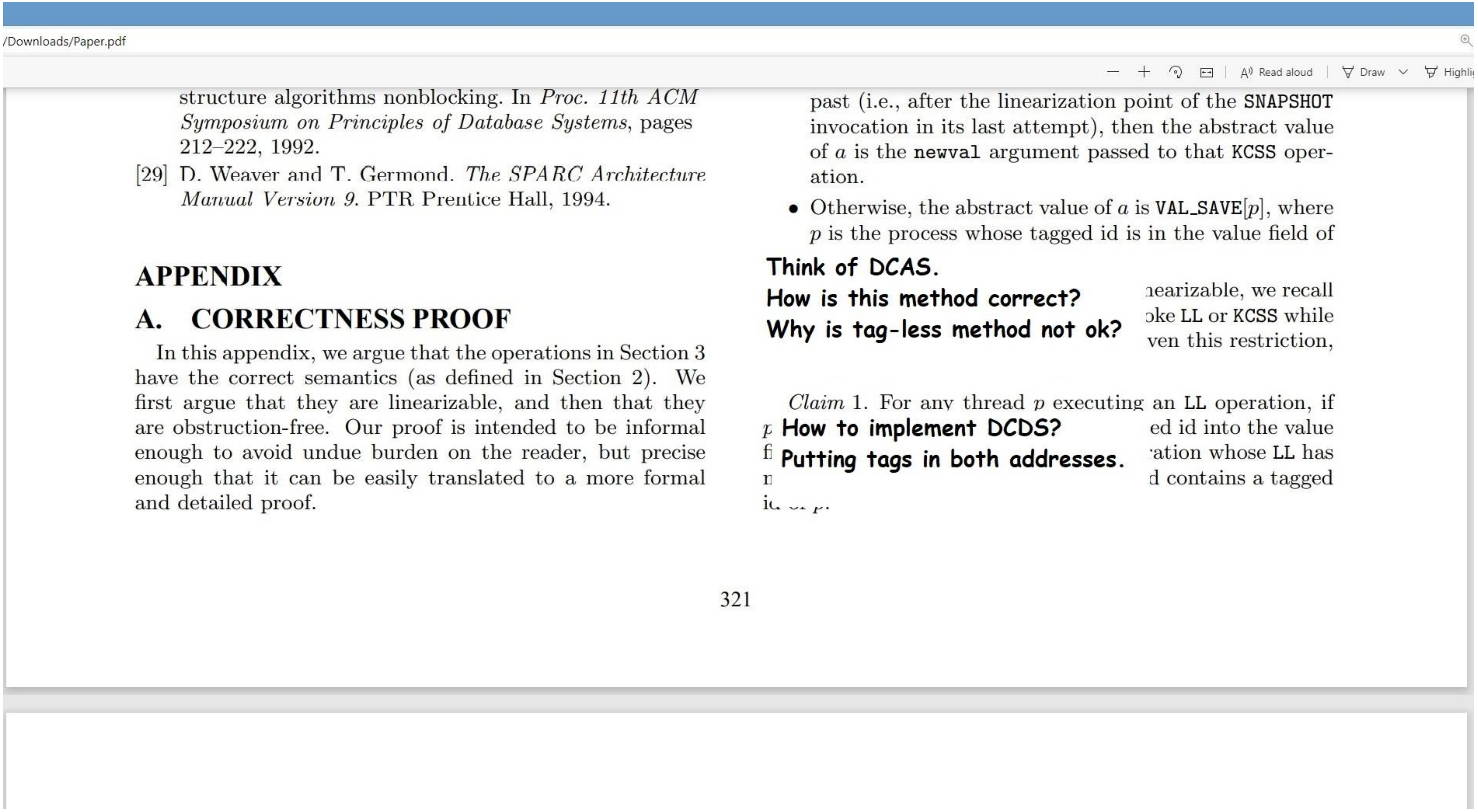
Lemma 11. *LL is obstruction-free.*

Proof. Suppose that thread p executes an LL(a) operation, and that after some point during the execution of this operation, only thread p takes steps. If p's current attempt at that point is inconclusive then it executes another attempt, during which no other thread takes steps. By Lemma 10, immediately after the READ operation on line 6 of this attempt, the value field of a contains the value returned by that READ. Therefore, the CAS in line 8 succeeds, and LL operation terminates. □

Lemma 12. *SNAPSHOT is obstruction-free.*

Proof. Suppose that thread p executes a SNAPSHOT operation, and that after some point during the execution of that operation, only thread p takes steps. If p's current attempt at that point is inconclusive then it executes another attempt, during which no other thread takes steps. By Lemma 10 and inspection of the code, after the first COLLECT_TAGGED_IDS and COLLECT_VALUES operations of this attempt, the tag and value fields contain the values returned by those operations. Thus, the second COLLECT_TAGGED_IDS and COLLECT_VALUES operations will return the same values and this attempt will be conclusive. □

Lemma 13. *If p executes an LL(a) operation without any other threads accessing location a then immediately after the LL operation, the value field of a contains p's MY_TAGGED_ID value.*



structure algorithms nonblocking. In *Proc. 11th ACM Symposium on Principles of Database Systems*, pages 212–222, 1992.

[29] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

APPENDIX

A. CORRECTNESS PROOF

In this appendix, we argue that the operations in Section 3 have the correct semantics (as defined in Section 2). We first argue that they are linearizable, and then that they are obstruction-free. Our proof is intended to be informal enough to avoid undue burden on the reader, but precise enough that it can be easily translated to a more formal and detailed proof.

past (i.e., after the linearization point of the `SNAPSHOT` invocation in its last attempt), then the abstract value of a is the `newval` argument passed to that `KCSS` operation.

- Otherwise, the abstract value of a is `VAL_SAVE`[p], where p is the process whose tagged id is in the value field of

Think of DCAS.

How is this method correct?

Why is tag-less method not ok?

ıearizable, we recall

oke `LL` or `KCSS` while

ven this restriction,

Claim 1. For any thread p executing an `LL` operation, if

p **How to implement DCDS?** ed id into the value

fi **Putting tags in both addresses.** ation whose `LL` has

n d contains a tagged

id of p.

/Downloads/Paper.pdf

to keep the presentation simple and clear.

4. OPTIMIZATIONS, EXTENSIONS, AND GENERALIZATIONS

From the basic ideas we have presented in this paper, numerous possible optimizations, extensions, and generalizations are possible. We describe a few of them here.

4.1 Optimizing

What is needed for KCKS?
Multiple tags.

Our READ operation
the CAS in line 3 of Fi
determined the abstract value of the location being accessed, which can be returned immediately without rereading.

4.2 Improving

Bank transaction with 2C2S.

As stated earlier, so that READ does not always have to reset a location that contains a tagged id: in some cases, reading a value from the VAL_SAVE location of the process whose tagged id is encountered, and then confirming that the tagged id is still in the location is sufficient to determine the correct abstract value. This does *not* work, however, in cases in which we linearize a modification to the location accessed by a LL/SC pair at a point other than the linearization point of the SC operation. In the operations we have presented, this is the case only for LL/SC sequences that are part of a higher-level KCSS operation. Therefore, if we extend the interface of LL so that the invoker can specify whether or not this is a "dangerous" use of LL/SC, then this information could be stored in the

collected at line S9 can be used for the first set of tags in the next iteration (we collect the tags again in the next iteration at line S6). Also, Harris [9] has noted that one can eliminate a complete sequence of reads from the snapshot, at the cost of a slightly more complex proof. We can also improve the performance of the KCSS by breaking the snapshot abstraction (for example, there is no need to take an entire snapshot if one of the early values read does not match the expected value).

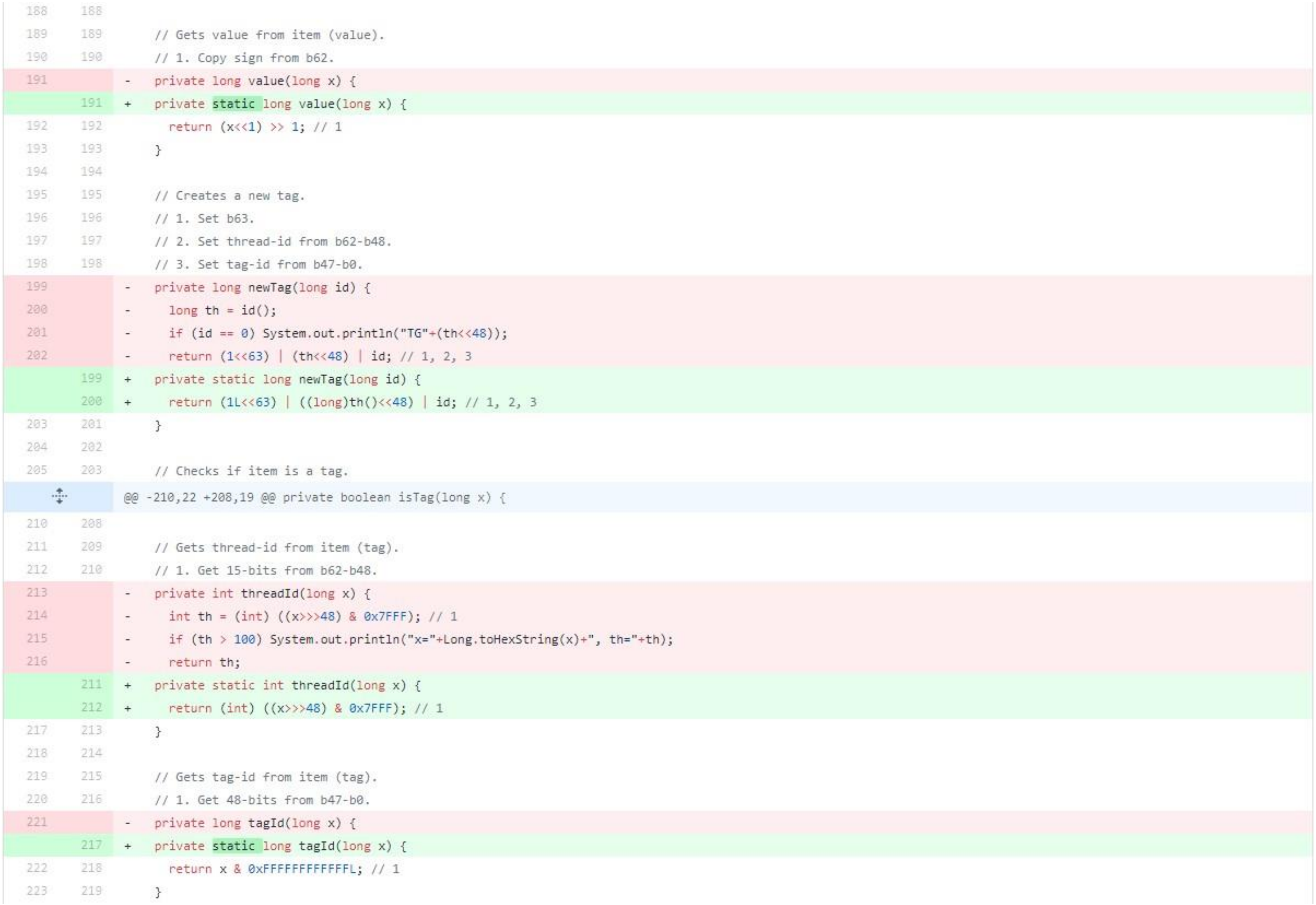
4.5 Single-modification transactions

We chose the KCSS API to demonstrate our ideas because its semantics is easy to state and understand. However, as we will show in the full paper, the ideas presented here can easily be extended to support *transactions* that modify only a single location. The basic idea is to have transactional loads record the information collected in the first half of the snapshot in our KCSS implementation, and transactional commit do the second half of the snapshot to determine if any of the values read had been modified by a concurrent operation since being read by the transactional load. In-te

This one can handle transaction with one write.

wo
an
fir
the transactional load. It would also be straightforward to provide a transactional "validate" operation that rechecks the values read so far in the transaction.

We believe that the ability provided by KCSS to "confirm" the abstract value of some locations, while modifying an-



```diff

   // Gets value from item (value).
   // 1. Copy sign from b62.
-  private long value(long x) {
+  private static long value(long x) {
     return (x<<1) >> 1; // 1
   }

   // Creates a new tag.
   // 1. Set b63.
   // 2. Set thread-id from b62-b48.
   // 3. Set tag-id from b47-b0.
-  private long newTag(long id) {
-    long th = id();
-    if (id == 0) System.out.println("TG"+(th<<48));
-    return (1<<63) | (th<<48) | id; // 1, 2, 3
+  private static long newTag(long id) {
+    return (1L<<63) | ((long)th()<<48) | id; // 1, 2, 3
   }

   // Checks if item is a tag.
@@ -210,22 +208,19 @@ private boolean isTag(long x) {

   // Gets thread-id from item (tag).
   // 1. Get 15-bits from b62-b48.
-  private int threadId(long x) {
-    int th = (int) ((x>>>48) & 0x7FFF); // 1
-    if (th > 100) System.out.println("x="+Long.toHexString(x)+", th="+th);
-    return th;
+  private static int threadId(long x) {
+    return (int) ((x>>>48) & 0x7FFF); // 1
   }

   // Gets tag-id from item (tag).
   // 1. Get 48-bits from b47-b0.
-  private long tagId(long x) {
+  private static long tagId(long x) {
     return x & 0xFFFFFFFFFFFFL; // 1
   }
```