Towards a Model Checking Framework for a New Collector Framework

Bochen Xu
College of Info. & Comp. Sciences
Univ. of Massachusetts Amherst
Amherst, MA, USA
bochenxu@umass.edu

Eliot Moss
College of Info. & Comp. Sciences
Univ. of Massachusetts Amherst
Amherst, MA, USA
moss@umass.edu

Stephen M. Blackburn
Google
Australia
steveblackburn@google.com

ABSTRACT

Garbage collectors provide memory safety, an important step toward program correctness. However, correctness of the collector itself can be challenging to establish, given both the style in which such systems are written and the weakly-ordered memory accesses of modern hardware. One way to maximize benefits is to use a framework in which effort can be focused on the correctness of small, modular critical components from which various collectors may be composed. Full proof of correctness is likely impractical, so we propose to gain a degree of confidence in collector correctness by applying model checking to critical kernels within a garbage collection framework. We further envisage a *model framework*, paralleling the framework nature of the collector, in hope that it will be easy to create new models for new collectors. We describe here a prototype model structure, and present results of model checking both stop-the-world and snapshot-at-the-beginning concurrent marking. We found useful regularities of model structure, and that models could be checked within possible time and space budgets on capable servers. This suggests that collectors built in a modular style might be model checked, and further that it may be worthwhile to develop a model checking framework with a domain-specific language from which to generate those models.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification*; **Runtime environments**.

KEYWORDS

model checking, garbage collection, stop-the-world, concurrent, snapshot-at-the-beginning

ACM Reference Format:
Bochen Xu, Eliot Moss, and Stephen M. Blackburn. 2022. Towards a Model Checking Framework for a New Collector Framework. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3546918.3546923

MPLR '22, September 14–15, 2022, Brussels, Belgium

ACM ISBN 978-1-4503-9696-7/22/09.
https://doi.org/10.1145/3546918.3546923

1 INTRODUCTION

The Chromium Project found that of 912 serious security bugs they analyzed since 2015, about 70% were due to memory safety violations, half of those due to use-after-free errors [4]. Garbage collected languages avoid this source of security bugs, but depend on correctness of their garbage collector. Given the complexity of production garbage collectors, reasoning about their correctness can be extremely challenging. This sometimes leads to real security bugs, such as use-after-free errors [19].

Here we explore a two-pronged approach to increasing confidence in the correctness of garbage collectors. We start by using MMTk, which has modularity and code reuse as first order design goals [2], and more recently has developed a system of generic work units and work scheduling which further modularizes the most frequently executed kernels within a collector [18]. Second, we develop models of certain kernels and their scheduling, and for the snapshot-at-the-beginning (SATB) write barrier [24] used by some concurrent collectors.

MMTk's work packet framework uses a pool of packets of work units of various kinds and a number of worker threads. Any worker can process any packet, but certain kinds of packets are legal to process only at certain times. The overall system is controlled by a releaser process that turns on permission to process kinds of work as related guard predicates become satisfied. The implementer of a specific collector designs the kinds of packets, guard predicates, etc., according to the needs of that collector. Thus, the framework is highly generic and customizable, and emphasizes a well-engineered highly concurrent work pool approach.

This approach may be great for performance, but concurrent collectors are error-prone due to interactions between the collector and other components of the system, such as mutators and the memory allocator. Further, writing the releaser in event-driven style also makes it tricky: guard predicates must be just right for correctness (as we experienced in developing our models!) An additional challenge is the weak memory access ordering of modern hardware, where operations performed by one thread in a given order may be perceived by other threads in other orders, unless special ordering instructions are used.

It would substantially assist collector developers if we could provide tools to build confidence in the correctness of their designs that target the MMTk framework. Since full machine-checked proofs of correctness appear impractical for everyday development, we propose to apply model checking. Model checking has been successful in building confidence in the correctness of concurrent systems, including some concurrent garbage collectors ([21, 22], for example). Our goal here is to develop a structure or framework of

model checking that parallels the structure of the MMTk framework. We have done so for two sample collectors and believe it feasible to design a domain specific language for MMTk for mechanically generating models for checking.

Our primary contributions are that we: (i) derive recurring model checking patterns from the MMTk work packet framework; (ii) develop models that check the critical parts of GC algorithms; (iii) illustrate our approach with models for stop-the-world and concurrent SATB mark sweep collection; (iv) exhibit that the models work under three different weak memory protocols; and (v) demonstrate that such models are feasible to check in reasonable time and space.

What this paper is and is not: This paper is not primarily about any existing or new GC algorithm, though we use two algorithms as examples. In contrast, it is about a new style for writing collectors. Second, this paper is not primarily about introducing or defending that style, though we describe it as necessary, but about how to gain confidence, via model checking, in collectors written using the style. While it points to the feasibility of a model checking framework, it does not propose a concrete framework. This paper *is* about the feasibility of writing checkable models in a stylized way that matches this new style of writing collectors.

2 RELATED WORK

Most prior work applies formal verification tools and techniques to specific GC algorithms, while we focus on checking the correctness of a GC *framework*, which is applicable to various GC algorithms. The closest previous work is that of Ugawa et al. [21], which applied model checking to critical parts of the copy phases of at least five specific concurrent copying collectors under six different memory models. We demonstrate that we can straightforwardly generalize our **memory** module to three memory models. The main distinction is that we target a GC framework, not particular algorithms.

Abe et al. [1] explored optimizations to mitigate the state explosion problem when model checking GC under weak memory models. Their focus was on the weak memory aspect, and their approach is rather different from ours, working from constraints on execution traces expressed as first-order predicates. Our weak-memory model is operational rather than predicate based.

Ugawa et al. [22] extended the JikesRVM MMTk with support for on-the-fly GC algorithms, providing thread-by-thread stack scanning and ragged phase changes. They further implemented Transactional Sapphire, a transactional memory version of Sapphire [13], over this library and model-checked it. While they generalized the JikesRVM MMTk, the MMTk we aim to model check is a much more general and more loosely controlled framework. They also check a single GC algorithm, albeit a complex one.

Vechev et al. [23] describe a framework to automatically *explore* new GC algorithms based on "building blocks." This differs from the framework with which we work, because our framework design emphasizes the implementation of GC algorithms rather than generation of them. The algorithm found by Vechev et al. [23] is also model-checked, but under fewer scenarios than ours.

Bowman et al. [3] showed how to model correctness properties of GC algorithms with temporal logic. They presented examples of CSS specifications of two simple GC algorithms and a more complex one (cyclic reference counting). We use much simpler LTL formulas to express correctness properties, because the transitive closure operator is not supported in Promela, and because we check specific small graphs. Bowman et al. [3] did not do actual model checking of their formulas.

For reasons of space we omit other, less related, work on model checking GC.

There has also been work on machine-checked proofs of GC correctness. Ericsson et al. [6, 7] verified the correctness of a generational GC algorithm for CakeML [8, 15–17] with HOL4 [11]. Gammie et al. [9] verified an on-the-fly concurrent mark-sweep GC algorithm under a weak memory model (TSO), with Isabelle/HOL [20]. Zakowski et al. [25] verified a concurrent GC algorithm with Coq [5]. These all checked a single algorithm, but more notably, full formal verification continues to be very difficult and time-consuming. The MMTk framework is intended to help construct GC implementations quickly, so rapid model checking is a better fit for most cases in practice. Perhaps at some future time researchers will discover a way to leverage model checking results toward more rapid full formal verification.

3 THE MMTK WORK PACKET FRAMEWORK

We now explain the various concepts and components of the MMTk work packet framework.

The framework was motivated by a number of goals, including maximizing: (i) *scalability* through concurrency among GC tasks while maintaining correct ordering among them; (ii) *code reuse* by providing a single, well-engineered task-processing framework within which most GC work could be performed; and (iii) *performance* from using highly optimized kernels.

The framework centers on *work units*, each of which is a discrete entity (as small as a pointer or as large as a stack) to be processed, and *work packets*, each of which contains multiple work units and code to perform the processing over them. GC worker threads simply consume and process work packets, agnostic to their purpose. Some work packets may generate additional work. The GC designer will prescribe rules that govern the scheduling of work in order to ensure the correctness of the algorithm. Depending on the algorithm, GC work may be performed concurrently or in mutual exclusion to mutator work. For example, a work packet might perform a simple mark of each of its work units. When a GC thread claims such a packet, it iterates through each of the work units in the packet, performing a mark operation and if the target is not already marked, creates a scan work unit for the object. A collector's scheduling rules dictate when packets may be processed. For example, finalizer packets might be allowed to run only after all tracing packets have been processed, and packets that move objects may be allowed to run only after all mutator threads have yielded. The framework assumes that work units of a given kind can be processed in any order, and unless otherwise indicated, can be processed concurrently.

Although motivated by performance and software engineering considerations, the design appears to lend itself to assurances of correctness. In practice, most work packet kernels tend to be just a few lines of code, and the guards that dictate their scheduling are often straightforward. We now describe the elements of the framework in more detail.

Work units: A work unit is the smallest piece of work that a given collector will create and process, e.g., "mark an object" or "scan a mutator stack."

Work kinds: A collector defines whatever kinds of work units it needs. Again, "mark an object" and "scan a thread stack" are typical examples. The key property of a work kind is that processing of a kind can be turned on and off as a collector moves through different parts of a collection cycle.

Worker (thread): A collector has one or more worker threads. Any worker can perform any work unit since work units have object-oriented dispatch to the code for processing them.

Packets: For efficiency, work units of the same kind are grouped into packets. A packet is what a worker thread can claim, and then process the work units within it.

Buckets: A bucket consists of one or more work kinds. A bucket may be *open*, meaning its kinds of work may be processed, or *closed*, meaning they may not be. A packet of a given kind can be processed if there is any open bucket that includes that kind. A collector may place restrictions on how many packets of a given kind may be processed concurrently. In particular, it can require that certain kinds of work be processed sequentially, which is useful when the work is coded in a way that does not support concurrent processing of work of that kind.

Guards: Each bucket has a guard, which is a condition under which the bucket may be opened. For example, in our sample collectors, we require that all mutator stacks be scanned before further marking proceeds.[1] A guard may also specify when to close a bucket.

Releaser: The releaser is a unique thread that watches for guards to be come true and triggers opening and closing of buckets.

Work pool: The overall pool of packets, of various kinds, that have been generated and can be claimed by workers, according to whether appropriate buckets are open. We model the pool as a set of buckets holding disjoint sets of packets. The pool and workers collaborate to provide a high performance concurrent work stealing collector.

As a concrete example of collector designs for this framework, we summarize in Table 1 the kinds, buckets, etc., of the example collectors we model checked. The table indicates how work units of one kind may create units of another kind. Marking completes when there are no unprocessed **markObj** and **scanObj** work units. The difference between the stop-the-world (STW) collector and snapshot-at-the-beginning (SATB) is only that mutators run during marking in SATB, employing a write barrier and marking newly allocated objects.

We believe the novelty of the MMTk work packet framework is its: (i) organization with event-triggered movement through the "phases" of collection, using guards and buckets, and (ii) emphasis on and provision of a highly concurrent work pool. While one can definitely express a sequential or incremental collector in the framework, it really comes into its own in supporting parallel and concurrent collection.

[1] This is not logically necessary. We did it to explore how to set up appropriate guards, etc.

4 EXAMPLE GC ALGORITHMS MODELED

To explore the possibility of model checking for the MMTk framework, we desired well-known example algorithms that are, on the one hand not *overly* complicated, but on the other hand, concurrent enough to be interesting. We settled on parallel stop-the-world (STW) marking and snapshot-at-the-beginning (SATB) concurrent marking [24]. Both use the same mark-sweep collector design. They differ in that SATB allows mutators to run during collection and thus employs a suitable write barrier and must also deal with allocation during marking.

We express these collectors in the MMTk framework using these work unit / packet kinds, with the indicated work:

- **startGC**: Sent once by a mutator to start the marking cycle. We model a single collection cycle at present.
- **scanStack**: One of these is sent by the **startGC** worker for each mutator. The work unit requires asking the mutator to *yield* to the collector. While a mutator is yielded it will do no work. Once the mutator yields, the worker scans the mutator's stack entries creating **markObj** work units for each non-null pointer it finds. It then drops the yield request and the mutator can resume its own work.
- **markObj**: One of these is sent for each pointer discovered, whether it be in stack scanning, object scanning, or by a write barrier. The work unit involves doing a **fetch-and-or** operation on the mark bit of the object. If the bit was not previously set, the worker creates a **scanObj** work unit for the object. Using an atomic **fetch-and-or** guarantees that there will be at most one **scanObj** request for the object. Allowing multiple **markObj** requests insures that model checking will explore a race between **markObj** workers.
- **scanObj**: One of these is sent for each object discovered to be reachable. The work is to examine each pointer of the object, and for non-null ones, request **markObj** for the target.

Actual collectors might fold **markObj** and **scanObj** together, or have the thread that discovers a pointer check the mark status of its referent, mark the object if necessary and request scanning. Instead, we chose this approach to explore more deeply how the different work is coordinated in the framework. Note also that a given object can have more than one **markObj** work unit queued for it, while it will have at most one **scanObj** work unit. This is another intentional variation to see how each might be modeled.

The releaser triggering is as follows. Initially, the **startGC** bucket is open. Once a **startGC** worker has entered the **scanStack** requests, it indicates that it is done. The releaser then opens the **scanStack** bucket. Once all stacks are scanned, the releaser opens the **mark** bucket, which include both **markObj** and **scanObj** work units. Marking completes when there is no remaining **markObj** or **scanObj** work unit.

To this collector organization, SATB adds the following. Each update to a field of an object includes a *field remembering* barrier.[2] This barrier, whose intent is to insure that the old referent of the field (if any) is made "at least gray" according to the tri-color marking

[2] MMTk enables this barrier at all times for performance reasons. We checked both enabling it all the time and enabling it only during marking.

Table 1: MMTk setup for parallel stop-the-world marking

Bucket	Kind	Work	Guard	Generates
start	**startGC**	request stack scans	initially open	**scanStack**
stack scan	**scanStack**	request mutator yield; upon yield, scan stack, release yield	start complete	**markObj**
mark	**markObj**	insure object marked; if not marked before, request object scan	stack scan complete	**scanObj**
	scanObj	examine object's pointers and request target objects be marked		**markObj**

invariant, creates a **markObj** work unit for the old referent.[3] This guarantees that the object will be marked if it is not already. This write barrier insures that, if the pointer being deleted is the last one pointing to its target, the target is still marked. This meets the criterion of snapshot-at-the-beginning, namely that every object reachable when marking began (which for this algorithm is when it starts scanning stacks) will be marked [24]. The second difference between the SATB and STW collectors is that if an object is allocated during or after stack scanning and marking in SATB, it is allocated "black," according to the tri-color invariant. To accomplish this, the allocating thread directly sets the mark bit of the object.

5 MODEL CHECKING WITH SPIN

We use the well-known model checker SPIN [12]. To use SPIN, one writes the desired models in the Promela modeling language. Promela provides concurrent *processes* that can communicate using *channels*, in the manner of *communicating sequential processes* [10]. The channel semantics include the ability not only to (try to) send or receive the next message, but to receive the next message that meets a particular pattern of values and wildcards, to test whether such a message is in the channel without receiving it, and to iterate over the messages in a channel without removing them. Processes can execute assignments similar to those of C, and control constructs include: guarded commands, where a statement guarded by a side-effect-free boolean expression can be executed only if the guard is true; conditionals; general iteration; counted iteration; **break**; and **goto**. Conditionals and general iteration allow guarded statements whose guards overlap, leading to non-determinism. Another source of non-determinism is the **select** statement, which chooses a number from a given range of integers. To control non-determinism and allow atomic operations in this concurrent setting, Promela provides **d_step** which groups a sequence of deterministic operations into one larger step (state transition), and **atomic**, which guarantees that no other process takes a step while its sequence of statements are carried out. (These differ in that **atomic** can be non-deterministic and may consist of multiple steps.) Data types in Promela include **bool**, **byte**, and some others that we did not use, as well as arrays of these things and an analog of C structure types. Notably Promela does not provide functions or procedures, though it is easy to use C macros since SPIN uses the C preprocessor, and Promela provides **inline**, which is essentially a C macro definition with slightly nicer syntax.

While it is possible to do one or more, possibly non-deterministic, simulation runs of a Promela model, SPIN can also do verification by generating an executable that can explore all possible paths. Further, Promela gives the model writer the ability to include one or more *linear temporal logic* (LTL) formulas that the executable can check. These formulas can refer to any global variables of the model. For example, one can use the LTL modal operator **[]**, often read "always," to express an invariant, and **<>**, often read "eventually," to express a goal that must be achieved. Further, to show that some property **P** can happen, but need not happen always, one can test both **[] P** and **[] !P**—both should fail. When checking of an LTL formula fails, SPIN can provide a *trace*, which is a history of the steps of the various processes of the model that led to violation of the LTL formula. This is helpful in model debugging.

For model checking generally, it is important to keep the overall program state, size of the programs, number of processes, number of steps needed, etc., as small as possible. While SPIN performs a number of sophisticated optimizations of the search, and offers various ways to compress states, it requires care to avoid combinatorial explosion and some thought about how best to represent the system state, where one can use **d_step** and **atomic** without undermining the intended model semantics, etc. Nevertheless, some of our models require hundreds of gigabytes of memory and hours to run.

We chose SPIN for this work because we were somewhat familiar with it. However, it does have particular advantages. One is that it is not committed toward any particular programming language, though it bears closest resemblance to C. Another is that its **select** statement allows arbitrary choice among alternatives, thus exploring the possible choices made by any policy rather than checking correctness of only a single concrete policy. In contrast, model checkers for C programs cannot check all possible policies for choosing the order in which to do work.

For reference, we used SPIN version 6.5.1, dated 2021/06/03. We used weak fairness, which requires that if a process continuously is able to take a step, then eventually it will. Thus, other processes can run arbitrarily far ahead of an enabled process, but not infinitely far. Like any system, SPIN has it idiosyncrasies, vices, and virtues, but we found it suited our purposes and the learning curve was not too steep.

6 WEAK MEMORY MODEL

The title of this section is intentionally ambiguous: we will explain which model of weak memory we assume for our model checking, and we will explain how we model it in Promela code. We begin with weak memory. We support these memory operations: **load**, **store**, **load-acquire**, **store-release**, **fence**, (atomic) **swap**, (atomic) **compare-and-swap**, and (atomic) **fetch-and-XXX** for **XXX** being **add**, **and**, and **or**. It would be relatively easy to add **lfence** and **sfence** operations if need be. Since we do not model

[3] Note that an object is *black* if it has been scanned, *gray* if it is marked or has been requested to be marked, but is not yet scanned, and *white* otherwise.

```
byte memory[HEAP_SIZE] = 0;
typedef mem_request {
  byte op; byte cpu; byte addr; byte newval; byte oldval;
};
chan hostToMemory = [HM_Q_LEN] of {mem_request};
typedef mem_response {
  byte op; byte addr; byte val;
  bool success; byte newval; byte oldval;
};
chan memoryToHost[CPUS] = [MH_Q_LEN] of {mem_response};
typedef SBEntry {byte cpu; byte address; byte value;};
chan storeBuffer = [SB_Q_LEN] of {SBEntry};
```

Algorithm 1. Memory request and channel declarations

writing or updating of instructions, we omit synchronization with instruction fetch. From the standpoint of the issuing thread, each of these operations has a beginning, those with a response have an end, and operations on the *same* memory location occur in the order they are issued, except that **load**s of the same location can pass one another. However, operations on *different* locations can be processed out of order at the memory, subject to these ordering constraints:

- A **load-acquire** must be processed before any operations that follow it from the same thread.
- A **store-release** must be processed after any operations that precede it from the same thread.
- A **fence** must be processed after preceding operations from the same thread and before following operations from the same thread.
- If a thread perceives the end of operation X before it issues operation Y, then X was definitely processed by the memory before Y.

These semantics correspond roughly with the relaxed and acquire-release semantics of the C/C++11 memory model. For stronger sequencing one can either wait for completion or use **fence** instructions. To be clear, our **compare-and-swap** is *strong*, meaning that it does not have spurious failures, since such failures can be gotten around by putting the **compare-and-swap** in a loop. Using build-time options,the memory module can also be configured to follow a sequentially consistent memory model (SC) or a total store order one (TSO).

Here is how we model this memory system in Promela (see Algorithms 1, 2, and 3). We have a single *memory* process that handles all memory requests. It models the coherent memory system of a modern multi-core processor as a single centralized memory (**memory**, a simple array of **byte**), since that is the semantics that a coherent cached memory system presents to programs. We use a single *hostToMemory* channel for all requests; a request indicates the operation, requesting cpu, address, new value, and (desired) old value (for **compare-and-swap**). This channel therefore holds all the as yet unprocessed memory requests from all cpus. We provide a separate channel for each cpu for responses from the memory to that cpu (**memoryToHost**). Responses indicate the operation, address, returned value (for loads), success code (for **compare-and-swap**), and new and old values (for atomic read-modify-write operations). The **[n] of** syntax in a **chan** declaration indicates the maximum number of messages the channel can hold. We size our channels so that sends never block.

```
inline check_store_buffer(addr, found, found_value) {
  d_step {
    found = false;
    for (ent in storeBuffer) {
      if
      :: ent.cpu == (CPUID) && ent.address == (addr) →
         found = true; found_value = ent.value;
      :: else → skip;
      fi
    }
  }
}
inline BEGIN_LOAD (addr) {
  d_step {
    check_store_buffer(addr, found, val);
    if
    :: found → memoryToHost[CPUID] !
                  mr_load((addr), val, true, 0, 0);
    :: else →
       hostToMemory ! m_load((CPUID), (addr), 0, 0);
    fi
  }
}
inline RECEIVE_LOAD(addr, var) {
  memoryToHost[CPUID] ?? mr_load(eval(addr), var, _, _, _);
}
```

Algorithm 2. Store buffer implementation

Real hardware includes a component called a *store buffer* for each cpu. This buffer holds all the stores from that cpu where the cpu has not yet received a response from the memory. We model store buffers with another Promela channel, using a single channel for the whole system, also shown in Algorithm 1. The store buffers are not used for messaging but as an associative lookup table. Load requests from cpus probe the store buffer to find the most recent store by the same cpu to the same address, and if there is one, the value stored is immediately sent back to the cpu, bypassing the memory. Otherwise the request is forwarded to the memory.

This is illustrated in Algorithm 2, which we include to show various features of Promela. **BEGIN_LOAD** calls **check_store_buffer** to determine whether there is a store buffer "hit". If there is (**found**), it immediately sends an **mr_load** (load response) message to the cpu, bypassing the memory; otherwise it sends an **m_load** (load request) to the memory. The syntax *chan* **!** *message* indicates to attempt to send *message* on *chan*. The syntax **if ::** $guard_1$ **->** $body_1$ ... **::** $guard_n$ **->** $body_n$ **fi** is a conditional statement. If $guard_i$ is true in the current state, then $body_i$ may be executed. The special guard **else** is true only if all other guards are false. In this case we see a simple if-then-else.

The **check_store_buffer** inline loops over all entries in **storeBuffer**, finding the last (most recently sent) store from the current cpu to the same address, if any. This is all done as a single deterministic step so that the check views an atomic snapshot of the store buffer state. The **d_step** of **BEGIN_LOAD** works because the computation is deterministic in any given state, and the message sends won't block because our queues are long enough.

The memory process executes the loop outlined in Algorithm 3. If the channel is non-empty, then the process atomically: discovers which requests are ready (legal to execute according to the ordering rules—these are what vary in case of the SC and TSO models), chooses one non-deterministically (**select**), and then processes it. The first **d_step** examines messages in the channel, in

```
do
:: nempty(hostToMemory) →
   atomic {
     d_step {
       // clear data structures (not shown)
       for (req in hostToMemory) {
         if
         :: req.op == m_load →
            if
            :: writer[req.cpu,req.addr] ||
                have_fence[req.cpu] → skip;
            :: else →
                ready[nready] = req; nready++;
                reader[req.cpu,req.addr] = true;
                any_access[req.cpu] = true;
         :: req.op == m_loadacq →
         ... // cases for other operations
         fi
       }
     }
     select (idx : 0 .. nready−1);
     d_step {
       hostToMemory ?? eval(ready[idx].op), ...;
       if
       :: ready[idx].op == m_load →
           memoryToHost[ready[idx].cpu] !
             mr_load, ready[idx].addr,
             memory[ready[idx].addr], ...
       :: ... // cases for other operations
       fi
     }
   }
   :: empty(hostToMemory) && (PROCS_RUNNING == 0) →
      break;
od
```

Algorithm 3. Memory process outline

```
active proctype releaser() {
  OPEN_START_GC();
  do
  :: MARK_DONE → break;
  :: else → d_step {
     if
     :: !SS_ALLOWED && SS_GUARD() →
        d_step {OPEN_SS();}
     :: !MARKING_ALLOWED && MARK_START_GUARD() →
        d_step {OPEN_MO(); OPEN_SO();}
     :: MARKING_ALLOWED && MARK_DONE_GUARD() →
        d_step {NOTE_MARKING_DONE();}
     vi
   }
  CD
}
```

Algorithm 4. Releaser

```
# define SS_GUARD() START_GC_DONE
bool MARK_DONE = false;
bool MARKING_ALLOWED = false;
# define MARK_START_GUARD() \
  (SS_ALLOWED && ALL_STACKS_SCANNED())
inline OPEN_START_GC () {START_GC_ALLOWED = true;}
inline OPEN_SS () {SS_ALLOWED = true;}
inline OPEN_MO () {
  MARKING_ALLOWED = true; MO_ALLOWED = true;
}
inline OPEN_SO () {SO_ALLOWED = true;}
# define MARK_DONE_GUARD() \
  (SO_BUCKET_EMPTY() && MO_BUCKET_EMPTY())
inline NOTE_MARKING_DONE() {MARK_DONE = true;}
```

Algorithm 5. Guard definitions

```
bool MUT_YIELD_REQUESTED[NUM_MUTATORS];
bool MUT_YIELDED [NUM_MUTATORS];
proctype mutator (byte MUT_ID) {
  ...
  do
  // Special case: start gc
  :: MUT_ID == 0 &&
      START_GC_ALLOWED && !START_GC_REQUESTED →
    REQUEST_START_GC();
  // Respond to yield request by yielding
  :: MUT_YIELD_REQUESTED[MUT_ID] && !MUT_YIELDED[MUT_ID] →
     MUT_YIELDED[MUT_ID] = true;
  // Respond to drop of yield request by unyielding
  :: !MUT_YIELD_REQUESTED[MUT_ID] && MUT_YIELDED[MUT_ID] →
     MUT_YIELDED[MUT_ID] = false;
  ...
  od
}
```

Algorithm 6. Mutator handshake

the order they were sent. It uses auxiliary local boolean arrays **reader[cpu, addr]** and **writer[cpu, addr]** to record whether it has seen a request to read (write) a given address by a given cpu, **have_fence[cpu]** to record if it has seen a **fence** from the cpu, and **any_access[cpu]** to record if it has seen any access from the given cpu. These are adequate to determine whether an operation is ready. The outline shows the handling for **load**. A more complex example is **load-acquire**, which is legal under the same conditions as **load**, but when it is ready also sets **have_fence[req.cpu]** to prevent later operations from being considered ready. The actual work is carried out in a final **d_step** that has a case for each operation. The **??** operator matches a particular message in the channel and receives (removes) it. This is non-blocking in this case because we just saw the message in the channel, and the whole loop iteration is in an **atomic**, preventing other processes from modifying the channel.

This pattern of a **d_step** to determine what work is available, a **select** to choose an available item, and a second **d_step** to do the chosen work is one we use repeatedly in our models. Conceivably the first **d_step** and the **select** could be implemented using a truly random (non-deterministic) message receive operator, but Promela does not provide that functionality—only receiving the first message that matches a given pattern.

There is an **inline** for each operation for a cpu to request that operation, and another one for it to receive the matching response, which involves waiting on its **memoryToHost** channel, as shown in Algorithm 2 for **RECEIVE_LOAD**. The **store** and **store-release** operations do not send responses. See Algorithm 14 in the appendix for details on how we incorporate TSO and SC into this Promela model.

7 MODELING THE GC ALGORITHMS IN PROMELA

We now describe our design of the MMTk work packet framework in Promela, and our models of the STW marking and SATB concurrent marking algorithms.

7.1 Assumptions

The main challenge we face is state space explosion. To manage this, we choose abstractions that simplify the original concepts, and that we hope do not affect correctness. The models follow these assumptions:

- *Weak fairness*: If a process is continuously able to take a step, it eventually will. While real schedulers can starve processes, we assume they cannot be starved forever.
- Atomicity of guard evaluation: A single instance of the releaser **proctype** evaluates all guards atomically, in a single **d_step**. Checking the actual implementation of a releaser is beyond the scope of this work.
- Atomicity of work polling: Recall that we are *not* trying to model details of the work pool and its synchronization, but assume that that infrastructure is correct.
- Order of work polling: This is unspecified in MMTk design, so we model it with non-determinism to check all possible orders.

The main differences between the model and the MMTk design are:

- A designated **proctype** for each kind of work packet: This is in contrast to the dynamic dispatching of the MMTk framework. Separate **proctype**s reduce state space explosion in model checking. Assuming there are at least as many workers as kinds of work, it is always *possible* that a given worker thread *happens* to handle a single kind of work. As explained in Section 8 we test scenarios with more than one worker of each given kind.
- One work unit per packet: While grouping work units into packets is important for *performance* of MMTk, we model the limiting case of packets of size one. Since work units from the same packet are assumed independent from each other, this change does not impact correctness. In fact, this actually challenges correctness *more*, since work units in the same packet cannot be processed concurrently in MMTk but can in model checking.
- One bucket per kind of work packet: MMTk allows a bucket to hold more than one kind of work, while our buckets hold a single kind. This is fully general considering that two buckets that open and close under certain conditions are equivalent to a single bucket that opens and closes under the same conditions. Our approach makes it a little easier to determine when a given work unit can be processed.

We do not model mutator stacks as part of the shared memory modeled by the **memory** module. Even though stack scanning is done by a separate process, the mutator is stopped when it happens, so there is no race condition. We also generally implement computations local to a mutator as atomic steps.

The shared heap of objects *is* kept in memory locations accessed via the **memory** module. Since our current models do not copy objects, non-pointer fields are ignored by our collectors, so we do not model them. Further, we believe that objects with two pointer fields are large enough to test the interesting cases. Therefore, our objects consist of a metadata field (for the mark bit) and two pointer fields, each stored in a distinct memory location.

We often abbreviate **scan_stack** to **ss**, **mark_obj** to **mo**, and **scan_obj** to **so** for brevity.

7.2 STW Marking

As described in Section 4, marking begins when the **startGC** bucket opens, followed later by the **scanStack** and then **mark** buckets. The **startGC** guard starts enabled by the releaser, and mutator 0 can request **startGC**, which allows the releaser to open that bucket. The **startGC** process is simple—it just requests **scanStack** for each mutator and then indicates it is done. That allows the releaser to open the **scanStack** bucket. This is slightly more interesting, with the **scanStack** worker and mutator engaging in a *handshake*. The mutator side is illustrated in Algorithm 6. Notice that the STW mutator does nothing other than respond to handshakes, and for mutator 0, kick off a collection. The **scanStack** worker follows the general worker pattern. Its declarations appear in Algorithm 15 and a sketch of its body in Algorithm 16 in the appendix. There can be at most one item for each mutator, so the work items are logically a set (in the mathematical sense) of mutators. We model this using a **typedef** with **bool** fields **requested** and **claimed** indexed by mutator number. The guard, **SS_ALLOWED**, starts **false**, and the counts of needed (requested and not yet done) and claimed items start at 0. **SS_REQUEST** makes the obvious adjustments, atomically. The bucket is empty when no more scans are needed.

As previously mentioned, the **mark** bucket comprises **markObj** and **scanObj** work units. Their declarations (Algorithm 7, Algorithm 8) are very similar to one another, but with an important difference: **scanObj** models a *set* of requests while **markObj** models a *multiset* of them, since there can be more than one request in process to mark a given object. Thus the **requested** and **claimed** fields of the work units of **markObj** are of type **byte** rather than **bool**. The relevant parts of the the workers appear in Algorithms 9 and 10. Note that any of **scanStack**, **markObj**, or **scanObj** can have more than one worker instance running concurrently.

The releaser is quite simple (Algorithm 4) and builds on the guard definitions (Algorithm 5).

7.3 SATB Marking

For snapshot-at-the-beginning concurrent marking, there are two extra components: we need the mutator to manipulate pointers on its stack so the model is interesting to check, and we need a proper write barrier and support for object allocation. For SATB we extend the mutator model previously shown so that it can take *steps*, except while yielding for stack scanning. This is shown in Algorithm 11. The variable **step** serves as a program counter for mutator work that is defined elsewhere, and **steps** indicates the total number of steps possible.[4] Algorithm 12 shows the example of allocating an object and storing the reference into a field of another object. This uses **alloc_obj** and **update_field**, which illustrate those primitives, including the write barrier (Algorithm 13). Note that the barrier and allocating marked happen only if the action occurs during marking (after stack scanning is allowed and before

[4] It would also be possible to write the mutator more sequentially, inserting allowed yield points, something we aim to try in the future to see how it affects the size of the search space.

```
typedef mo_packet {byte requests; byte claims;}
mo_packet MO_BUCKET[NOBJS];

bool MO_ALLOWED = false;
byte num_mo_needed = 0;
byte num_mo_unclaimed = 0;
...
inline REQUEST_MO (obj) {d_step {
    MO_BUCKET[obj].requests++;
    num_mo_needed++;
    num_mo_unclaimed++;
  }
}
# define MO_BUCKET_EMPTY() (num_mo_needed == 0)
```

Algorithm 7. Mark object declaration

```
typedef so_packet {bool requested; bool claimed;}
so_packet SO_BUCKET[NOBJS];

bool SO_ALLOWED = false;
byte num_so_needed = 0;
byte num_so_unclaimed = 0;
...
inline REQUEST_SO (obj) {d_step {
    SO_BUCKET[obj].requested = true;
    num_so_needed++;
    num_so_unclaimed++;
  }
}
# define SO_BUCKET_EMPTY() (num_so_needed == 0)
```

Algorithm 8. Scan object declaration

```
proctype mo_worker (...) {
  ...
  do
  :: MO_ALLOWED && MO_HAS_UNCLAIMED() → {
    // poll for a work unit, obj_id, non-deterministically; may do:
      MO_BUCKET[obj_id].claims++; // claim from multiset
      num_mo_unclaimed--;
    if // process work unit
    :: obj_id == HEAP_NULLPTR → skip;
    :: else → {
        ... // mark, may do a REQUEST_SO
        num_mo_needed--;
    }
    fi
  }
  :: MARK_DONE → break;
  od
}
```

Algorithm 9. Mark object worker

```
proctype so_worker (...) {
  ...
  do
  :: SO_ALLOWED && SO_HAS_UNCLAIMED() → {
    // poll for a work unit, obj_id, non-deterministically; may do:
        SO_BUCKET[obj_id].claimed = true; // claim from set
        num_so_unclaimed--;
    if // process work unit
    :: obj_id == HEAP_NULLPTR → skip;
    :: else → {
        ... // scan, may do one or more REQUEST_MO
        num_so_needed--;
    }
    fi
  }
  :: MARK_DONE → break;
  od
}
```

Algorithm 10. Scan object worker

```
proctype mutator (byte MUT_ID) {
  byte step = 0; byte steps;
  mut_steps(MUT_ID, steps);
  do
  ... // previous cases
  // If not yielded, there are more steps,
  // and stepping is allowed, we can take a step
  :: !MUT_YIELDED[MUT_ID] && step < steps &&
       MUT_STEP_ALLOWED() →
      mut_step(MUT_ID, step);
  // If not yielded, all steps done, and MARK_DONE, we can stop
  :: MARK_DONE && !MUT_YIELDED[MUT_ID] && step == steps →
     break;
  od
}
```

Algorithm 11. SATB mutator skeleton

```
inline mut_step (MUT_ID, step) {
  if
  :: MUT_ID == 0 && step == 0 → {
      alloc_obj(1, 1);
      step++;
    }
  :: MUT_ID == 0 && step == 1 → {
      update_field(DATA_OF(0), STACK_SLOT(MUT_ID, 1));
      step++;
    }
  :: MUT_ID == 0 && step == 2 → d_step {
      STACK_SLOT(MUT_ID, 1) = HEAP_NULLPTR;
      step++;
    }
  fi
}
```

Algorithm 12. SATB mutator steps: allocation

marking is done). Some other scenarios, described in Section 8 use other mutator steps.

In sum, key patterns that emerged from writing the example collectors in Promela to match the MMTk style are: the atomic **d_step-select-d_step** technique for choosing and processing work items, possibly non-deterministically, and the organization of work items into set or multisets, appropriately indexed.

8 EXPERIMENTS

We now describe the specific models and LTL formulas that we checked. It is first worth mentioning that we constructed 19 models with a total of 81 LTL formulas to check the semantics of the weakly ordered memory component. These were also checked under the TSO and SC models, where a small number of them intentionally behave differently (to test the TSO and SC models themselves). These all check in a matter of at most a few milliseconds since they have the style of small "litmus tests." The LTL formulas check both that certain outcomes of concurrent operations are possible and that certain others are disallowed.

Moving on to marking, we have 6 idle-mutator scenarios to check stack scanning and marking generally, detailed in Figure 1. These were designed to check the following conditions, some of

```
1  inline insure_at_least_gray (obj) {
2    if
3    :: obj ≠ HEAP_NULLPTR →
4       BEGIN_READ(META_OF(obj));
5       byte meta;
6       RECEIVE_READ(META_OF(obj), meta);
7       if
8       :: meta_is_unmarked(meta) → REQUEST_MARK_OBJ(obj);
9       :: else → skip;
10      fi
11   :: else → skip;
12   fi
13 }
14 inline update_field (addr, newval) {
15   if
16   :: DURING_MARKING →
17      byte oldval;
18      BEGIN_READ(addr);
19      RECEIVE_READ(addr, oldval);
20      insure_at_least_gray(oldval);
21   :: else → skip;
22   fi
23   WRITE(addr, newval);
24 }
25 inline alloc_obj (obj, stack_slot) {
26   if
27   :: DURING_MARKING → WRITE(META_OF(obj), MARKED);
28   :: else → skip;
29   fi
30   update_stack_slot(stack_slot, obj);
31 }
```

Algorithm 13. Update and allocation primitives

which are checked only when multiple threads of a given kind run concurrently:

- All stacks and all stack slots are scanned.
- All object fields are scanned.
- An unreachable object is not marked, while all reachable objects are marked.
- Shared graph structure (more than one incoming pointer) is handled.
- Multiple levels of objects are handled.
- Cycles with one object (self-loop) and with more than one are handled.
- Each of **scanStack**, **markObj**, and **scanObj** have occasions of (a) competition for a work unit, and (b) concurrent processing of different work units.

An idle mutator handshakes for stack scanning, but does no other work. Of the scenarios, 2 have more than one stack, and thus more than one mutator, the others only one mutator. We ran each scenario with one process each of kinds **scanStack**, **markObj**, and **scanObj**, and also with two **scanStack**, or with two **markObj**, or with two **scanObj** processes. Together this is 24 configurations of objects and processes.

All of these configurations test these two LTL formula schemes:

(1) **[] ((is_marked(i) -> [] is_marked(i)) && ...)** with a conjunct for each object that might become marked. This says that once an object is marked, it stays marked.
(2) **<> (memory_done && is_marked(i) && ...)** with a conjunct for each object that *should* become marked. This says that eventually the **memory** process should be done (the memory does not finish until all the processes that might access it finish) and the indicated objects should be marked.

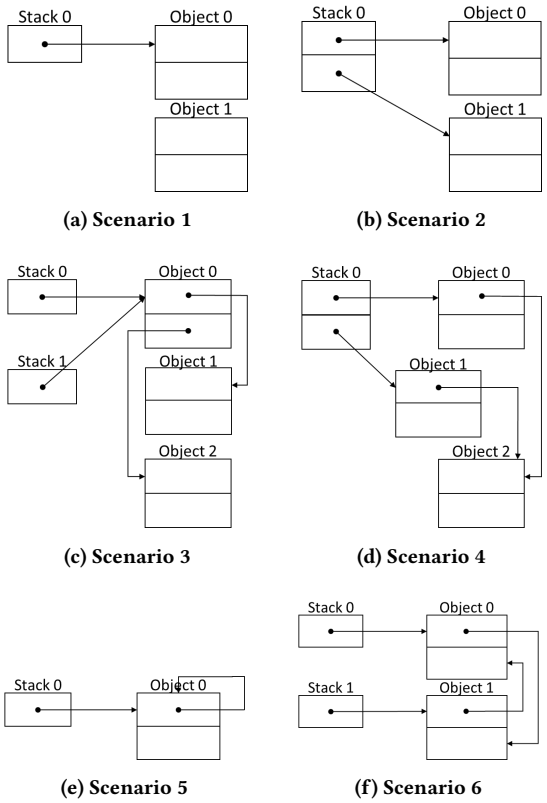
Additionally, Scenario 1 checks that Object 1 is *not* marked using LTL formula **[] is_unmarked(1)**. This claims that Object 1 is never marked. All of these formulas are ones that should succeed for the idle-mutator case. We end up with 52 model checking runs (2 LTL formulas times 24 configurations of numbers of processes, plus 4 tests that Object 1 is never marked) for the idle-mutator case. However, to this we add runs under the TSO and SC models, giving 156 runs.

Having idle mutators does not distinguish STW and SATB, or show anything about soundness of collection in the face of mutator activity. Therefore we devised three specific active-mutator scenarios:

(1) Starting from an initial configuration where the mutator stack holds references to objects 0 and 1, and object 0 has a reference to object 2, the mutator loads the reference to object 2, overwrites the reference from 0 to 2 with a null pointer, creates a reference from 1 to 2, and drops its immediate reference to 2. This is illustrated in Figure 2. We call this the *pointer shuffling* scenario, and it can cause SATB collection to fail in the absence of a write barrier. If object 1 is marked and scanned, then the mutator does the shuffle, and then object 0 is marked, the collector fails to mark object 2.
(2) Starting from an initial configuration where the mutator stack refers to object 0, which refers to object 1, the mutator deletes the edge from 0 to 1 by storing a null pointer into that field of object 0. We call this the *pointer deletion* scenario. It is problematic for SATB if the deletion happens during marking but before object 0 is scanned, since object 1 will not be marked even though it was reachable at the beginning of marking.
(3) Starting from an initial configuration where the mutator stack refers to object 0, allocate a new object, number 1, create a reference from 0 to 1 and drop the mutator's immediate reference to 1. In this case the mutator should itself mark object 1 if the allocation happens during marking. We call this the *allocate marked* scenario.

We use the same three LTL formulas as for STW Scenario 1, except we adjust the second one slightly for the pointer deletion and object allocation scenarios since whether Object 1 should be marked depends on when a particular mutator step occurred during marking.

To check STW active mutators, we added a build-time flag that, if set, prohibits the mutator from taking steps during marking. To demonstrate the necessity of the write barrier and allocate-marked behaviors for SATB but not STW, we ran both STW and SATB scenarios where those behaviors were turned off. We further checked the scenarios with the write barrier enabled all the time and with it enabled only during marking. Again, all these were checked also under the TSO and SC memory models. In sum, the active mutator cases were run in five different modes: SATB with write barrier only during marking, SATB with write barrier always on, SATB with write barrier disable, STW with write barrier, and STW without write barrier. Again, these were all run also with TSO and SC. This lead to $5 \times 3 \times 3 \times 3 = 135$ model checking runs.



(a) Scenario 1 (b) Scenario 2

(c) Scenario 3 (d) Scenario 4

(e) Scenario 5 (f) Scenario 6

Figure 1: Heap scenarios 1 through 6

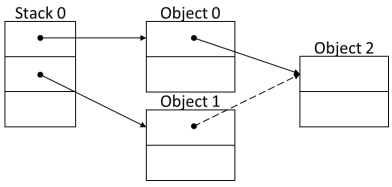


Figure 2: Pointer shuffling scenario

Lastly, we were curious to know whether a layered model checking approach might be feasible, where we first check certain operation in terms of their low-level memory accesses, and then if the operations are seen to be suitably atomic, switch to atomic versions of them in the larger algorithms. We introduced a build-time option that made all accesses to the metadata word of objects (the mark bit) atomic in Promela. We call this the *atomic meta* version. We added tests in the style of the memory module's "litmus tests" to verify that the metadata primitives, when written to use the memory module, are indeed atomic, and then tested the *atomic meta* versions of all the 156 + 135 = 291 previous runs, giving a total run count of 582.

We now offer some summary statistics. The 52 base STW tests take from under a second to over 5 hours with a geometric mean of 99 sec (for runs requiring at least 1 sec) and maximum of 19,200 sec; geometric mean of 1.7 million states explored (maximum 655 million); typical exploration rate of 120,000 states/sec without a huge range; and a geometric mean of 683 Mb of memory required (maximum 50 Gb). The total time for those runs was 83,334 sec (about 23 cpu-hrs). The TSO statistics for these were very similar while the SC ones were around 2/3, reflecting the fact that SC somewhat reduces the search space. The *atomic meta* cases were all noticeably smaller than their counterparts without *atomic meta*, but not by a factor of 2. The total time for all variants including *atomic meta* was about 93 cpu-hrs, and we were able to achieve significant parallelism on our larger servers so as to reduce wall clock time.

Turning to the SATB tests, the pointer shuffling scenario took particularly long with the third LTL requiring 100,000 sec (nearly 28 hours) to check using somewhat more that 250 Gb of memory. The second LTL check faster (over 14 hours) but required more memory (about 340 Gb). Using *atomic meta* cut this by more than 1/2. TSO reduced by only 10–20% and SC by about 50%. This is clearly a scenario where it would be good to cut its state space down if it can be done while preserving its interesting nature. It is worth mentioning that running it revealed limitation in our initial LTL formulae when, for example, the only copy of the pointer was in the Promela channel heading toward the memory—the mutator had not guaranteed that its write to memory was complete before it yielded and allowed the collector to scan its stack.

The other two SATB scenarios tested in seconds to a small number of minutes.

The version that used the write barrier all the time produced similar statistics, while the one with the write barrier disabled checked noticeably faster (since it only has to find one bad case to show the barrier is necessary, rather than showing that all cases are good).

The STW active mutator cases all completed within a few minutes.

In sum, if we neglect the 54 particularly large and slow pointer shuffling SATB runs, which took about 350 cpu-hrs, the remaining runs used less that 4 cpu-hrs. We conclude that, especially with access to servers with significant memory and computing capacity, most of the model checking can be done within a day, and deeper analysis of why certain runs took longer may inspire ways to write the tests to reduce their time and space requirements.

9 RESULTS

Here are the results of our experiments, beyond the statistics, previously mentioned. First, all LTL formulas checked properly, that is, all succeeded except those that should not have. SPIN's lossless compression mode (**COLLAPSE**) was adequate for full state space search to succeed in the memory budget available (250 GB, except for a handful of runs that required 300–350 GB); we did not need to resort to probabilistic checking.

Our modeling effort showed that we could exploit common patterns, paralleling those of the MMTk framework, to wit:

- For each work kind, a fairly standard pattern of declarations for the work units of that kind (as sets or multisets indexed by object number) and of counters for tracking requested, claimed, and completed work, that keep the SPIN state small and enable simple termination checks.
- Guards for each kind of work written along with the declarations for the work units.
- One Promela process kind for each work unit kind.

- The atomic **d_step-select-d_step** pattern for checking for and claiming work from the work pool.
- One memory process modeling the weak memory.
- One releaser process.

We further began to see some ways to structure the code using standard modules with call backs, as seen with the active mutator cases, which allows a fixed mutator module to model a range of mutator behaviors.

Another interesting result is that, even after all team members agreed that the write barrier looked correct, model checking revealed a flaw: we had turned the write barrier on only after the **mark** bucket was open, but it needed to be turned on as soon as **scanStack** was open, since **scanStack** is really part of the overall marking process. SPIN's ability to show a failing execution's trace was very helpful in tracking this down and fixing it. We had several other similar experiences in the course of this work.

10 CONCLUSION AND FUTURE WORK

We consider our exploration of a possible model checking framework for the MMTk work packet framework to be a success. We were able to develop models with a significant degree of repeated patterns and structures. A key goal for this prototype was to determine whether a domain-specific language (DSL) for writing collector components for the framework, or at least model components, would be a reasonable undertaking. We now believe that it would be. Considering what such a language and its processor might be like is a promising direction for future work.

Another key goal was to see if developing and checking models *early* might positively influence the design of the MMTk framework. Early reactions from the MMTk developers confirmed helpful impact of our model checking effort.

It would be good to expand the diversity of GC algorithms within the modeling framework we have now. We believe there are more commonalities and patterns we can extract before designing a DSL. Supporting copying collection is an obvious direction. However, we think that there are ways the models can be made more modular using more hooks and callbacks. Read and write barriers are one example of that. However, there are some instances where hooks for what we might call instrumentation are also useful because in model checking we may need information in LTL formulas that is not readily apparent from the system state. An example of that is whether a given pointer is over-written and under what conditions (e.g., during marking or not). The **memory** process could offer a callback when it processes an operation that would be helpful with that.

There also were clear cases where we need to try to control state space explosion better. Writing the active mutator code more as a sequence with allowed yield points marked might be better than our current encoding as a state machine, though whether it would have much impact is not clear.

Expanding our modeling coverage to multiple full cycles of collection would clearly be interesting. Also, at present we offer a single (though definitely weakly ordered) memory model. A rather different, but still very useful, project would be model checking the MMTk work pool data structures in detail, operating over weakly ordered memory.

Our final conclusion is that model checking is a good fit with concurrent GC and should go hand in hand in hand with developing future concurrent memory managers and garbage collectors. There appear to be good prospects for an approach that would generate both models and implementation code fragments from a DSL designed for that purpose.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2018372 and Grant No. 1909731. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

A APPENDIX

We support the TSO (total store order) and SC (sequentially consistent) memory models via C macro flags MM_TSO and MM_SEQ_CST (at most one of which may be set), in addition to the relaxed-plus-acquire-release model, as shown in Algorithm 14. We mark the changes with **.

```
1  do
2  :: nempty(hostToMemory) →
3     atomic {
4       d_step {
5  # if MM_TSO // **
6        bool mem_anywrites[CPUS];
7  # endif
8        // clear data structures (not shown)
9        for (req in hostToMemory) {
10         if
11         :: req.op == m_load →
12            if
13            :: writer[req.cpu,req.addr] ||
14               have_fence[req.cpu] ||
15               (MM_SEQ_CST && mem_anyaccess[req.cpu]) →
16               skip; // **
17            :: else →
18               ready[nready] = req; nready++;
19               reader[req.cpu,req.addr] = true;
20               any_access[req.cpu] = true;
21         :: req.op == m_write →
22            if
23            :: reader[req.cpu,req.addr] ||
24               writer[req.cpu,req.addr] || have_fence[req.cpu] ||
25 # if MM_TSO // **
26               mem_anywrites[req.cpu] ||
27 # endif
28               (MM_SEQ_CST && mem_anyaccess[req.cpu] →
29               skip; // **
30         ... // cases for other operations
31         fi
32       }
33     }
34     ...
```

Algorithm 14. Memory process outline

The declarations for **scanStack** follow those for a set (not multiset) of requests.

```
typedef ss_item {bool requested; bool claimed;}
ss_item SS_BUCKET[NUM_MUTATORS];

bool SS_ALLOWED = false;
byte num_ss_needed = 0;
byte num_ss_unclaimed = 0;

inline REQUEST_SS (mut_id) {d_step {
    SS_BUCKET[mut_id].requested = true;
    num_ss_needed++; num_ss_unclaimed++;
  }
}
# define ALL_STACKS_SCANNED() (num_ss_needed == 0)
```

Algorithm 15. Scan stack declarations

The polling for **scanStack** atomically chooses a requested but not claimed item, if there is one, putting its mutator number into **m**, using the atomic **d_step-select-d_step** pattern.

```
proctype ss_worker (...) {
  do
  :: SS_ALLOWED && SS_HAS_UNCLAIMED() → {
    atomic {
      byte ssw_count = 0;
      byte ssw_mapping[NUM_MUTATORS];
      d_step {
        // find all requested but not claimed packets
        for (ssw_i : 0 .. NUM_MUTATORS−1) {
          if
          :: SS_BUCKET[ssw_i].requested &&
              !SS_BUCKET[ssw_i].claimed →
              ssw_mapping[ssw_count] = ssw_i;
              ssw_count++;
          :: else → skip;
          fi
        }
      }
      if // choose a packet, if any
      :: ssw_count > 0 →
         select(ssw_i : 0 .. ssw_count−1);
         d_step {
           m = ssw_mapping[ssw_i];
           SS_BUCKET[m].claimed = true;
           num_ss_unclaimed--;
         }
      :: else → skip;
      fi
    } // end atomic polling
    if
    :: m < NUM_MUTATORS →
      do
      :: !MUT_YIELD_REQUESTED[m] → // request yield
        MUT_YIELD_REQUESTED[m] = true;
      :: MUT_YIELD_REQUESTED[m] && MUT_YIELDED[m] → {
        ... // do REQUEST_MO for non-null stack entries
        d_step {
          MUT_YIELD_REQUESTED[m] = false;
          num_ss_needed--;
        }
        break;
      }
      od;
    :: else → skip;
    fi
  }
  :: SS_ALLOWED && !SS_HAS_UNCLAIMED() → break;
  od
}
```

Algorithm 16. Scan stack worker

REFERENCES

[1] Tatsuya Abe, Tomoharu Ugawa, Toshiyuki Maeda1, and Kousuke Matsumoto. 2016. Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models. arXiv:1608.05893.

[2] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering*. IEEE Computer Society Press, Edinburgh, 137–146. https://doi.org/10.1109/ICSE.2004.1317436

[3] Howard Bowman, John Derrick, and Richard E. Jones. 1993. Modelling Garbage Collection Algorithms. In *International Workshop on Concurrency in Computational Logic, City University, London, 13 December 1993*.

[4] The Chromium Project. 2022. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/

[5] The Coq Development Team. 2017. *The Coq Proof Assistant, version 8.7.0*. https://doi.org/10.5281/zenodo.1028037

[6] Adam Sandberg Ericsson, Magnus O. Myreen, , and Johannes øAman Pohjola. 2017. A Verified Generational Garbage Collector for CakeML, See [14].

[7] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning (JAR)* 63 (2019). https://doi.org/10.1007/s10817-018-9487-z

[8] Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 125–137. https://doi.org/10.1145/3018610.3018621

[9] Peter Gammie, Antony L. Hosking, and Kai Engelhardt. 2015. Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Portland, OR. https://doi.org/10.1145/2737924.2738006

[10] Charles Antony Richard Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21 (1978), 666–677.

[11] HOL4 development team. 2022. HOL4 web site. https://hol-theorem-prover.org/

[12] Gerard J. Holzmann. 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

[13] 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, Palo Alto, CA, 48–57. https://doi.org/10.1145/376656.376810

[14] ITP 2017. *8th International Conference on Interactive Theorem Proving (ITP)*. IEEE Press, Brasília, Brazil.

[15] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming (ICFP)*. ACM Press, 60–73. https://doi.org/10.1145/2951913.2951924

[16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), E2. https://doi.org/10.1017/S0956796818000229

[17] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

[18] MMTk development team. 2022. MMTk web site. https://mmtk.io

[19] Man Yue Mo. 2021. Chrome in-the-wild bug analysis: CVE-2021-37975. https://securitylab.github.com/research/in_the_wild_chrome_cve_2021_37975/

[20] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[21] Tomoharu Ugawa, Tatsuya Abe, and Toshiyuki Maeda. 2017. Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Vancouver, 26. https://doi.org/10.1145/3133877

[22] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. 2018. Transactional Sapphire: Lessons in High-Performance, On-the-fly Garbage Collection. *ACM Transactions on Programming Languages and Systems* 40, 4, Article 15 (Dec. 2018), 56 pages. https://doi.org/10.1145/3226225

[23] Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. 2007. CGCExplorer: A Semi-Automated Search Procedure for Provably Correct Concurrent Collectors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices 42(6))*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM Press, San Diego, CA, 456–467. https://doi.org/10.1145/1250734.1250787

[24] Taiichi Yuasa. 1990. Real-Time Garbage Collection on General-Purpose Machines. *Journal of Systems and Software* 11, 3 (March 1990), 181–198. https://doi.org/10.1016/0164-1212(90)90084-Y

[25] Yannick Zakowski, David Cachera, Delphine Demange, Gustavo Petri, David Pichardie, Suresh Jagannathan, and Jan Vitek. 2017. Verifying a Concurrent Garbage Collector using a Rely-Guarantee Methodology, See [14], 496–513. https://doi.org/10.1007/978-3-319-66107-0_31