Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models

TOMOHARU UGAWA, Kochi University of Technology, Japan
TATSUYA ABE and TOSHIYUKI MAEDA, STAIR Lab, Chiba Institute of Technology, Japan

Modern concurrent copying garbage collection (GC), in particular, real-time GC, uses fine-grained synchronizations with a mutator, which is the application program that mutates memory, when it moves objects in its copy phase. It resolves a data race using a concurrent copying protocol, which is implemented as interactions between the collector threads and the read and write barriers that the mutator threads execute. The behavioral effects of the concurrent copying protocol rely on the memory model of the CPUs and the programming languages in which the GC is implemented. It is difficult, however, to formally investigate the behavioral properties of concurrent copying protocols against various memory models.

To address this problem, we studied the feasibility of the bounded model checking of concurrent copying protocols with memory models. We investigated a correctness-related behavioral property of copying protocols of various concurrent copying GC algorithms, including real-time GC Stopless, Clover, Chicken, Staccato, and Schism against six memory models, total store ordering (TSO), partial store ordering (PSO), relaxed memory ordering (RMO), and their variants, in addition to sequential consistency (SC) using bounded model checking. For each combination of a protocol and memory model, we conducted model checking with a model of a mutator. In this wide range of case studies, we found faults in two GC algorithms, one of which is relevant to the memory model. We fixed these faults with the great help of counterexamples. We also modified some protocols so that they work under some memory models weaker than those for which the original protocols were designed, and checked them using model checking. We believe that bounded model checking is a feasible approach to investigate behavioral properties of concurrent copying protocols under weak memory models.

CCS Concepts: • **Software and its engineering → Garbage collection**; *Model checking*;

Additional Key Words and Phrases: garbage collection, memory model, model checking, concurrency

ACM Reference Format:
Tomoharu Ugawa, Tatsuya Abe, and Toshiyuki Maeda. 2017. Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 53 (October 2017), 26 pages. https://doi.org/10.1145/3133877

1 INTRODUCTION

Concurrent copying garbage collection (GC) [Jones et al. 2012] moves live objects while application threads, or mutator threads (*mutators*), are running. Because both collector threads (*collectors*) and mutators may access the heap simultaneously, concurrent copying GC requires careful synchronization between the collectors and mutators to ensure that objects are moved correctly. In particular, if they are not correctly synchronized in the copy phase, in which the collectors copy objects to be

Authors' addresses: Tomoharu Ugawa, Kochi University of Technology, Japan, ugawa.tomoharu@kochi-tech.ac.jp; Tatsuya Abe; Toshiyuki Maeda, STAIR Lab, Chiba Institute of Technology, Japan, {abet,tosh}@stair.center.

2475-1421/2017/10-ART53
https://doi.org/10.1145/3133877

moved to their new locations, when a mutator writes to an object that a collector is copying, the value that the mutator is writing may be lost.

Assume that a collector is copying the value in field `f` of object `o` to the corresponding field of its copy `oo` using local variable `x` in the following program:

```
x = o.f;
oo.f = x;
```

Additionally, assume that the mutators are using `o` at first, but change to using `oo` at some point. If a mutator writes to `o.f` after the collector executes the first line, the latest value is not copied, and so mutators will see the stale value after it changes to using `oo`. If a mutator writes to `oo.f` before the collector executes the second line, the latest value will be lost because it is overwritten on the second line. Although coarse-grained synchronizations such as locking objects with an ordinary mutex, can avoid this value lossage problem, modern concurrent copying GC, in particular real-time GC, uses fine-grained synchronizations [Hudson and Moss 2001, 2003; McCloskey et al. 2008; Pizlo et al. 2007b, 2008; Ritson et al. 2014], such as a compare-and-swap (CAS) instruction, for performance reasons and has the requirement of the lock-freedom of mutators. In this paper, we say that GC offers the "lock-freedom of mutators" if the read and write operations of at least one mutator can make progress.

In addition to the inherent difficulty of concurrent copying, the weak memory models of modern CPUs [Adve and Gharachorloo 1996] further complicate concurrent copying GC because read or write instructions, or both, may be executed out of order. Modern computing systems are based on concurrent/parallel processing designs for their performance advantages, and therefore, programs must also be written to exploit the designs. Writing such programs is, however, even more difficult and error-prone. For example, the x86 architecture [Intel 2016] allows writes to be reordered with succeeding reads. Such reordering can result in the value written by a mutator being lost, even in programs that should work properly provided that their instructions are executed in program order. In fact, we found such a fault in a concurrent copying GC algorithm, as described in Section 4.2.

Regarding correctness, the more synchronizations an implementation uses, the easier it is for the GC implementers to be confident of correctness. However, they would like to minimize the number of synchronizations because synchronizations impose a heavy overhead. Therefore, one challenge in concurrent copying GC is to design a *concurrent copying protocol* with a small but sufficient number of synchronizations. A concurrent copying protocol defines the following: (1) how the collectors copy objects; and (2) how the mutators should cooperate with the collectors. Mutator cooperation usually takes the form of *barriers* and *yieldpoints* [Azul 2008; Lin et al. 2015]. When a mutator reads from an object, it executes the *read barrier* procedure rather than reading directly from the memory location of the object. For writes, it executes the *write barrier* procedure. In addition to executing barrier procedures, each mutator may periodically poll a request for some work from the collectors at yieldpoints. Yieldpoints are typically used to ensure that the mutators have recognized the current value written by the collector.

However, there exists no formal and mechanical way to investigate the correctness of concurrent copying protocols under various memory models. For example, although various concurrent copying protocols have been proposed [McCloskey et al. 2008; Pizlo et al. 2007a, 2008], none of their authors have formally discussed their correctness against memory models, except for Sapphire [Hudson and Moss 2003], for which the correctness of its concurrent copying protocol was discussed in natural language. Such an investigative approach is also desired when porting a concurrent copying GC algorithm to other memory models from the model that the algorithm assumes. GC implementers have to modify the copying protocol of the GC algorithm according to the target memory models

until the protocol works correctly, that is, they have to investigate the correctness of the modified protocol multiple times.

To address the problem, we studied the feasibility of model checking of the concurrent copying protocols with weak memory models. In this paper, we report our experience of investigation of a correctness-related property of concurrent copying protocols with various memory models using model checking; the investigated property is described in Section 2. Model checking is a formal verification method in which the program to be checked is translated into a state transition system that models the program. All possible states of the model reachable from the initial state are visited, and whether a given property holds in every state is checked. We used the McSPIN model checker [Abe and Maeda 2014], which checks a program under a given memory model that the user defines in a domain specific language.

To check a wide range of concurrent copying protocols, we constructed common models of the heap and mutators, as described in Section 3. For each concurrent copying algorithm, we constructed its model, combined the model of the protocol with the common models, and conducted model checking using McSPIN, as described in Section 4.

The contributions of this paper include the following:

- We modeled various concurrent copying protocols with fine-grained synchronizations, that is, Stopless [Pizlo et al. 2007b], Clover [Pizlo et al. 2008], Chicken [Pizlo et al. 2008], Staccato [McCloskey et al. 2008], and Sapphire [Hudson and Moss 2001, 2003; Ritson et al. 2014], in the modeling language of McSPIN [Abe and Maeda 2014]. We also modeled concurrent copying protocols with coarse-grained synchronizations (i.e., locking) or no synchronizations (using fragmented allocation) that appear in the GC handbook [Jones et al. 2012], that is, the concurrent version of the Baker-style GC [Halstead 1985], replication GC [Nettles and O'Toole 1993; Nettles et al. 1992], Compressor [Kermany and Petrank 2006], Pauseless [Azul 2008], and Schism [Pizlo et al. 2010].
- We investigated a correctness-related behavioral property of concurrent copying protocols against six memory models, total store ordering (TSO), partial store ordering (PSO), relaxed memory ordering (RMO), and their variants, in addition to sequential consistency (SC) using bounded model checking.
- We found that Stopless did not exhibit errors under some memory models that were weaker than x86 CPUs if a CAS instruction provided the full memory barrier effect. Furthermore, we modified the concurrent copying protocol of Stopless so that it can work with a CAS instruction that does not provide a memory barrier effect.
- We found that Staccato, which supports PSO and RMO, had a fault that prevented it from working under PSO or RMO. Although this fault could be caught if a memory model expert carefully reviewed the code, no one has reported it to the best of our knowledge, and we did not noticed before the model checker reported it. We modified the algorithm and fixed the fault.
- We also found that Clover did not work correctly, even with SC, because a corner case was missed.

2 PROPERTY TO BE VERIFIED

In this paper, we verify a correctness-related property that is defined as follows:

> Under a single collector and a single mutator, the value a mutator reads from an object field is
> (1) the most recently written value, if the field has been written, or otherwise
> (2) the initial value.

The reader might question whether the property is obviously true because it only considers one mutator, and no other mutator exists that may write a value to the slot.

However, this is not the case because not only the mutator but also the collector can access the same slot simultaneously. Unlike traditional concurrent copying GC algorithms, such as Halstead [1985], in which mutators and collectors lock objects during the process of accessing them, modern concurrent copying GC algorithms use fine-grained synchronizations [Hudson and Moss 2001, 2003; McCloskey et al. 2008; Pizlo et al. 2007b, 2008; Ritson et al. 2014], for example, using a CAS instruction, for performance reasons and have the requirement of the lock-freedom of mutators. Their concurrent copying protocols are complicated and thus have uncertain correctness, in particular, under weak memory models. In fact, we found faults while investigating the property, as explained in Section 4.

We focused on one collector and one mutator in this paper because model checking under multiple mutators could not be completed within a reasonable memory and CPU time. In theory, there are no difficulties in processing multiple mutators, but increasing the number of mutators under weak memory models easily causes the state explosion problem. It is worth noting that the faults found in this study, explained in Section 4, are independent of the number of mutators, although there may exist unfound faults that occur only with two or more mutators.

To be able to process multiple mutators, it may be useful to impose additional assumptions/restrictions on mutators to mitigate the state explosion problem. For example, it is known that, if programs under TSO are data race free, they behave in the same manner as they would if they were run under SC [Owens 2010; Saraswat et al. 2007]. This cannot be directly applied to our scenario because there is a race between the mutator/collector, but it suggests that if multiple mutators are data race free with respect to each other, it may be possible to reduce the state space to be explored during model checking under TSO. However, we do not formally explore the possibilities because they are beyond the scope of this paper.

3 METHODOLOGY

We investigated the property described in Section 2 using bounded model checking with McSPIN [Abe and Maeda 2014]. McSPIN receives a definition of the memory model, in addition to a model of the program to be checked, and checks all admissible execution traces in the memory model. More specifically, we defined six memory models, and constructed models for the heap and mutators as a common framework to check the wide range of concurrent copying protocols with McSPIN.

In this section, we describe the memory models and the common framework. The missing parts, that is, the models for concurrent copying protocols, are constructed in Section 4.

3.1 Memory Models

We defined six memory models for McSPIN and conducted model checking with them. The memory models are listed in Table 1. They are combinations of the allowed orderings of memory accesses and the memory barrier effect provided by the CAS instruction.

We defined the following four commonly acknowledged orderings [Adve and Gharachorloo 1996]:

SC disallows any reordering.
TSO allows write instructions to be reordered with subsequent read instructions.
PSO allows reordering among write instructions in addition to TSO.
RMO allows the reordering of any pair of read and write instructions.

Table 1. Memory models used for model checking.

Model	Reordering	CAS
SC	SC	
TSO	TSO	
PSO-full	PSO	full memory barrier
PSO-no	PSO	no memory barrier
RMO-full	RMO	full memory barrier
RMO-no	RMO	no memory barrier

The memory model of the x86 architecture [Intel 2016] is a variant of TSO. The memory models of PowerPC [Sarkar et al. 2011] and ARM [ARM 2016] are much more relaxed than our RMO. Although the memory models of PowerPC and ARM and our RMO share the property that the reordering of any pair of read and write instructions is allowed, there are two major differences: PowerPC and ARM support speculative execution whereas our RMO does not execute speculatively beyond conditional branches, and PowerPC and some versions of ARM [Flur et al. 2016] do not guarantee multi-copy-atomicity. In fact, we defined a memory model with speculative execution and conducted model checking with the model, but it was necessary to explore too large a state space to complete model checking with stock hardware. More specifically, simply allowing speculative execution on a particular conditional branch required more than 1 TB of memory for model checking [Abe et al. 2016]. Regarding multi-copy-atomicity, this difference did not affect the results of our model checking because our models had only two threads: a collector and mutator.

We defined two variations of CAS instructions:

- full memory barrier effect: complete all read or write instructions that appear preceding a CAS instruction in the program before executing the CAS instruction, and complete the execution of the CAS before executing its following read and write instructions; and
- no memory barrier effect.

Note that all memory operations included in the CAS instruction are assumed to be executed atomically. Additionally, note that the CAS instruction with no memory barrier effect still prohibits the reordering of memory access instructions that appear preceding the CAS and with those that follow the CAS if the target addresses of the memory access instructions are the same as the target of the CAS.

The combinations of SC and both variants of CAS are the same memory model because SC does not allow any reordering. The combinations of TSO and both variants of CAS are also the same memory model; hence, they yield the same execution traces. The CAS executes a read operation (R_{CAS}) followed by a write operation (W_{CAS}) atomically. As a requirement of TSO, an execution of any memory access instruction must complete before starting the W_{CAS} operation; hence, it must complete before executing the CAS. Similarly, any memory access instruction cannot be executed until the R_{CAS} operation completes; hence, it cannot be executed until the CAS completes.

The `cmpxchg` instruction of the x86 memory model is a counterpart of the CAS. PowerPC does not have a counterpart, but the manual [IBM 2015] shows an example of the CAS routine implemented using load-linked/store-conditional (LL/SC) instructions. This routine prohibits the execution of subsequent write instructions before the execution of the CAS routine.

3.2 Common Framework

To avoid the trouble of constructing an entire model from scratch for each concurrent copying protocol, we define a common framework that consists of common models of the heap (and the

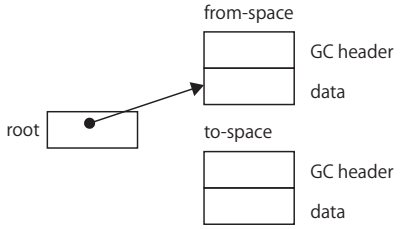

Fig. 1. Variable and objects of our model.

root set) and the mutators that are independent of differences among concurrent copying protocols. By simply defining the models for concurrent copying protocols and combining them with the common framework, we can conduct the model checking of the property mentioned in Section 2 with various concurrent copying protocols. More specifically, the models of concurrent copying protocols have to provide one collector thread and three mutator procedures. The task of the collector thread is to copy the object and update the local variable so that it points to the copy of the object. In the common framework, there exists a single object and single local variable, as we describe below. Please note that updating the local variable is not a task to be performed in the copy phase but in the flip phase. It is necessary, however, to let the mutator access the copy to check that the object has been copied properly.

The mutator procedures are as follows:

- `read(o,v)`: the read barrier procedure, which reads the value from object `o`. The read value is returned to `v`.
- `write(o,v)`: the write barrier procedure, which writes value `v` to object `o`.
- `yieldpoint()`: the yieldpoint, which cooperates with the collector in various ways. The concurrent copying protocols can expect that the mutator calls the `yieldpoint` sufficiently frequently.

Because mitigating the state explosion is a challenge in model checking, we made the common models of the common framework as small as possible. Our approach was to focus sharply on checking whether the values written by a mutator were ever lost in a manner that violated SC.

3.2.1 Heap and Root Set. Figure 1 shows the variable and objects in our common framework. The model has only a single local variable of the mutator (`root`). There are two pre-allocated copies of an object in the heap; one for the from-space object and the other for the to-space copy; for the Stopless algorithm, we needed to introduce another copy, but we will not elaborate on it here. Note that the mutator does not recognize that the object may have multiple copies. These copies are semantically (i.e., from the viewpoint of the mutator) a single object; hence, the mutator expects that it reads the value that it has written even if the collector switches the copy that is responsible for storing the up-to-date value.

The object has a single payload slot (`data`) and a header (`GC header`) for the garbage collector. The mutator writes zero or one to the payload slot. In the remainder of this subsection, we explain the features of this mutator model and the reasons why we used it.

First, the mutator model considers a static object graph. Because our interest is in the copy phase, and we assume that the set of objects to be copied has been determined in a previous phase, we did not consider the mutation of the object graph by the mutator. Furthermore, we did not model a memory allocator; instead, we assumed that the to-space copy had been allocated in some manner.

Second, the mutator model has a single object that has a single payload slot. Because we concentrated on cases of a single mutator, as mentioned in Section 2, we did not need to perform

verification at multiple locations unless a collector processes multiple memory locations in a non-uniform manner (e.g., when processing two memory locations, a collector performs a CAS on one memory location and, if the CAS succeeds, does not perform another CAS on another memory location). It is worth noting that all the collectors addressed in this paper process multiple memory locations uniformly. A possibly unacceptable behavior of the mutator relevant to multiple locations in this situation is that two writes to different locations are observed to be committed in reverse order. However, if the mutator can always read the latest value written to each slot, which is a requirement of SC, such unacceptable behavior never occurs.

Third, we assumed that there was a single collector. If we verified with multiple collectors, model checking would require much more time and memory.

3.2.2 Mutator. We restricted the mutator's behavior so as to mitigate the state explosion. Ideally, a mutator model that covers any mutator behavior has an infinite loop in which the mutator nondeterministically reads, writes, or calls the `yieldpoint`. However, it requires an exploration in infinite space. McSPIN unrolls every loop so that it can process reordering between memory access operations in different iterations. Even if there is no infinite loop, the longer the sequence of mutator's operations, the more time and memory model checking consumes.

Therefore, we defined two reasonable behavior scenarios for the mutator. Each of the two scenarios corresponds to items (1) and (2) of the property that we are investigating, as listed in Section 2:

WR The mutator writes a value different from the initial value to the slot and then it reads from the slot. In this scenario, we expect the mutator to read the value it has written. If the collector wrongly overwrites the mutator's value, or the mutator writes to the from-space object after the collector copies it, the initial value is read.

RW The mutator reads from a slot and then it writes a value different from the initial value to the slot. In this scenario, we expect the mutator to read the initial value.

The read and write operations are performed by calling the `read` and `write` procedures that the model for the concurrent copying protocol provides.

Some of the concurrent copying protocols that we verified required mutators that could answer a soft handshake for ragged synchronizations [Bacon et al. 2001]. Staccato requires two soft handshakes to complete copying the object. Therefore, we included two yieldpoints where the mutator could answer the handshake: at the head of each scenario and between the reads and writes.

The mutator models are like those shown in Figure 2, which is for the **WR** scenario. The `read(root, readval)` procedure reads a slot of the object referred to by `root` and writes the read value in the local variable `readval`. `ASSERT` checks whether the read value is one, which is the value that the preceding `write` has written. It is worth noting that the `ASSERT` macro has a full memory barrier effect; thus, the read instruction completes before the assertion is tested. Thus, `ASSERT` may restrict the reordering of memory access instructions allowed by the memory model. To check all possible reordering, we added the "`McSPIN nondeterministic`" annotation to assertion macros if they were between memory access instructions so that the model checker could check the execution paths in which the assertion macros were not executed.

3.3 Limitations

The common models of the common framework have several limitations:

- We investigated one of the correctness-related properties.
- The object graph in our model is tiny and fixed. Hence, memory allocation is out of scope.
- We consider the case of a single mutator and a single collector.

```
#define INITIAL_VALUE 0
#define UNINTIALISED 2

void* mutator(void* arg) {
  int readval;
  yieldpoint();
  yieldpoint();
  write(root, 1);
  yieldpoint();
  yieldpoint();
  read(root, readval);
  ASSERT(1 == McSPIN_variable(readval,0,0));
}
```

Fig. 2. Mutator model for **WR** scenario.

- The mutator behaves in accordance with one of two fixed scenarios.
- McSPIN is a bounded model checker; thus, it only checks execution traces that have at most a given number of backward jumps; McSPIN executes the body of each while-loop once without spending the backward jump budget. Because of memory limitations, we set the number to zero. Thus, we did not verify execution traces that iterated do-while loop multiple times.

Because of these limitations, even if the models of the concurrent copying protocols have errors, our model checking may fail to detect the errors.

Nevertheless, the results of our model checking are useful from both theoretical and practical perspectives. If the result was negative, that is, if our model checking determined a counterexample when investigating a property of a concurrent copying protocol, it is formally ensured that the algorithm definitely did not satisfy the property under the given memory model provided the model for the concurrent copying protocol was faithful. In fact, our model checking actually found real faults in two concurrent copying protocols, as explained in Section 4, and we were able to fix the faults. As mentioned above, we were not able to formally ensure that there were no other faults in the fixed algorithms, but we greatly increased confidence in the algorithms.

4 INVESTIGATING ALGORITHMS AND THEIR RESULTS

In this section, we report on our case study. We investigated the property mentioned in Section 2 of concurrent copying protocols of various concurrent copying GC algorithms. In the GC handbook [Jones et al. 2012] Section 19.7, Stopless, Staccato, Chicken, and Clover are introduced as real-time concurrent GC algorithms that move objects. We investigated the property of their concurrent copying protocols, in addition to that of Sapphire, which appears in Section 17.6 of the GC handbook. Although Sapphire is not a real-time GC, its concurrent copying protocol uses fine-grained synchronizations. In addition to the concurrent copying protocols with fine-grained synchronizations, we investigated the property of those with coarse-grained synchronizations. We used the following environment for model checking: The CPU was Core i7-6700K 4.00 GHz, memory was DDR4 32 GB with 300 GB swap on an Intel SSD 750 Series, and OS was Ubuntu 14.04.5. McSPIN used SPIN version 6.4.6 and GCC version 4.8.4. The largest model consumed 109 GB of memory to complete model checking.

Because we modeled all the concurrent copying protocols in a similar manner, we describe the details of modeling only in Section 4.1.

ALGORITHM 1: Copy method and barriers of Stopless.

```
/* collector's copying method: copy object o */
copy(o):
  wide ← AllocateWideObject(sizeof(o));
  CAS(&o.forward, o, wide);
  wide ← o.forward;
  foreach (f in fields of o):
    ⟨v,s⟩ ← wide[f];
    CASW(&wide[f],
         ⟨v,IN_ORIGINAL⟩, ⟨o[f],IN_WIDE⟩);
  copy ← AllocateToSpace(sizeof(o));
  wide.forward ← copy;
  foreach(f in fields of wide):
    do:
      ⟨v,s⟩ ← wide[f];
      copy[f] ← v;
    while (not CASW(&wide[f],
                    ⟨v,s⟩, ⟨v,IN_COPY⟩));
  o.forward ← copy;
```

```
/* read barrier: read from field f of object o */
read(o, f):
  oo ← o.forward;
  if (oo is wide): /* test low order bits of oo */
    ⟨v,s⟩ = oo[f];
    if (s = IN_ORIGINAL):
      return o[f];
    else if (s = IN_WIDE):
      return oo[f];
    else if (s = IN_COPY):
      return oo.forward[f];
  else:
    return oo[f];
/* write barrier: write value v to field f of object o */
write(o, f, v):
  if (o.forward = o):
    wide ← AllocateWideObject(sizeof(o));
    CAS(&o.forward, o, wide);
  oo ← o.forward;
  if (oo is wide): /* test low order bits of oo */
    do:
      ⟨u,s⟩ ← oo[f];
      if (s = IN_COPY):
        oo.forward[f] ← v;
        return;
    while (not CASW(&oo[f], ⟨u,s⟩, ⟨v,IN_WIDE⟩));
  else:
    oo[f] ← v;
```

4.1 Stopless

Stopless [Pizlo et al. 2007a] is a concurrent compacting garbage collector that offers lock-freedom of mutators. Although the memory models this collector can run under were not specified, the experimental environment indicated that it supports the x86 memory model.

4.1.1 Algorithm. The main idea of Stopless is that, when copying an object in the from-space to the to-space, it first copies the object to an intermediate object called a *wide* version, and then copies the wide version to the new copy in the to-space. It introduces the wide version because it enables asynchronous accesses by mutators and collectors. More specifically, each field of the wide version is associated with its status value, which indicates which copy of the object to access when accessing the field, that is, the original object, wide version, or new copy.

Because pseudocode was not provided in Pizlo et al. [2007a], we wrote pseudocode to clarify the algorithm to be verified. Our pseudocode for Stopless is shown in Algorithm 1, where `copy` is the procedure used by the collector to copy object `o`, and `o[f]` is the field at offset `f` in object `o`.

First, the collector allocates a wide version (`wide`) and installs the forwarding pointer. As we describe below, the mutator may also allocate a wide version. The forwarding pointer is installed

```
1  /* object */
2  #define FROM_OBJECT 0
3  #define WIDE_OBJECT 1
4  #define TO_OBJECT 2
5  /* heap */
6  int from_fwd = FROM_OBJECT;
7  int from_data = INITIAL_VALUE;
8  int wide_fwd = FROM_OBJECT;
9  int wide_data = UNINITIALISED;
10 int to_fwd = FROM_OBJECT;
11 int to_data = UNINITIALISED;
```

```
12 /* status */
13 #define IN_ORIGINAL 0
14 #define IN_WIDE 1
15 #define IN_COPY 2
16 /* mutator's variable */
17 int root = FROM_OBJECT;
```

Fig. 3. Heap model for Stopless.

using the CAS to resolve the data race. Then, the collector fills the wide version by copying from `o`. Again, the wide version field is updated using the double-word CAS instruction (`CASW`) to resolve the race with the mutator writing to the same field. Throughout this paper, `CAS` and `CASW` in algorithms represent the CAS instructions that atomically write the third argument to the location indicated by the first argument and return true if the value of the location is the same as that of the second argument, or return false otherwise. Note that we also use `CAS` to represent the CAS instruction in models, that is, in figures, but it returns whether it succeeded through the fourth argument.

Once all fields of the wide version are filled, the collector allocates the to-space copy and installs the pointer pointing at the to-space copy in the forwarding pointer field of the wide version. Then, the collector copies each field to the to-space copy. After copying each field, it updates the status word to `IN_COPY` using `CASW` that asserts that the value in the payload field is unchanged. When all the payload fields have been copied, the collector updates the forwarding pointer field of the from-space object with the pointer to the to-space copy. Thus, the mutator can access the to-space copy without accessing the wide version.

The read barrier procedure (`read`) activated during the copy phase reads the value in field `f` of object `o`. To read from the up-to-date location of the field, it first reads the forwarding pointer, and, if the forwarding pointer points to the wide version, it further reads the status word.

The write barrier procedure (`write`) activated during the copy phase writes value `v` to field `f` of object `o`. The write barrier pushes the collection ahead; it creates a wide version for the object on behalf of the collector if the collector has not yet done so. Thus, the forwarding pointer is installed using the CAS. Regardless of whether the up-to-date location of the field is in the from-space object or wide version, the write barrier writes to the wide version and updates the status word to `IN_WIDE` simultaneously. If the up-to-date location is in the to-space object, it writes to the to-space object with a plain write.

4.1.2 Models. In this subsection, we explain the models that we used for checking Stopless.

Figure 3 shows the heap model for Stopless. Each copy of the object is identified by an integer, which models the memory address of the copy. Because Stopless creates a temporary wide version of the object being copied, we used three object identifiers, 0, 1, and 2, for `FROM_OBJECT`, `WIDE_OBJECT`, and `TO_OBJECT`, respectively. Each copy of the object has a GC header, which includes a forwarding pointer, and payload field. The forwarding pointer fields of the from-space, wide version, and to-space objects are modeled by `from_fwd`, `wide_fwd`, and `to_fwd`, respectively. The initial value of `from_fwd` is the from-space object itself as it is in Stopless. Because the wide version and the to-space copy are not allocated at the beginning of the copy phase in reality, we used `FROM_OBJECT`

```
#define WIDE(v,s) (((s) << 2) | (v))
#define WIDE_VALUE(x) ((x) & 3)
#define WIDE_STATUS(x) ((x) >> 2)

void* collector(void* arg) {
  int x, y, success;
  CAS_NORET(from_fwd, FROM_OBJECT,
            WIDE_OBJECT);
  x = WIDE(WIDE_VALUE(wide_data),
           IN_ORIGINAL);
  y = WIDE(from_data, IN_WIDE);
  CAS_NORET(wide_data, x, y);
  wide_fwd = TO_OBJECT;
  /* McSPIN_fence(); */
  to_fwd = TO_OBJECT;
  success = FALSE;
  while (success == FALSE) {
    x = wide_data;
    y = WIDE_VALUE(x);
    to_data = y;
    /* McSPIN_fence(); */
    CAS(wide_data, x, WIDE(y, IN_COPY),
        success);
  }
  from_fwd = TO_OBJECT;
  /* McSPIN_fence() */
  root = from_fwd; /* flip */
}
```

Fig. 4. Collector model for Stopless.

```
static inline read(int obj,int retval){
  int x, l_from_fwd, wide_status;
  if (obj == FROM_OBJECT) {
    l_from_fwd = from_fwd;
    if (l_from_fwd == FROM_OBJECT)
      retval = from_data;
    else if(l_from_fwd == WIDE_OBJECT){
      x = wide_data;
      wide_status = WIDE_STATUS(x);
      if (wide_status == IN_ORIGINAL)
        retval = from_data;
      else if (wide_status == IN_WIDE)
        retval = WIDE_VALUE(x);
      else if (wide_status == IN_COPY){
        ASSERT_EQ(wide_fwd, TO_OBJECT);
        retval = to_data;
      }
    } else if (l_from_fwd == TO_OBJECT)
      retval = to_data;
  } else {
    ASSERT_EQ(to_fwd, TO_OBJECT);
    retval = to_data;
  }
}
```

```
static inline write(int obj, int val) {
  int x, success;
  if (obj == FROM_OBJECT) {
    CAS_NORET(from_fwd, FROM_OBJECT,
              WIDE_OBJECT);
    if (from_fwd == WIDE_OBJECT) {
      success = FALSE;
      while(success == FALSE) {
        x = wide_data;
        if (WIDE_STATUS(x) == IN_COPY) {
          ASSERT_EQ(wide_fwd,TO_OBJECT);
          to_data = val;
          success = TRUE;
        } else/* IN_ORIGINAL or IN_WIDE */
        CAS(wide_data, x,
            WIDE(val, IN_WIDE), success);
      }
    } else /* from_fwd == TO_OBJECT */
      to_data = val;
  } else {
    ASSERT_EQ(to_fwd, TO_OBJECT);
    to_data = val;
  }
}
```

Fig. 5. Read and write barrier models for Stopless.

as the initial values of `wide_fwd` and `to_fwd` so that we could detect an initialization failure for these fields. Both fields should be initialized with the pointers to the to-space copy when the to-space copy is allocated. Note that memory allocation is not considered in the model. We made the mutator's private variable (`root`) a shared variable so that the collector could flip it.

Figure 4 shows the collector model for Stopless. The `collector` function copies the object modeled using `from_fwd` and `from_data`. Because the model does not consider memory allocation, the first operation is the installation of the forwarding pointer using `CAS_NORET`, which is a CAS model used when it does not matter whether the CAS succeeds. Then, the value in the slot of the from-space object is copied to the wide version using the double-word CAS instruction. To express this instruction, we combined the payload and status word into a single integer using the `WIDE` macro. Once the model has copied the value in the slot of the from-space object, or observed the failure of the double-word CAS indicating that the mutator has written to the wide version, it allocates the to-space copy and installs a pointer pointing at the to-space copy into the forwarding pointer fields of the wide version and the to-space copy. Again, as our model does not consider memory allocation, it merely writes `TO_OBJECT` to `wide_fwd` and `to_fwd`. It then copies the field of the wide version to the to-space copy and completes copying by updating the status word of the wide version. Because the mutator may update the payload field, this copy may need to be retried. Once the up-to-date locations of all slots (although the object has only a single slot) have been moved to the to-space copy, the model updates the forwarding pointer field of the from-space object with the pointer to the to-space copy. The copy phase then finishes, and the collector model flips the mutator's root (`root`).

Figure 5 shows the read and write barrier models for Stopless. The `read` and `write` inline functions are models of the read and write barriers. Because Stopless does not need a soft handshake, the model of the `yieldpoint` is an empty function, which Figure 5 does not show.

We did not use a direct approach, for example, the array lookup, to model indirect memory access, such as reading from the forwarding pointer field of the object pointed to from `obj`. Instead, we dispatched a routine that read from the appropriate scalar variable. This is because the calculation of array indices expands the state space to be explored.

The `read` function reads the value from the slot of object `obj` and returns the value through the argument `retval`, which is a call-by-name argument. First, the read barrier reads the forwarding pointer from the header of the object to which `obj` points. If `obj` is `FROM_OBJECT`, then the read barrier reads the forwarding pointer from `from_fwd`. Otherwise, we assert that `to_fwd` is `TO_OBJECT` using `ASSERT_EQ` because `obj` should be `TO_OBJECT` and the forwarding pointer of the to-space copy should have been initialized to point to the copy itself. The modeling of the remainder of the read barrier is straightforward. The modeling of the write barrier is also straightforward. Note that we omitted guard testing the forwarding pointer field of the from-space object preceding the `CAS` that installs the forwarding pointer because this test is performed in the `CAS` anyway.

4.1.3 Results and Findings. The row labeled "Stopless" in Table 2 shows the results. A checkmark indicates that verification succeeded, that is, the mutator successfully read the expected value and no assertions in the read and write barriers failed.

TSO with the CAS providing the full memory barrier effect showed no error. Given that Stopless was evaluated on an x86 system [Pizlo et al. 2007a], this is not surprising. Unexpectedly, model checking with weaker memory models also succeeded when the CAS provided the full memory barrier effect.

We examined a counterexample, that is, an execution trace that violated an assertion. In an execution of **WR** with the PSO-no memory model, the assertion in the read barrier on line 21 in Figure 5 failed. This was because the collector's initialization of `to_fwd` on line 15 in Figure 4 was delayed until after updating `root` on line 27. This counterexample modeled the failure that delays the installion of the forwarding pointer until the flip phase. This failure is unlikely to occur in real implementations because GC implementers usually implement some synchronizations between

Table 2. Results of model checking of all the GC algorithms that we investigated: checkmark indicates that both scenarios showed no errors; dash indicates we did not conduct model checking because the concurrent copying protocol did not use the CAS.

GC	Sect.	SC	TSO	PSO-full	PSO-no	RMO-full	RMO-no
Stopless	4.1	✓	✓	✓	both fail	✓	both fail
Stopless(for RMO)	4.1	✓	✓	✓	✓	✓	✓
Chicken	4.2	✓	✓	**wr** fail	both fail	**wr** fail	both fail
Staccato	4.2	✓	✓	**wr** fail	**wr** fail	**wr** fail	**wr** fail
Staccato(fix with `fence`)	4.2	✓	✓	✓	✓	✓	✓
Staccato(fix with `rel_fence`)	4.2	✓	✓	✓	✓	✓	✓
Clover($\alpha = 0$)	4.3	**wr** fail	**wr** fail	**wr** fail	**wr** fail	**wr** fail	**wr** fail
Clover($\alpha = 1$)	4.3	✓	✓	✓	both fail	✓	both fail
Clover($\alpha = 2$)	4.3	✓	✓	✓	both fail	✓	both fail
Clover(fix,$\alpha = 0$)	4.3	✓	✓	✓	✓	✓	✓
Clover(fix,$\alpha = 1$)	4.3	✓	✓	✓	both fail	✓	both fail
Clover(fix,$\alpha = 2$)	4.3	✓	✓	✓	both fail	✓	both fail
Clover(for RMO, $\alpha = 0$)	4.3	✓	✓	✓	✓	✓	✓
Clover(for RMO, $\alpha = 1$)	4.3	✓	✓	✓	✓	✓	✓
Clover(for RMO, $\alpha = 2$)	4.3	✓	✓	✓	✓	✓	✓
Sapphire (iteration part)	4.4	✓	✓	✓	—	✓	—
Sapphire (LL/SC part)	4.4	✓	✓	✓	—	✓	—
x86-Sapphire	4.4	✓	✓	**wr** fail	—	**wr** fail	—
Baker	4.5	✓	✓	✓	—	✓	—
Replication GC	4.5	✓	✓	✓	—	✓	—
Compressor	4.5	✓	✓	✓	✓	✓	✓
Pauseless	4.5	✓	✓	✓	—	✓	—
Schism	4.5	✓	✓	✓	—	✓	—

phases. We inserted a full memory barrier before flipping, that is, uncommented `McSPIN_fence()` on line 26 in Figure 4.

We still found a similar error after inserting this memory barrier: the collector's write to `wide_fwd` on line 13 in Figure 4 was delayed until after updating `from_fwd` on line 25, and assertions on `wide_fwd` in the read and write barriers in Figure 5 failed. This failure indicates that a memory barrier is required after installing a forwarding pointer. We modified the model so that the collector performed a full memory barrier after installing forwarding pointers, that is, uncommented `McSPIN_fence()` on line 14 in Figure 4.

Again, we found another counterexample with a subtle failure: the mutator read the initial value (zero) after writing one. This happened in the following execution.

(1) The collector copied the initial value (zero) from the from-space object to the wide version.
(2) The collector started copying from the wide version to the to-space copy; it read zero from `wide_data` and attempted to write it to `to_data` on line 20 in Figure 4. However, the completion of the write was delayed.
(3) The collector updated the status word in the wide version with `IN_COPY` on line 22.

(4) The mutator wrote one to the slot. Because the status word was `IN_COPY`, which indicated that the up-to-date location was in the to-space copy, the mutator wrote one to `to_data` on line 36 in Figure 5.
(5) The collector's write in Step 2 was completed. This wrote over `to_data` with zero, and the mutator's write was lost.
(6) Finally, the mutator read the stale value zero from the to-space copy.

We inserted another full memory barrier after the collector wrote to `to_data`, that is, uncommented `McSPIN_fence()` on line 21 in Figure 4. Model checking after this modification with all memory models for both scenarios revealed no errors, as shown in the row labeled "Stopless (for RMO)" in Table 2.

The full memory barriers we added correspond to the following timings in Algorithm 1:

(1) after the collector installs a forwarding pointer in the wide version; and
(2) after the collector writes to each field of the to-space copy.

Inserting memory barriers does not violate the lock-freedom of mutators. Furthermore, because these memory barriers were added only in the collector, the mutator's throughput was not affected.

4.2 Chicken and Staccato

Chicken [Pizlo et al. 2008] and Staccato [McCloskey et al. 2008] are concurrent copying algorithms that are based on the same idea: the collector copies each object optimistically, and if a mutator writes to the object during copying, then copying is aborted. These algorithms offer the wait-freedom of mutators. In this paper, we say that GC offers the "wait-freedom of mutators" if the read and write operations of any mutator can make progress.

Chicken was implemented in the Bartok system, which is a CLI compiler and its runtime. Its authors evaluated the performance on an x86 CPU; although they mentioned LL/SC instructions, which are often implemented in weaker memory models, such as PowerPC. Staccato is "suitable for both strongly and weakly ordered multiprocessors" [McCloskey et al. 2008], in particular, it seems, for supporting PowerPC because the authors mentioned the `sync` instruction of PowerPC.

4.2.1 Algorithm. We first explain the simpler algorithm, Chicken, and then we explain the difference between Chicken and Staccato.

The key concept of Chicken is that it introduces a status flag to an object that indicates that the object is being copied, but the copy is not complete. More specifically, a collector sets the flag when copying starts, and a mutator checks the flag when writing to the object. If the flag is not set, the write is performed on the from-space object or the to-space object, according to the forwarding pointer. If the flag is set, the copy is aborted.

Figure 6 shows the state transition of an object in Chicken. At first, the object is in the `IDLE` state. In this state, the up-to-date locations of all fields of the object are in the from-space object. When the collector starts copying, it proceeds to the `COPYING` state. After the collector has copied all fields, it proceeds to the `COPIED` state; from that time, the up-to-date locations of all fields of the object are in the to-space copy. If the object is in the `COPYING` state, the mutator may not write to the object; the mutator has to move the object's state back to the `IDLE` state before writing; Staccato disallows reading from `COPYING` state objects, in addition to writing. The transition from the `COPYING` state is performed atomically.

Algorithm 2 shows the pseudocode for Chicken. The role of each procedure is the same as the corresponding role in Stopless. The lines marked # are from Staccato. They do not appear in Chicken. We simplified the pseudocode so that the collector copies a single object for simplicity, as was done previously [Pizlo et al. 2008].

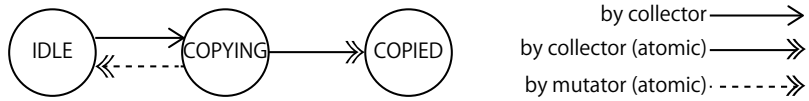

Fig. 6. Object state transition in Chicken.

ALGORITHM 2: Copying method and barriers of Chicken.

```
copy(o):
  /* prepare */
  o.forward ← o | COPYING;
  /* sync */
  SoftHandshake();
# ReadSync();
  /* copy */
  oo ← AllocateToSpace(sizeof(o));
  CopyBytes(o, oo, sizeof(o));
  oo.forward ← oo;
  /* sync */
# WriteSync();
# SoftHandshake();
  /* commit */
  CAS(&o.forward, o | COPYING, oo);
```

```
read(o, f):
  return (o.forward & ~COPYING)[f]

write(o, f, v):
  if ((o.forward & COPYING) ≠ 0):
    CAS(o.forward, o | COPYING, o);
  (o.forward & ~COPYING)[f] = v;
```

First, in the preparation step, the collector asserts that it is about to copy the object by setting the `COPYING` flag in a spare bit of the forwarding pointer. The collector then initiates `SoftHandshake` with each mutator. The mutators answer at yieldpoints, which are outside the write barrier. This ensures that all write barriers that observe the `COPYING` flag as being unset complete. In the copy step, the collector allocates the to-space copy and copies the object as a whole. If a mutator attempts to write to the object during copying, the mutator resets the `COPYING` flag before writing to the object. If the collector completes copying before any mutator resets the `COPYING` flag, it commits the copy by simultaneously clearing the `COPYING` flag and installing the pointer to the to-space copy into the forwarding pointer field.

Thus, the collector committing the copy and a mutator writing to the object may race. Chicken resolves this race using the CAS. If the collector loses the race, the collector does not copy the object until the next GC cycle, that is, an attempt to copy the object is performed only once in a single GC cycle.

Staccato is similar to Chicken in the sense that it uses a copy status flag in the same manner as Chicken, but the key difference is that it performs additional synchronizations to manage weaker memory models than Chicken, as mentioned above. All of the differences are listed as follows:

- The collector has extra synchronizations, which are indicated by the # mark in Algorithm 2.
- The collector sets the `COPYING` flag using the CAS; this is to resolve the race between collectors because Staccato supports parallel collectors.
- Not only the write barrier but also the read barrier resets the `COPYING` flag. The read and write barriers determine the up-to-date version of the target object by calling `access`, as shown in Algorithm 3. They then read or write the up-to-date version of the object using a plain read or write.

ALGORITHM 3: Access barrier of Staccato.

```
/* return the up-to-date version of object o */
access(o):
  forward ← o.forward;
  if ((forward & COPYING) = 0):
    return forward;
  if (CAS(&o.forward, forward, o))
    return o;
  return o.forward; /* atomic read */
```

```
1  /* executed by mutator */
2  static inline yieldpoint() {
3    if (hs_req == TRUE) {
4      McSPIN_acq_fence(); /* should be full fence */
5      hs_req = FALSE;
6    }
7  }
8
9  /* executed by collector */
10 static inline SoftHandshake() {
11   hs_req = TRUE;
12   while (hs_req == TRUE) ;
13 }
```

Fig. 7. Model of a soft handshake.

- The last read in access is an atomic read that reads the current value written using the CAS.
- The soft handshake requires the mutator to perform a memory read synchronization.

4.2.2 Models. Most parts of the models for the collector and the read and write barriers are constructed straightforwardly in a similar manner to those for Stopless. The complete models are shown in the appendix. We focus on the parts that are unlike those for Stopless.

We used the McSPIN_acq_fence and McSPIN_rel_fence constructs for modeling ReadSync and WriteSync in Algorithm 2. McSPIN_acq_fence prohibits reordering between preceding read instructions and following memory access instructions, and McSPIN_rel_fence prohibits reordering between preceding memory access instructions and following write instructions. For the atomic read in the access barrier, we used a read surrounded by special memory barriers that prevent the read from being reordered with other memory access instructions whose targets are the forwarding pointer. Without these barriers, the atomic read can be reordered with other reads from the forwarding pointer under RMO, although there are no such reads in the model for Staccato because of control dependency and CAS.

Figure 7 shows the model for the soft handshake. The collector initiates a soft handshake by setting shared variable hs_req to true. At each yield point, the mutator polls the variable. If it is true, the mutator performs a memory read synchronization (McSPIN_acq_fence) and answers by resetting hs_req.

4.2.3 Results and Findings. As expected, the results for Chicken were negative for memory models weaker than TSO, as shown in the row labeled "Chicken" in Table 2. The resulting counterexamples were, however, different from those that we expected. The counterexamples indicated that a memory barrier after the installation of the forwarding pointer in the to-space copy was needed as the case of Stopless with weak memory models. We resolved this problem by inserting a full memory barrier and conducted model checking for the modified Chicken model. Then, the modified Chicken model still exhibited errors, as we expected. The resulting counterexamples indicated that the mutator read an unexpected value. This indicates that the ordering among stores is mandatory.

Surprisingly, Staccato also showed errors as a result of model checking with memory models weaker than TSO, as shown in the row labeled "Staccato" in Table 2.

We examined a counterexample for the **WR** scenario under the PSO-full memory model. The mutator read the initial value zero from the to-space copy after the following steps:

(1) The mutator started the execution of the write barrier and confirmed that the `COPYING` flag had not been set.
(2) The collector set the `COPYING` flag and started the soft handshake.
(3) The mutator *attempted to* write one to the from-space object, but this write was delayed.
(4) The mutator answered the soft handshake while the write in Step 3 was pending.
(5) The collector copied from the from-space object to the to-space copy. The copied value was the initial value.
(6) The write in Step 3 finally completed and one was written to the *from-space* object.
(7) The collector installed the forwarding pointer into the to-space copy.

The mutator's write should have been completed before the collector's copy.

We could insert a memory barrier in the write barrier so that mutator writes are executed immediately. Performing memory barriers frequently, however, imposes a large performance penalty. We thus modified our model so that the mutator performed `McSPIN_rel_fence` when the mutator answered the soft handshake. At that point, the model already had `McSPIN_acq_fence` (line 4 in Figure 7). Thus, we replaced it with a full memory barrier, as indicated by the comment in Figure 7. As a result, model checking with PSO and RMO did not reveal errors, as shown in the row labeled "Staccato (fix with `fence`)" in Table 2.

We tried to weaken this fence. We replaced `McSPIN_acq_fence` with `McSPIN_rel_fence`, and Staccato still revealed no error, as shown in the row labeled "Staccato (fix with `rel_fence`)" in Table 2.

Note that McCloskey et al. [2008] is unclear about the fence required at the point of the error. Although McCloskey et al. [2008] states that "all mutators will have performed a memory read synchronization" at the second step in Section 3.2.1, their implementation seems to use a full fence. The paper also states the followings: "On weakly ordered machines, the ragged barriers must perform memory-fence operations (e.g. `sync` on the PowerPC)." The GC handbook [Jones et al. 2012] interpreted it as a read fence, which is a reasonable interpretation of McCloskey et al. [2008] if one does not actually look at the resulting code. The present paper reasonably models that as an acquire fence. The model checker showed this to be incorrect. On manual re-examination, this is not universally surprising because the code implied by Jones et al. [2012] does not actually look very plausible to experts in weak memory models.

In addition to the above, there is another unclear point in McCloskey et al. [2008]. McCloskey et al. [2008] states that the collector performs "a memory write synchronization" at the third step in Section 3.2.1, but this is inconsistent with the corresponding pseudocode that uses "`readSync`."

The present paper models it as a memory read synchronization because GC handbook [Jones et al. 2012] interpreted it as a "read fence".

4.3 Clover

Clover [Pizlo et al. 2008] is a concurrent copying GC algorithm that usually provides a wait-freedom of mutators, but it blocks the mutators with very low probability. Although the memory models this collector can run under are not specified, the experimental environment indicates that it supports the x86 memory model.

4.3.1 Algorithm. The basic idea of Clover is to choose a random value α and assume that it seldom occurs that a mutator writes the chosen value. Unless a mutator writes α to an object, a collector can use the value α for notifying a mutator that a field of an object that is being copied has been copied without blocking synchronizations. More specifically, when copying a field of an object in the from-space to the to-space, the collector sets the value α to the field in the from-space after it copies the field. Then, the mutator is able to know whether to follow the forwarding pointer of the object by checking whether the field is equal to α. If the mutator writes α to the object, the mutator blocks until the copy phase completes.

Algorithm 4 shows the pseudocode for Clover. The `copy` procedure copies object `o` to the to-space. This procedure assumes that the memory to which `o` is copied has been allocated, and a pointer to the address has been installed into the forwarding pointer field `o.forward`. This procedure also assumes that all the to-space locations are initialized to α. Pizlo et al. [2008] described the pseudocode for copying a single field. Thus, we added an outer loop so that the `copy` procedure copies an object as the pseudocode for other algorithms shown in this paper.

After the collector copies a field, it writes α to the field of the from-space copy. If the value that a mutator reads is α, the mutator reads the up-to-date value from the to-space copy. To resolve a data race between the collector writing α and a mutator writing a value, both write using the CAS.

There are two corner cases. First, the mutator may write α. In this unfortunate case, the mutator is blocked until the copy phase ends. Second, a from-space object has value α before a GC cycle starts. Fortunately, the collector can merely skip copying this field as shown by the lines marked # in Algorithm 4 because all the fields of the to-space copy are initialized to α. The consideration of the second case is not described in the paper [Pizlo et al. 2008]. As we mentioned before, the authors provided pseudocode for copying a single field. The pseudocode does not contain the lines marked # in Algorithm 4. We could not find any description of managing this case in the text, either. As we describe later, we found this corner case from the counterexample produced by model checking, although this is not relevant to the memory model.

4.3.2 Models. We modeled Clover in a similar manner to other algorithms. Figure 8 shows the collector model and mutator subroutines. The lines marked # in Figure 8 correspond to those in Algorithm 4. The complete models are shown in the appendix.

Because GC implementers can choose a value for α arbitrarily, we constructed three models that used different values of α. Our mutator scenarios consider two values, an initial value (zero) and the values the mutator writes (zero and one). Therefore, we chose 0, 1, and 2 for α for each model.

A subtle part of Clover is that the mutator may block in the write barrier until the copy phase ends. We introduced global variable `phase` to indicate the current phase in the same manner as the original algorithm [Pizlo et al. 2008]. Initially, it indicates the `COPY` phase. When the collector has copied the object, it sets `phase` to `IDLE`, and then performs a soft handshake with the mutator (see Section 4.2 for the details of the soft handshake). When the mutator blocks in `WaitUntilCopyingEnds`, it polls `phase` while calling `yieldpoint`.

ALGORITHM 4: Copy method and barriers of Clover.

```
/* Assume that the memory to which o is copied
has been allocated, and a pointer to the address have
been installed to o.forward.
Additionally, assume that all fields of the copy has been
initialized to a certain value α. */
copy(o):
  foreach (f in fields of o):
#   if (value = α):
#     continue;
    while (true):
      value ← o[f];
      o.forward[f] ← value;
      if (CAS(&o[f], value, α)):
        break
```

```
read(o,f):
  value ← o[f];
  if (value = α):
    return o.forward[f];
  else:
    return value;

write(o,f,v):
  if (value = α):
    WaitUntilCopyingEnds();
  while (true):
    old ← o[f];
    if (old = α):
      o.forward[f] ← v:
      break;
    else if (CAS(&o[f], old, v)):
      break;
```

```
1  int phase = COPY;
2  void* collector(void* arg)
3  {
4    int success = FALSE;
5    while (success == FALSE) {
6      int value;
7      value = from_data;
8  #   if (value != ALPHA) {
9        to_data = value;
10       /* McSPIN_fence(); */
11       CAS(from_data, value, ALPHA,
12           success);
13 #    }
14   }
15   root = TO_OBJECT;
16   phase = IDLE;
17   SoftHandshake();
18 }
```

```
19 /* collector subroutine */
20 static inline SoftHandshake() {
21   hs_req = TRUE;
22   while (hs_req == TRUE) ;
23 }
24 /* mutator subroutine */
25 static inline yieldpoint() {
26   if (hs_req == TRUE)
27     hs_req = FALSE;
28 }
29 static inline WaitUntilCopyingEnds() {
30     while (phase != IDLE)
31       yieldpoint();
32 }
```

Fig. 8. Collector model and models for mutator subroutines for Clover.

4.3.3 Results and Findings. We constructed models and conducted model checking. As a result, we found an error in the model whose α was the initial value (zero) as shown in the rows labeled "Clover" in Table 2. This error occurred even with SC. It is worth noting that, in our modeling of Clover, zero does not model the value that a newly allocated object has but a value that an object has in a field by chance before GC. We fixed the model, as mentioned in Section 4.3.1.

We again conducted model checking. As we expected, the fixed model showed errors during model checking with PSO and RMO if the CAS did not provide the memory barrier effect, as shown

in the rows labeled "Clover(fix)" in Table 2. Note that, when α was zero, the model did not show errors by chance.

The counterexamples demonstrated that the cause of the error was that the mutator observed that the collector installed α into the from-space object on line 11 in Figure 8 before it observed the collector write the value of the from-space object to the to-space copy on line 9. After we inserted `McSPIN_fence` before the collector installed α into the from-space object on line 10, model checking with PSO and RMO with the CAS that had no memory barrier effect did not show errors, as shown in the rows labeled "Clover(for RMO)" in Table 2.

4.4 Sapphire

Sapphire was proposed by Hudson and Moss [2001, 2003]. The authors did not specify a particular memory model, but they specified lines that should be executed in order in their pseudocode. Ritson et al. [2014] proposed a variant of Sapphire for the x86 Haswell processor. In this section, we consider both algorithms.

4.4.1 Algorithm. The key of Sapphire is to maintain an invariant that a from-space object always, even after the object has been copied, has the up-to-date location. As a result of this invariant, the mutator can read from from-space objects without read barriers. The write barrier, however, has to write to both the from-space object and its to-space copy to keep the from-space object and its to-space copy consistent. The collector needs to be careful not to overwrite the value that the mutator has written to the to-space copy with a stale value.

Algorithm 5 shows the pseudocode for Sapphire of Hudson and Moss [2003]. Hudson and Moss introduced the mark, allocate, and copy phases separately, and demonstrated a technique to merge these phases. We concentrated on the separated copy phase. Thus, the `forwarded` flag was omitted from Algorithm 5. The lines marked with a dollar sign ($) are specified to be executed in order. The `copy` procedure assumes that the memory for the to-space copy has been allocated, and the pointer to the to-space copy can be obtained by calling `ForwardingInfo`. In the copy phase, the mutators keep using the from-space objects. Hence, the mutators can freely read from the from-space objects. For writes, the mutators are required to cooperate with the collector; they write the same value to both the original object and to-space copy. More specifically, they first write a value to the original object and then write the same value to the to-space copy. Note that, when a mutator writes a pointer, it does not write the same value but an equivalent to-space pointer, that is, when it writes a pointer pointing at an object X into a from-space object Y, it writes a pointer pointing at the to-space copy of X to the to-space copy of Y. We omitted this special treatment of pointers from Algorithm 5 because we do not consider pointers in this paper. Thus, the collector copying a field and the mutator writing to the field may race. Therefore, the collector checks after copying if the to-space copy has the latest value, which is in the from-space object. If the check fails `MAX_CYCLES` times, it falls back to the copying protocol that uses the LL/SC instructions.

Ritson et al. [2014] is similar to Algorithm 5, but it does not have the protocol that uses the LL/SC. It also permits reordering and has a fence after the line of `vo = o[f]` in the `copy` procedure.

4.4.2 Results and Findings. We modeled Sapphire straightforwardly. We inserted full memory barriers between reads and writes that were specified to be executed in order. Note that McSPIN provides LL/SC instructions. The complete models are shown in the appendix. Unlike the concurrent copying protocols that we checked in the prior sections, Sapphire does not use the CAS. Thus, we conducted model checking against memory models only with the CAS that provided the full memory barrier effect. As a result, we ensured that Hudson and Moss [2003] worked with all the memory models and Ritson et al. [2014] worked only with SC and TSO. The results are shown in Table 2.

ALGORITHM 5: Copy method and barriers of Sapphire.

```
copy(o):
  NEXT_FIELD:
  foreach (f in fields of o):
    oo = ForwardingInfo(o);
    for (i = MAX_CYCLES; i > 0; i--):
      vn = o[f];                          $
      oo[f] = vn;                         $
      vo = o[f];                          $
      if (vo = vn):
        continue NEXT_FIELD;
    LL(oo[f]);                            $
    vn = o[f];                            $
    SC(oo[f], vn);                        $

read(o,f):
  return o[f];

write(o,f,v):
  o[f] = v;                                    $
  oo = ForwardingInfo(o);
  oo[f] = v;                                   $
```

4.5 Other GC Algorithms

We investigated the property that we mentioned in Section 2 of concurrent copying protocols with coarse-grained synchronizations. Some of the GC algorithms that we investigated stop all mutators at yieldpoints while collectors are executing a critical section. We modeled the critical section of the collector using an atomic block. Instead of forcing the mutator to block at a yieldpoint, we ceased the execution path if the atomic block was executed when the mutator was not at a yieldpoint. As shown in Table 2, none of the protocols showed errors. Below, we provide the notes for modeling.

Baker: Halstead [1985] proposed a Baker-style [Baker 1979] mostly concurrent GC algorithm. It copies all objects directly pointed to from the root set atomically at the beginning of GC. Thus, GC finished atomically in our framework.

Replication GC: Replication GC [Nettles and O'Toole 1993; Nettles et al. 1992] constructs a replica of the object graph in the to-space while the mutators are using the from-space. When a mutator writes to from-space objects, this event is notified to the collector through the mutation log, on which the collector and mutators synchronize. The collector applies the mutator's updates to the to-space copies.

Compressor: Compressor [Kermany and Petrank 2006] and Pauseless [Azul 2008] use page-level synchronization using the page protection mechanism of the OS and/or hardware. We modeled page protection using a shared flag. Because Pauseless copies the root objects atomically in the same manner as Baker, GC finished atomically in our framework.

Schism: In Schism [Pizlo et al. 2010], each object consists of at most three parts: a sentinel, spine, and payload fragments. Only spines have non-uniform sizes, and hence, need to be moved for defragmentation, but they are immutable. Thus, the values the mutators write are never lost.

5 RELATED WORK

There are few works on the formal and mechanical verification of concurrent copying GC against weak memory models. Except our previous work [Abe et al. 2016], which we mention later, the most connected work is Gammie et al. [2015]. The authors proved the safety of the collector of Doligez and Gonthier [1994] under the x86 TSO memory model. Their work is different from ours in three respects: First, they did not consider copying. Second, they used a single memory

model. Third, they proved safety using the proof assistant Isabelle/HOL, whereas our approach used model checking. Vechev et al. [2007] verified the correctness of auto-generated probably correct concurrent mark sweep algorithms. Unlike our bounded model checking, they introduced a sound abstraction based on shape analysis (Sagiv et al. [2002]) to represent an unbounded number of heap locations using a bounded abstract representation. However, they did not discuss memory models. Hawblitzel and Petrank [2009] verified a copying collector and a mark sweep collector using the Z3 theorem prover. The collectors were not concurrent collectors. Ugawa et al. [2014] proposed a method for implementing weak references in on-the-fly collectors and verified their proposed method using model checking. Their model checking was not memory model conscious. GC algorithms have sometimes been proposed with a formal discussion of correctness. Doligez and Gonthier [1994] proposed an on-the-fly garbage collector with a formal proof of correctness. Sapphire [Hudson and Moss 2003] discussed the correctness of its concurrent copying protocol.

There exist some studies that include the verification of not only garbage collectors but also concurrent data structures. Burckhardt et al. [Burckhardt et al. 2007] developed CheckFence, which statically checks the consistency of data type implementations for given bounded test programs and memory models, and found several bugs by modeling five concurrent data structures in the CBMC [Kroening and Tautschnig 2014] style. They also demonstrated ablation experiments by inserting/removing some memory fences. However, they considered only two memory models, SC and their own memory model, which is stronger than TSO/PSO/RMO, but weaker than the memory model of PowerPC because it lacks multi-copy-atomicity [Sarkar et al. 2011], and implemented one type of CAS. In this paper, we considered six memory models.

Norris and Demsky [2013], Ou and Demsky [2017], and Travkin et al. [2013] demonstrated the verification of several concurrent data structures, such as concurrent queues. However, Norris and Demsky [2013] and Ou and Demsky [2017] considered the C/C++ memory model [ISO/IEC 14882:2011 2011; ISO/IEC 9899:2011 2011] only, and Travkin et al. [2013] considered TSO only. An objective of our paper is to conduct model checking of concurrent copying protocols with *various* memory models, and distinguish the behaviors of the protocols with one memory model from those with another memory model.

Memory model-conscious model checking has been actively studied [Jonsson 2008; Linden and Wolper 2010; Yang et al. 2005], and model checkers are being developed [Abdulla et al. 016l; Abe and Maeda 2014; Travkin and Wehrheim 2016]. Additionally, existing model checkers have been extended to support weak memory models [Tomasco et al. 2016; van der Berg 2013]. However, these works considered only small models, such as mutual exclusion algorithms and competition benchmarks [SV-COMP 2016]. To the best of our knowledge, there have been no works that have investigated a property of larger models comparable to those that we have constructed in this paper, except for that by Abe et al. [2016]. Abe et al. [2016] improved the McSPIN model checker to reduce its space exploration and evaluated it using the preliminary (and limited) version of our models as benchmarks.

Ridge [2010], Ferreira et al. [2010], Vafeiadis et al. [Lahav and Vafeiadis 2015; Turon et al. 2014; Vafeiadis and Narayan 2013], Abe et al. [Abe and Maeda 2016, 2017], and Kaiser et al. [2017] proposed logic for weak memory models. Furthermore, Alexander J. Summers [2017] proposed *automatic* verification using separation logic for the C11 memory model, and demonstrated the verification of several programs. However, these are different approaches from our approach, which is model checking with weak memory models.

6 CONCLUSION

We investigated a correctness-related property of concurrent copying protocols of various concurrent copying GC algorithms. We used the McSPIN model checker and conducted bounded

model checking with six memory models. In this wide range of case studies, we found faults in two concurrent copying protocols and were able to fix them; Staccato did not work correctly under PSO and RMO, and Clover did not work even under SC; the problem with Clover is not relevant to memory models. We also modified the concurrent copying protocol of Stopless so that it works under memory models other than those that the original Stopless assumes, that is, PSO and RMO with the CAS providing no memory barrier effect. To fix and modify concurrent copying protocols, we added memory barriers one by one in the models by referencing the counterexamples generated by McSPIN. Whenever we added a memory barrier, we checked the modified model. This was possible because McSPIN fully automatically checked the property in a reasonable time: approximately an hour for a long case. One might want to prove a theorem to be totally confident about the correctness. However, it might not be as easy as translating an algorithm into a model for non-experts, even using proof assistants. These experiences demonstrated that bounded model checking is a feasible approach to investigate behavioral properties of concurrent copying protocols under weak memory models.

Although we demonstrated the feasibility of bounded model checking, there are still several limitations. In this study, we investigated one correctness-related property: the values written by the mutator are never lost in a way that violates SC. Although this property is important in the copy phase, to guarantee the correctness of the entire concurrent copying GC, investigating other properties, such as safety and progress, is necessary. Studying the mechanical verification of other correctness-related properties in all the phases, including the mark and flip phases, and developing automated verification of the entire practical concurrent copying GC is future work. We also limited the target of verification to the behavior of one collector and one mutator. In theory, there are no difficulties in considering multiple collectors and multiple mutators, but it could not be completed within a reasonable memory and CPU time. Improving modeling techniques, for example, using abstraction, may enable us to verify such programs within a reasonable memory size, and this is future work. Additionally, we did not directly consider real-world memory models defined by real CPU architectures, although the memory models considered in this paper represent the essential characteristics of real-world memory models, as explained in Section 3.1. The direct management of real-world memory models is on our list of future work.

ACKNOWLEDGMENTS

We would like to thank Richard Jones, Scott Owens, Masahiro Yasugi, Shigeyuki Sato, and the anonymous referees for their valuable comments and helpful suggestions. We thank Maxine Garcia from Edanz Group (www.edanzediting.com/ac) for English proofreading of a draft of this manuscript. This work was partly supported by the JSPS KAKENHI Grant Numbers 16K00103 and 16K21335.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016l. Stateless Model Checking for POWER. In *Proceedings of 28th International Conference on Computer Aided Verification (CAV '16) (LNCS)*, Vol. 9780. Springer-Verlag, 134–156.

Tatsuya Abe and Toshiyuki Maeda. 2014. A General Model Checking Framework for Various Memory Consistency Models. In *Proceedings of 19th Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '14)*. IEEE, 332–341.

Tatsuya Abe and Toshiyuki Maeda. 2016. Observation-based Concurrent Program Logic for Relaxed Memory Consistency Models. In *Proceedings of 15th Asian Symposium on Programming Languages and Systems (APLAS '16) (LNCS)*, Vol. 10017. Springer-Verlag, 63–84.

Tatsuya Abe and Toshiyuki Maeda. 2017. Concurrent Program Logic for Relaxed Memory Consistency Models with Dependencies across Loop Iterations. *Journal of Information Processing* 25 (2017), 244–255.

Tatsuya Abe, Tomoharu Ugawa, Toshiyuki Maeda, and Kousuke Matsumoto. 2016. Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models. In *Proceedings of Symposium on Dependable Software*

Engineering Theories, Tools and Applications (SETTA '16) (LNCS), Vol. 9984. Springer-Verlag, 118–135.

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: a tutorial. *IEEE Computer* 29, 12 (Dec. 1996), 66–76.

Peter MÃijller Alexander J. Summers. 2017. Automating Deductive Verification for Weak-Memory Programs. (2017). arXiv:1703.06368.

ARM 2016. *ARM Architechture Reference Manual ARMv8, for ARMv8-A architechture profile.* ARM.

Azul. 2008. *Pauseless Garbage Collection.* White paper AWP–005–020. Azul Systems Inc. http://www.azulsystems.com/products/whitepaper/wp_pgc.pdf

David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. 2001. Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM Press, 92–103. https://doi.org/10.1145/378795.378819

Henry G. Baker. 1979. Optimizing Allocation and Garbage Collection of Spaces in MacLLisp. In *Artificial Intelligence: An MIT Perspective*, Winston and Brown (Eds.). MIT Press. http://home.pipeline.com/~hbaker1/OptAlloc.ps.gz

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Proceedings of 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM Press, 12–21.

Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Proceedings of 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM Press, 70–83. https://doi.org/10.1145/174675.174673

Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized Memory Models and Concurrent Separation Logic. In *Proceedings of 19th European Symposium on Programming (ESOP '10)*. 267–286.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM Press, 608–621.

Peter Gammie, Antony L. Hosking, and Kai Engelhardt. 2015. Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO. In *Proceedings of 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM Press, 99–109. https://doi.org/10.1145/2737924.2738006

Robert H. Halstead. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Chris Hawblitzel and Erez Petrank. 2009. Automated Verification of Practical Garbage Collectors. In *Proceedings of 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 441–453. https://doi.org/10.1145/1480881.1480935

Richard L. Hudson and J. Eliot B. Moss. 2001. Sapphire: Copying GC Without Stopping The World. In *Joint ACM-ISCOPE Conference on Java Grande*. ACM Press, 48–57. https://doi.org/10.1145/376656.376810

Richard L. Hudson and J. Eliot B. Moss. 2003. Sapphire: Copying Garbage Collection Without Stopping the World. *Concurrency and Computation: Practice and Experience* 15, 3–5 (2003), 223–261. https://doi.org/10.1002/cpe.712

IBM. 2015. *Power ISA(TM) Version 3.0.*

Intel. 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*

ISO/IEC 14882:2011. 2011. *Programming Language C++.*

ISO/IEC 9899:2011. 2011. *Programming Language C.*

Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall.

Bengt Jonsson. 2008. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Computer Architecture News* 36, 5 (2008), 65–71.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *Proceedings of 31st European Conference on Object-Oriented Programming (ECOOP '17) (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 17:1–17:29.

Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental and Parallel Compaction. In *Proceedings of 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM Press, 354–363. https://doi.org/10.1145/1133981.1134023

Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS '14) (LNCS)*, Vol. 8413. Springer-Verlag, 389–391.

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Proceedings of 42nd International Colloquium on Automata, Languages, and Programming (ICALP '15)*. 311–323.

Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of 2015 International Symposium on Memory Management (ISMM '15)*. ACM Press,

70–80. https://doi.org/10.1145/10.1145/2754169.2754187

Alexander Linden and Pierre Wolper. 2010. An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In *Proceedings of 17th International SPIN Workshop on Model Checking of Software (SPIN '10) (LNCS)*, Vol. 6349. Springer-Verlag, 212–226.

Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. 2008. *Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors.* IBM Research Report RC24505. IBM Research. http://domino.watson.ibm.com/comm/research_people.nsf/pages/dgrove.rc24504.html

Scott Nettles and James O'Toole. 1993. Real-Time Replication-Based Garbage Collection. In *Proceedings of 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '93)*. ACM Press, 217–226. https://doi.org/10.1145/155090.155111

Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. 1992. Replication-Based Incremental Copying Collection. In *Proceedings of International Workshop on Memory Management (IWMM '92)*. ACM Press, 357–364. https://doi.org/10.1007/BFb0017201

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM Press, 131–150.

Peizhao Ou and Brian Demsky. 2017. Checking Concurrent Data Structures Under the C/C++11 Memory Model. In *Proceedings of 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM Press, 45–59.

Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP '10) (LNCS)*, Vol. 6183. Springer-Verlag, 478–503.

Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. 2007a. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *Proceedings of 6th International Symposium on Memory Management (ISMM '07)*. ACM Press, 159–172. https://doi.org/10.1145/1296907.1296927

Filip Pizlo, Antony L. Hosking, and Jan Vitek. 2007b. Hierarchical Real-time Garbage Collection. In *Proceedings of 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*. ACM Press, 123–133. https://doi.org/10.1145/1254766.1254784

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. A Study of Concurrent Real-Time Garbage Collectors. In *Proceedings of 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM Press, 33–44. https://doi.org/10.1145/1379022.1375587

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: Fragmentation-Tolerant Real-Time Garbage collection. In *Proceedings of 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM Press, 146–159. https://doi.org/10.1145/1806596.1806615

Tom Ridge. 2010. A Rely-Guarantee Proof System for x86-TSO. In *Proceedings of International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE '10)*. Springer-Verlag, 55–70.

Carl G. Ritson, Tomoharu Ugawa, and Richard Jones. 2014. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *Proceedings of 2014 International Symposium on Memory Management (ISMM '14)*. ACM Press, 105–115. https://doi.org/10.1145/2602988.2602992

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems* 24, 3 (2002), 217–298.

Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. 2007. A Theory of Memory Models. In *Proceedings of 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM Press, 161–172. https://doi.org/10.1145/1229428.1229469

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM Press, 175–186.

SV-COMP. 2016. Competition on Software Verification. (2016). https://sv-comp.sosy-lab.org/.

Ermenegildo Tomasco, Omar Inverso Truc Nguyen Lam, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '16)*. IEEE, 193–200.

Oleg Travkin, Annika Mütze, and Heike Wehrheim. 2013. SPIN as a Linearizability Checker under Weak Memory Models. In *Proceedings of Haifa Verification Conference (LNCS)*, Vol. 8244. Springer-Verlag, 311–326.

Oleg Travkin and Heike Wehrheim. 2016. Verification of Concurrent Programs on Weak Memory Models. In *Proceedings of International Confederation for Thermal Analysis and Calorimetry (ICTAC '16) (LNCS)*, Vol. 9965. Springer-Verlag, 3–24.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. ACM Press, 691–707.

Tomoharu Ugawa, Richard Jones, and Carl G. Ritson. 2014. Reference Object Processing in On-The-Fly Garbage Collection. In *Proceedings of 2014 International Symposium on Memory Management (ISMM '14)*. ACM Press, 59–69. https://doi.org/10.1145/2602988.2602991

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM Press, 867–884.

F.I. van der Berg. 2013. *Model checking LLVM IR using LTSmin: using relaxed memory model semantics*. Master's thesis. University of Twente.

Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. 2007. CGCExplorer: A Semi-Automated Search Procedure for Provably Correct Concurrent Collectors. In *Proceedings of 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM Press, 456–467. https://doi.org/10.1145/1250734.1250787

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. 2005. UMM: an operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience* 17, 5-6 (2005), 465–487.