Portland State University

PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

6-2009

Operational Verification of a Relativistic Program

Robert T. Bauer
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

 Part of the OS and Networks Commons, and the Programming Languages and Compilers Commons

Let us know how access to this document benefits you.

Citation Details

Bauer, Robert T., "Operational Verification of a Relativistic Program" (2009). *Computer Science Faculty Publications and Presentations*. 214.
https://pdxscholar.library.pdx.edu/compsci_fac/214

This Technical Report is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Operational Verification of a Relativistic Program

Robert T. Bauer

Portland State University
Portland, Oregon

26 June 2009

TR–09–04

Abstract. Engineering efforts to achieve scalable multiprocessor performance for concurrent reader-writer programs have resulted in a family of algorithms that are non-blocking and that tolerate interprocessor interference. Because these algorithms accept a unique frame of reference for each processor's accesses to memory, they typify a concurrent programming technique for shared memory multicore architectures called relativistic programmming.

Rigorous verification of these algorithms is not possible with existing semantic based approaches because the semantics under approximates multiprocessor behavior and the algorithms rely on abstruse interactions with the operating system that aren't reconciled with language semantics.

The Read-Copy Update (RCU) algorithm is the protypical example of relativisitc programming; it is used in more than 2000 places in the Linux kernel and has thus far resisted analysis. In this paper a simple language for a sequentially consistent multiprocessor is defined and we implement RCU in that language. Both the RCU implementation and the language semantics are instrumented to prove that RCU does not collect live objects and that it is memory safe; restrictions on the definition and use of local RCU pointers that eliminate the instrumentation are introduced. Thus, RCU implementations that conform to these restrictions do not collect live objects and are memory safe.

These restrictions are readily accommodated by program analysis tools to certify RCU implementations.

1 Introduction

Engineering efforts to achieve scalable multiprocessor performance for concurrent reader-writer programs have resulted in a family of algorithms that are non-blocking and that tolerate interprocessor interference[28]. Because these algorithms accept a unique frame of reference for each processor's accesses to memory, they typify a concurrent programming technique for shared memory multicore architectures called relativistic programmming[16].

The Read-Copy Update (RCU) algorithm[24–26] is the protypical example of relativisitc programming; it is used in more than 2000 places in the Linux kernel[27] and achieves near optimal multiprocessor scalability by allowing concurrent accessors and a mutator to access possibly different versions of a data object. Despite RCU's growing importance, verification of RCU use and implementation is almost entirely based on testing[23]. This hinders RCU use in trustworthy kernels and critical mission software.

RCU has resisted rigorous analysis for at least three reasons: 1) Traditional semantic based approaches for reasoning about program correctness under approximates modern processor-memory systems and tend to focus on data race free programs[43] — RCU is a racey program; 2) RCU relies on abstruse operating system interactions that aren't reconciled with language semantics; and 3) Behavioral specifications for RCU protected data structures have not been established. Each of these issues represents a relatively formidable challenge to the existing state-of-the-art.

At this point in time, a proof of correctness for a sophisticated implementation of RCU executing on a modern processor-memory system is not feasible. However, as a first step towards that goal, this work tackles proving that an unsophisticated variant – RCU "Classic" – executing on a sequentially consistent multiprocessor that supports multitasking does not collect live objects and is memory safe.

Our approach to prove that RCU "Classic" has the live object and memory safety properties is to define a minimal language capable of expressing RCU. This language incorporates context switch behavior and is defined using a small-step operational semantics that maps language constructs to configuration transformations. We express the live object and memory safety properties in linear temporal logic and use the set of traces comprised of sequences of configurations that are created by simulated execution of the RCU program via the operational semantics as models to determine the validity of these properties. We instrument both the code and the language semantics to simplify the verification effort — we prove that instrumented RCU executing instrumented semantics does not collect live objects and is memory safe. We then introduce restrictions on the definition and use of the local RCU pointers that allow us to eliminate the instrumentation; thus, RCU implementations that conform to these restrictions do not collect live objects and are memory safe.

Our work advances the state-of-the-art by capturing a facet of the real behavior of an operating system context switch in a way that exposes the intermediate state of a system where one thread has been switched out and the next is yet to be switched in. We show that this intermediate state is used by RCU to achieve synchronization without the use of synchronization primitives.

We formalize restrictions for the definition and use of the RCU local pointer. These restrictions can help certify RCU implementations and are readily accommodated by static analysis tools.

This work demonstrates that operational verification techniques are useful for proving liveness and safety properties of concurrent, reactive programs. We give, for the first time, proofs that RCU does not collect live objects and that it is memory safe.

This work is organized as follows. Section 2 gives the background for this work and describes our overall approach. Section 3 describes related work. Section 4 defines the machine abstraction for a sequentially consistent multiprocessor that supports cooperative multitasking. The machine language syntax is given in Section 5. The semantics are given in Section 6. RCU "Classic" is described in Section 7. The properties are verified in Section 8. Section 9 gives our conclusions and Section 10 describes future work.

2 Background

Mutual exclusion (locking) is the standard technique to prevent concurrent programs from corrupting shared memory structures[2, 46]. When access to the shared data structure can be partitioned into reader and writer accesses (readers don't modify the data), reader-writer locks reduce latency by allowing concurrent readers and enforcing mutual exclusion only when a writer is accessing the shared data structure[31]. To further improve performance, RCU allows a writer to execute concurrently with possibly many readers; mutual exclusion is enforced only on writers (i.e., at most one writer is executing at any given time).

Reader-writer mutual exclusion guarantees that concurrent programs always access data in a consistent state — readers cannot acquire an object in some partially updated state and writers cannot partially overwrite the effects of another write. Owicki[36] showed that reasoning about such systems is straight forward because the intermediate effects of a multistep update are invisible — she showed that we can treat the critical region as "non-concurrent" code whose effects are atomic and visible only after the writer exits its critical region. For example, updating an object using reader-writer mutual exclusion is accomplished either by updating the object in place or by creating a new object, copying the existing data, and then updating the new object. Since the update is done within a critical region, it's not possible that any reader (or writer) is using the object being updated; thus, in the case where the old object is replaced by a new object, the old object's storage can be collected once the existing data is copied to the new object.

RCU also guarantees that concurrent programs always access data in a consistent state. Writers cannot partially overwrite the effects of another write and readers cannot acquire an object in some partially updated state. However, because RCU doesn't use mutual exclusion, the intermediate effects of a multistep update are visible. To prevent corruption, RCU updates an object by creating a new (local) object, copying the existing data, updating the new object, and then atomically storing a single pointer to simultaneously make the new object accessible and the old one inaccessible. After the pointer update, there are at least two versions of the object — the new one and the old one. Storage for the old object cannot always be reclaimed immediately after the pointer update because some reader may be using it.

An object's storage is safe to reclaim only when the object is no longer live (informally, not being used and inaccessible); RCU "Classic" claims it is safe to reclaim the old object's storage after the pointer is updated and every reader has executed a context switch. This seems like a very easy claim to prove; however, traditional programming language semantics treat a context switch as a "skip" — the semantics abstract away from the behavior of the scheduler by treating yields to the kernel as semantic no-ops[37, 38]. In contrast, RCU treats these yields as critical rendezvous points between processes and avoids adding synchronization primitives by piggy-backing on synchronization actions in the kernel. Our semantics captures this and lets us forumlate the claim *after the pointer is updated and every reader has executed a context switch* as a predicate on the assignment and rendezvous actions.

An operating system context switch suspends the execution of a process and resumes the execution of one that had previously been suspended[4]. In a multiprocessor, a suspended process resumes on any available processor unless the suspended process is tagged to run on a specific one[3]. The RCU writer program is constructed so that after the pointer update, it runs on each processor before collecting the old object's storage. How it does this is straight forward. Using an operating system primitive, the RCU writer's execution is suspended on one processor and resumed another; our operational semantics captures this with the `resched` instruction.

For example, the RCU writer on a n processor system executes:

```
Update Pointer

i <- current processor
resched 0
resched 1
...
resched n-1
resched i

Collect Old Object
```

The sequence of `resched` instructions direct the RCU writer to execute first on Processor 0 (`resched 0`), then on Processor 1 (`resched 1`), etc. After executing on the "last" processor, the RCU writer continues execution on the initial processor and collects the old object's storage.

How does this ensure that it safe to collect the old object? Consider what happens when `resched` j is executed. The RCU reader executing on Processor j–1 (`resched` j executes on Processor j–1) must have executed a `resched` for otherwise no context switch is possible. The RCU guidelines tell us that RCU readers are not allowed to hold a reference to an RCU protected object beyond the RCU read-side critical section — and the read-side critical section must end before the context switch. This effectively says that from the perspective of RCU reader executing on Processor j–1, the old object is not live when the RCU writer executes `resched` j on Processor j–1. By simple induction, when the RCU writer is ready to collect the old object's storage it is the case that for every RCU reader, the old object is "effectively" not live and it is safe to collect the old object's storage.

Note that RCU does not guarantee that the old object is not live — RCU is warranting that if a particular coding rule is followed, the old object won't be live at the time its storage is collected. However, the published implementation of RCU "Classic" shows that the old object can be live at the time it is collected. Yet, in looking at the published implementation, one gets the intuition that it is "safe" to collect the old object.

Our research is meant to replace informal arguments that RCU has certain properties with a formal proof. A formal proof that RCU has particular properties gives us confidence that it actually has those properties and it tells us explicitly what set of coding rules need to be followed to have this confidence.

This research focuses on proving that RCU doesn't collect live objects (live object safety) and the dual property that RCU doesn't access objects that have been collected (memory safety).

2.1 Approach

Our approach to proving the live object and memory safety properties is an operational verification using pencil and paper proofs. We use pencil and paper proofs because mechanizing a complex proof is a formidable challenge in-and-of-itself and is usually predicated on the availability of the pencil and paper proof.

An operational verification that program P has property S requires three things:

- An operational semantics of the implemenation language used to express program P
- A formal specification language in which to express S
- A deductive apparatus capable of carrying out the proof that $P \Rightarrow S$.

The remainder of this section describes our approach for proving the live object and memory safety properties vis-a-vis our operational semantics, specification of the properties, and the deductive apparatus.

Operational Semantics of Implementation Language

The operational semantics[38, 39] of a language describes the execution of programs in terms of sequences of configurations of an abstract machine. Our abstract machine is meant to be like a real multiprocessor, but with the minimum machinery necessary to explain the semantic aspects of the instructions used to implement RCU "Classic." The abstract machine we define is quite simple. Each processor has registers and a ready list. All processors share a common memory. Registers have the same type as memory, except that they are local to a processor and named differently so as to distinguish them from memory. A processor's ready list maintains processes that are eligble to run when the processor executes a `resched` instruction.

In our semantics, processors sequentially execute their programs and each memory operation is completed before the next is initiated. Concurrency is represented as a non-determistic choice of processor execution. Thus, our semantics describe a sequentially consistent multiprocessor[18] — sequential consistency is a global property where all memory accesses represent an interleaving that is consistent with the program order of each processor.

Our abstract machine differs little from modern sequential processors (e.g., ARM, Z80, 8096, etc.) and our semantics is faithful to them as well. Indeed, our semantics is faithful to any multiprocessor comprised of processors that execute instructions in program order and that serialize access to memory. The set of execution sequences realized by our operational semantics for any given program is compatible with the execution sequence realized by any real sequentially consistent multiprocessor.

The difference between our abstract machine and a real sequentially consistent multiprocessor is in our support of process rescheduling. Many processors and most modern ones provide support for context switching while depending on the operating system to determine the "next" thread that executes. Our semantics represents the scheduling and context switch behavior in the `resched` instruction and the information needed to effect a context switch is carried about in a processor's ready list. Processes in a real system are interrupted (either by preemption or voluntarily) so that the operating system can switch from one process to another; the intermediate state where one process is switched out and the next is yet to be switched in is captured in our semantics as the R state. If the ready list is empty when a processor executes a `resched` instruction, the processor stays in the R state until the ready list is loaded with a process. In a real operating system, if the ready list is empty, the processor executes a null process.

Whereas traditional semantic approaches for non-deterministic program execution treat a context switch as a "skip"[35], our semantics includes the processor

ready lists and the operating system intermediate state in the abstract machine's configuration. For this reason, the set of configuration sequences that define RCU as determined by our operational semantics closely approximates the real behavior of a sequentially consistent processor executing RCU.

Specification of Properties

Reactive programs, such as RCU, maintain an ongoing interaction with each other (concurrency) and/or their environment rather than produce a final value on termination[21, 22]. Reasoning about reactive programs requires specifying and verifying behaviors[40]. Temporal logic[6] is a standard formalism for expressing the behavior of reactive systems[19].

Programs are traditionally specified with linear temporal logic (LTL)[17] because the "interleaving" model of concurrency quite naturally corresponds with describing a computation as a set of executions and because the model-theoretic semantics of temporal logic relate formulae in the logic to sets of models that satisfy them.

The models in linear temporal logic take the form of sequences. Each configuration sequence in the set of executions determined by our operational semantics is a model; this establishes a direct relation relation between programs (and in particular our implementation of RCU) and the temporal logic formulae used to specify properties.

Deductive Apparatus

Our deductive apparatus is first order linear temporal logic. Specifications are written in linear temporal logic and the configuration sequences obtained by applying our operational semantics to the implementation of RCU "Classic" provide models for the verification effort.

The verificaton that RCU has a particular property is carried out as follows. Let $\Pi_P = \{\pi_0, \pi_1, \ldots\}$ be the set of execution sequences that define P. Let S be the specification of the property being verified, then $\forall \pi \in \Pi_P \ \pi \models S \vdash P \Rightarrow S$.

3 Related Work

Liu and Moore[20] have applied operational techniques to the verification of Java programs. Their approach attacks verification at the byte code level and does not consider liveness and safety properties. They prove correctness of two computational programs.

Atalli[1] uses operational techniques to create formal executable semantics of Java; she mixes small-step semantics to describe concurrency with big-step semantics to describe Java object-oriented features. Meseguer[32] defines concurrency in terms of rewriting and relates the execution of non-deterministic operational semantics to rewriting.

Schacht[45] creates a framework for reasoning about the temporal properties of Actor programs by defining the Actor concurrency model in terms of an operational semantics. The operational semantics are used to create traces (of configurations) that serve as models for validating linear temporal logic property specifications.

Compton[5] uses operational techniques to verify a safety property of Stenning's (UDP) algorithm. He uses a fragment of the Ocaml language (miniCaml), including the sockets library for UDP to demonstrate the verification in a model that accurately represents the operating system API and network environment in which the program would execute. Compton creates an operational semantics such that the network emits a label for every action performed and a list of these these labels forms a trace. He shows that each step of a program's execution corresponds to some step in the network trace. His safety theorem is given as a property for each trace.

Ridge[42] uses operational reasoning techniques to prove correctness of Peterson's algorithm on sequential and weak memory systems. He creates a small-step semantics for a functional language and implements Peterson's algorithm in that language. He uses the theorem prover Isabelle to symbolically execute the program. He verifies that the two processes cannot be in the critical section at the same time. His approach is to annotate trace information obtained from execution of the operational semantics so that it's easy to determine when execution for a thread is in a critical section.

Ridge approaches weak memory as follows. He maintains a history of writes and updates a memory location from this history when a memory barrier for that location is executed. He does not define a read — his proof relies on the side-condition that a memory location has only one value at any time.

Mckenney[29, 30] has used the SPIN Model Checker[15] to verify a variety of RCU implementation; however, he does not tackle liveness and safety properties.

4 Multiprocessor Abstraction

The operational semantics of a language describe the execution of programs in terms of sequences of configurations of an abstract machine[38, 39]. Our abstract machine is much simpler than a real processor as we support only those instructions necessary to create an implementation of RCU "Classic". Our abstract machine supports the basic load and store operations (load and store registers and immediate values), a load indirect operation, a resched instruction, and a halt instruction; we use some syntactical sugar to implement looping. Our abstract machine does not include a program counter, stack, machine status register, interrupts, etc. Memory operations are serialized.

The basic operations (load, store, load indirect) and looping construction are straight-forward. Load instructions put values in registers, store operations put values into memory. The value returned by an indirect operation isn't used in

our implemenation of RCU "Classic" because we don't really use RCU as part of a larger solution. For this reason, our semantics define a fail-stop semantics for dereferencing collected memory; to prove prove that RCU doesn't collect live objects and is memory safe requires showing that null pointers aren't dereferenced and that freed memory isn't accessed. To this end, we provide two semantics for dereferenced operations. The processor semantics for a load indirect implement a "skip" (the returned value isn't used) while the instrumented semantics indicate indicate an error if a null pointer is accessed or if freed memory is accessed[1]. These two semantics agree on computations that don't fail (we prove this in Section 8).

The resched instruction moves the sequence of instructions currently executing on a processor and the processor's registers to a ready list. This sequence of instructions and register values is called a context. Next, the resched instruction arbitrarily chooses a context from the processors ready list. The instruction sequence of the selected context becomes the currently executing instruction sequence and the processor registers are updated with the register values of the selected context. Once selected, the context is removed from the ready list.

A context switch occurs only when a processor executes a resched instruction. At that time, the program can reschedule itself to run on the other processor or the current processor. If a program A running on processor 1 executes a resched, placing program A on processor 0's ready list and if processor 1's ready list is empty, then after the processor 1's instruction stream and register values are placed in processor 0's ready list processor 1 will be idle until another processor executes a resched that puts a context in processor 1's ready list.

This behavior abstracts how processors execute a context switch and how an operating system dispatches processes to run on a particular processor. We now give a formal description of the multiprocessor abstraction; its language syntax and semantics are given in subsequent sections.

The multiprocessor has two processors, a common memory, a ReadyList for processor 0, and one for processor 1. Processors have local registers that are managed as an array. For example $regs(0)$ refers to register 0. Registers map a location to an integer value.

Multiprocessor = Processor0 × Processor1 × Mem × ReadyList0 × ReadyList1
Processor0 = regs
Processor1 = regs
regs = Array of val indexed by N Range(0 .. maxReg-1)

Mem maps locations to integer values. For example, $Mem(20)$ refers to memory location 20.

Mem = Array of val indexed by N Range(0 .. maxMem-1)

[1] Accessing previously freed or reallocated memory generally doesn't raise an exception on a real processor but the effect of using such memory is corruption that leads to data inconsistency and program crashes.

Ready lists maintain the "context" for the execution of processes that aren't currently running. The context for a process's execution are the values of the processor's registers and a list of instructions to be sequentially executed.

ReadyList0 = {(regs, cmds)}
ReadyList1 = {(regs, cmds)}

Values are integers:

val $\in \mathcal{N}$ $\quad 0 \leq val \leq maxInt$

The multiprocessor's instructions:

cmds = List of cmd
cmd = load + loadi + store + storei + dr + di + resched + forever + halt

$\sigma \in$ Processor0 $\times$ Processor1 $\times$ Mem $\times$ ReadyList0 $\times$ ReadyList1

5 Syntax

5.1 Syntax of Processor Instruction Set

```
cmd ::= 'load'   reg loc        // reg <-- contents of loc (r0 := [0])
      | 'loadi'  reg val        // reg <-- val             (r0 := 2)
      | 'store'  loc reg        // loc <-- contents of reg ([0] := r0)
      | 'storei' loc val        // loc <-- val             ([0] := 3)
      | 'dr'     reg            // dereference reg
      | 'di'     loc            // dereference loc
      | 'resched' procId        // context switch; continue execution on processor procId
      | 'forever' cmds          // loop
      | 'halt'                  // terminate execution

reg ::=  digits
loc ::=  digits
val ::=  digits | 'Nil'         // 'Nil' is "instrumentation" discussed later
procId ::= digits

digits ::= digits digit
         | digit

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

5.2 Program Syntax

A multiprocessor program consists of two sequences of commands separated by a ||. Commands to the left of || execute on processor 0 while those on the right execute on processor 1.

```
mp ::= cmds || cmds

cmds ::= cmds ';' cmd
       | cmd
```

6 Semantics

The small step semantics in this section formally describe the execution of multiprocessor programs as a transition system over configurations.

There are three forms of configurations:

$\langle c, \sigma \rangle$: A pair (c, σ) with command $c \in cmd$ and "state" σ.

$\langle c_0 \parallel c_1, \sigma \rangle$: A triple (c_0, c_1, σ) with Processor 0's continuation, $c_0 \in cmds$; Processor 1's continuation, $c_1 \in cmds$; and the "state", σ.

The third form is either of these two forms where the command is replaced with R — a context switch instruction (`resched`) transitions the configuration to an intermediate configuration whose continuation is R.

The state (as defined earlier) is $\sigma \in Processor_0 \times Processor_1 \times ReadyList_0 \times ReadyList_1 \times Mem$.

The semantics of the `dr` and `di` instructions have two forms: Instrumented and Processor. The instrumented semantics for these instructions gives transitions that can lead to an error configuration. The processor semantics for these instructions do not have such transitions.

To support the instrumented semantics, we have instrumented the processor instructions so that `val` can be a number (`digits`) or `Nil`.

6.1 Commands

The following subsections give the semantics for each command. To keeps things short, whenever there are processor specific differences in the semantics, we give the semantics with respect to Processor 0.

`load reg loc`

Register `reg` is assigned the contents of memory location `loc`.

$$\frac{}{\langle \texttt{load reg loc}, \sigma \rangle \rightarrow \sigma[\texttt{Processor0.regs(reg)} = \texttt{Mem(loc)}]}$$

`loadi reg val`

Register `reg` is assigned `val`.

$$\frac{}{\langle \texttt{loadi reg val}, \sigma\rangle \rightarrow \sigma[\texttt{Processor0.regs(reg)} = \texttt{val}]}$$

`store loc reg`

Memory location `loc` is assigned the contents of register `reg`.

$$\frac{}{\langle \texttt{store loc reg}, \sigma\rangle \rightarrow \sigma[\texttt{Mem(loc)} = \texttt{Processor0.regs(reg)}]}$$

`storei loc val`

Memory location `loc` is assigned `val`.

$$\frac{}{\langle \texttt{storei loc val}, \sigma\rangle \rightarrow \sigma[\texttt{Mem(loc)} = \texttt{val}]}$$

`dr reg`

Dereference memory location pointed to by `reg`.

Instrumented Semantics:

$$\frac{\sigma.\texttt{Processor}_0\texttt{.Regs(reg)} = \texttt{Nil}}{\langle \texttt{dr reg}, \sigma\rangle \rightarrow \langle \texttt{Error}, \sigma\rangle}$$

$$\frac{\sigma.\texttt{Processor}_0\texttt{.Regs(reg)} \neq Nil \quad \texttt{isFree(Processor}_0\texttt{.Regs(reg))}}{\langle \texttt{dr reg}, \sigma\rangle \rightarrow \langle \texttt{Error}, \sigma\rangle}$$

$$\frac{\sigma.\texttt{Processor}_0\texttt{.Regs(reg)} \neq Nil \quad \neg\,\texttt{isFree(Processor}_0\texttt{.Regs(reg))}}{\langle \texttt{dr reg}, \sigma\rangle \rightarrow \sigma}$$

Processor (uninstrumented) Semantics

$$\frac{}{\langle \texttt{dr reg}, \sigma\rangle \rightarrow \sigma}$$

Note: The auxiliary predicate `isFree` is discussed at the end of this section.

`di loc`

Dereference memory location pointed to by `loc`.

Instrumented Semantics

$$\frac{\sigma.\texttt{Mem(loc)} = \texttt{Nil}}{\langle \texttt{di loc}, \sigma\rangle \rightarrow \langle \texttt{Error}, \sigma\rangle}$$

$$\frac{\sigma.\texttt{Mem(loc)} \neq \texttt{Nil} \quad \texttt{isFree(Mem(loc))}}{\langle \texttt{di loc}, \sigma\rangle \rightarrow \langle \texttt{Error}, \sigma\rangle}$$

$$\frac{\sigma.\texttt{Mem(loc)} \neq \texttt{Nil} \quad \neg\, \texttt{isFree(Mem(loc))}}{\langle \texttt{di loc}, \sigma\rangle \rightarrow \sigma}$$

Processor (uninstrumented) Semantics

$$\frac{}{\langle \texttt{di loc}, \sigma\rangle \rightarrow \sigma}$$

Note: The auxiliary predicate `isFree` is discussed at the end of this section.

`Error`

Terminate execution.

$$\frac{}{\langle \texttt{Error}, \sigma\rangle \rightarrow \langle \texttt{Error}, \sigma\rangle}$$

`halt`

Terminate execution.

$$\frac{}{\langle \texttt{halt}, \sigma\rangle \rightarrow \langle \texttt{halt}, \sigma\rangle}$$

`forever`

Execute an instruction or sequence of instructions over and over again.

$$\frac{}{\langle \texttt{forever c}, \sigma\rangle \rightarrow \langle \texttt{c}\,;\, \texttt{forever c}, \sigma\rangle}$$

`resched` n

The sequence of instructions following `resched` are to be executed on processor n.

$$\texttt{resched 0}\ \frac{}{\langle \texttt{resched 0; c}, \sigma\rangle \rightarrow \langle \texttt{R}, \sigma\,[\texttt{ReadyList}_0 \cup (\texttt{regs} = \texttt{Processor}_0\texttt{.regs,c})]\rangle}$$

$$\texttt{resched R}\ \frac{\{(\texttt{regs,c})\} \in \texttt{ReadyList}_0}{\langle \texttt{R}, \sigma\rangle \rightarrow \langle \texttt{c}, \sigma\,[\texttt{Processor}_0\texttt{.regs} = \texttt{regs}, \texttt{ReadyList}_0 \setminus \{(\texttt{regs,c})\}]\rangle}$$

$$\texttt{resched 1}\ \frac{}{\langle \texttt{resched 1; c}, \sigma\rangle \rightarrow \langle \texttt{R}, \sigma\,[\texttt{ReadyList}_1 \cup (\texttt{regs} = \texttt{Processor}_0\texttt{.regs,c})]\rangle}$$

`isFree` *loc*

The predicate `isFree`(*loc*) tells whether the location `loc` has been freed.

A future revision of this semantics adds type states (e.g., Unassigned, Assigned, and Free) to memory locations and gives implementations of `Free` and `Allocate`.

An `unAssignedList` will track the memory locations that may be allocated. Once a memory location is freed, it won't be reallocated. Strategies that allow memory locations to be reallocated make it difficult to detect situations where a reference held by one thread is no longer valid because the memory to which it refers has been freed and reallocated. A *once freed, never reallocated* policy makes it relatively easy to detect invalid references.

6.2 Sequence

$$\texttt{Seq}_1 \frac{\langle \mathtt{c}_0,\, \sigma\rangle \to \sigma'}{\langle \mathtt{c}_0; \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Seq}_2 \frac{\langle \mathtt{c}_0,\, \sigma\rangle \to \langle \mathtt{c}_0',\, \sigma'\rangle}{\langle \mathtt{c}_0; \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_0'; \mathtt{c}_1,\, \sigma'\rangle}$$

6.3 Interleaving Concurrency

$$\texttt{Int}_1 \frac{\langle \mathtt{c}_0,\, \sigma\rangle \to \langle \mathtt{c}_0',\, \sigma'\rangle}{\langle \mathtt{c}_0 \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_0' \parallel \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Int}_2 \frac{\langle \mathtt{c}_0,\, \sigma\rangle \to \sigma'}{\langle \mathtt{c}_0 \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Int}_3 \frac{\langle \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_1',\, \sigma'\rangle}{\langle \mathtt{c}_0 \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_0 \parallel \mathtt{c}_1',\, \sigma'\rangle}$$

$$\texttt{Int}_4 \frac{\langle \mathtt{c}_1,\, \sigma\rangle \to \sigma'}{\langle \mathtt{c}_0 \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_0,\, \sigma'\rangle}$$

6.4 Cooperative Multitasking

$$\texttt{Mul}_1 \frac{\langle \texttt{Resched}\ 0\,;\ \mathtt{c}_0,\, \sigma\rangle \to \langle \mathtt{R},\, \sigma'\rangle}{\langle (\texttt{Resched}\ 0\,;\ \mathtt{c}_0) \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{R} \parallel \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Mul}_2 \frac{\langle \texttt{Resched}\ 1\,;\ \mathtt{c}_0,\, \sigma\rangle \to \langle \mathtt{R},\, \sigma'\rangle}{\langle (\texttt{Resched}\ 1\,;\ \mathtt{c}_0) \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{R} \parallel \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Mul}_3 \frac{\langle \mathtt{R},\, \sigma\rangle \to \langle \mathtt{c}_0,\, \sigma\rangle}{\langle \mathtt{R} \parallel \mathtt{c}_1,\, \sigma\rangle \to \langle \mathtt{c}_0 \parallel \mathtt{c}_1,\, \sigma'\rangle}$$

$$\texttt{Mul}_4 \frac{\langle \texttt{Resched}\ 0\,;\ \mathtt{c}_0,\, \sigma\rangle \to \langle \mathtt{R},\, \sigma'\rangle}{\langle \mathtt{c}_1 \parallel (\texttt{Resched}\ 0\,;\ \mathtt{c}_0),\, \sigma\rangle \to \langle \mathtt{c}_1 \parallel \mathtt{R},\, \sigma'\rangle}$$

$$\texttt{Mul}_5 \frac{\langle \texttt{Resched } 1; \texttt{c}_0, \sigma\rangle \rightarrow \langle \texttt{R}, \sigma'\rangle}{\langle \texttt{c}_1 \parallel (\texttt{Resched } 1 ; \texttt{c}_0), \sigma\rangle \rightarrow \langle \texttt{c}_1 \parallel \texttt{R}, \sigma'\rangle}$$

$$\texttt{Mul}_6 \frac{\langle \texttt{R}, \sigma\rangle \rightarrow \langle \texttt{c}_0, \sigma\rangle}{\langle \texttt{c}_1 \parallel \texttt{R}, \sigma\rangle \rightarrow \langle \texttt{c}_1 \parallel \texttt{c}_0, \sigma'\rangle}$$

7 RCU "Classic"

RCU allows a mutator, without using blocking synchronization, to update a shared data structure concurrent with accessors utilizing that data structure. Figure 1 illustrates RCU "Classic" and shows how the new object is published by (atomically) writing a pointer (g). It is through this pointer that accessors retrieve objects. Since the mutator publishes the new object concurrently with accessor execution, accessors that retrieve a reference (by reading g) to the object before the pointer update coexist with accessors that retrieve a reference after the update. This means that the old object cannot be collected immediately after the global pointer is updated. Instead, the RCU mutator collects the old object's storage only after all processors have passed through a quiescent state[28] which is defined as a context switch, idle loop, or user code[44]. To prevent accessing the old object, a, after it has been collected (which can lead to all sorts of errors), accessors are not allowed to hold references to RCU protected objects through a quiescent state[24–26].

Figure 2 gives our implementation of RCU using the language we defined in Sections 5 and 6. The RCU Mutator begins execution on Processor 0, but after updating the global RCU pointer it reschedules itself to run on Processor 1. Since $ReadyList_0$ is empty, Processor 0 stays in the `R` configuration. Only after Processor 1 executes `resched 0` does Processor 0 transition from the `R` configuration to executing `storei 0 0` and collecting the "old" object's memory.

Our implementation of RCU is also discussed in Section 8, Definition 13 where we give the initial configuration of the RCU "Classic" Program.

8 Verifying RCU Correctness

8.1 Definitions

Definition 1 (Atomic Configuration Formula).

A configuration is a triple (Section 6) comprised of: Processor 0*'s continuation,* $Continuation_0 \in cmds$*; Processor* 1*'s continuation,* $Continuation_1 \in cmds$*; and the "state",* $\sigma \in Processor_0 \times Processor_1 \times ReadyList_0 \times ReadyList_1 \times Mem$.

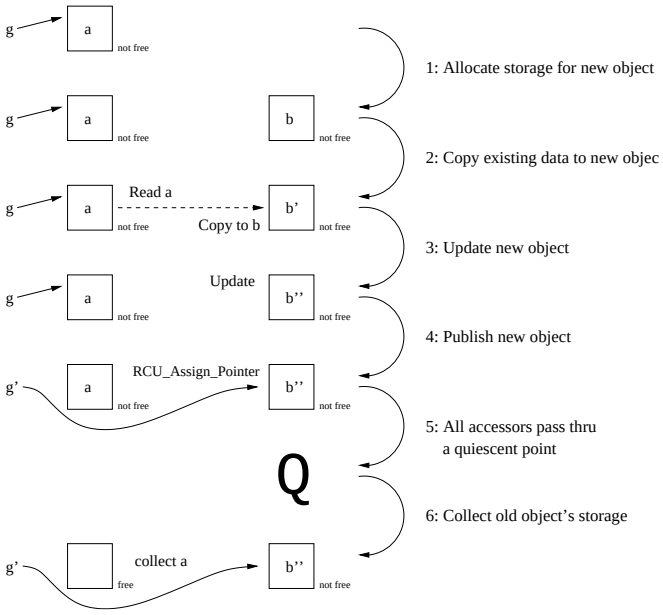

Fig. 1. RCU "Classic". *g is the global pointer that references the shared object. When the shared object (a) is to be modified, RCU creates b, the new object, copies the old to-be-modified-object to b, modifies b, and then "switches" the global pointer so that it points to b (the new object). After all accessors pass through a "quiescent" state, the RCU Mutator collects the old object's storage.*

An atomic configuration formula ranges over a configuration as given by the following BNF:

$$
\begin{aligned}
p ::= \quad & \\
& Continuation_0 = cmds \\
& \mid Continuation_1 = cmds \\
& \mid Processor_0.regs(loc) = val \\
& \mid Processor_1.regs(loc) = val \\
& \mid ReadyList_0 \ setop \ \{rc | rc \in (regs, cmds)\} \\
& \mid ReadyList_1 \ setop \ \{rc | rc \in (regs, cmds)\} \\
& \mid Mem(loc) = val
\end{aligned}
$$

$$
\begin{aligned}
setop ::= \quad & \\
& \subset \mid \subseteq \mid = \mid \neq \mid \cdots
\end{aligned}
$$

Definition 2 (Configuration Formula).

A configuration formula ranges over atomic configuration formulae. The following BNF defines a configuration formula in terms of atomic con-

```
RCU Multiprocessor Implementation
RCU Mutator --- Processor 0:

  load 0 1    ; Reg(0) <-- Mem(1)   *** Read
  store 2 0   ; Mem(2) <-- Reg(0)   *** Copy
  storei 2 2  ; Mem(2) <-- 2        *** Update
  storei 3 2  ; Mem(3) <-- 2        *** RCU_Assign_Ptr
  resched 1   ; Context Switch      *** Run_on Processor 1
  resched 0   ; Context Switch      *** Run_on Processor 0
  storei 0 0  ; Mem(0) <-- 0        *** Collect (free) address 1
  halt

RCU Accessor --- Processor 1:

  forever
                                    *** RCU Lock
    load 0 3  ; Reg(0)<-- Mem(3)
    . . . . .
    . . . . .
                                    *** RCU Unlock
    resched 1 ; Context Switch      *** Run_on Processor 1

Summary of Memory Usage:

   Mem(0): Type state of Mem(1)
   Mem(1): "Old" Object
   Mem(2): "New" Object
   Mem(3): RCU Global Pointer
```

Fig. 2. RCU Multiprocessor Implementation.

figuration formulae:

$\phi ::= (p) \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \mid (\forall i\, \phi) \mid (\exists i\, \phi)$

where p is any atomic configuration formula.

Definition 3 (Specification Formula).

Specifications are first order, linear temporal logic formulae that range over configuration formulae. The following BNF gives the syntax of a specification formula:

$\psi ::= T \mid F \mid (\phi) \mid (\neg\psi) \mid (\psi \vee \psi) \mid (\psi \wedge \psi) \mid (\psi \Rightarrow \psi) \mid (\psi \Leftrightarrow \psi) \mid (\Box\psi) \mid (\Diamond\psi)$

where ϕ is any configuration formula.

Definition 4 (Computation).

A computation is a sequence of configurations, $\omega_0, \ldots, \omega_n$ such that each configuration in the sequence, except for the first, is related to the previous configuration by the application of a semantic rule (Section 6):

$$\omega_i \rightarrow \omega_{i+1}$$

We write, $\omega_0 \xrightarrow{+} \omega_n$ to mean the computation with initial configuration ω_0 and that leads to configuration ω_n by one or more applications of the semantic rules.

Since more than one semantic rule may apply to a given configuration, a configuration may have several possible successor configurations. Thus, there can be many different computations that have a given initial configuration. Any particular computation is but one "branch" of the "tree" of executions. A computation corresponds to what is commonly called an "interleaving".

We write $\Pi_P = \{\pi_0, \ldots\}$ to mean the set of computations that describe all executions of multiprocessor program P (i.e., the initial configuration of each $\pi \in \Pi_P$ is the program P). We omit the subscript P when it's clear by context which program is being described.

Definition 5 (Racey Computations).

Let,

$$\Sigma' = \{\sigma' | \langle C_0 \parallel C_1, \sigma \rangle \xrightarrow{*} \langle C_0' \parallel C_1', \sigma' \rangle\}$$

be the set of possible outcomes of executing the multiprocessor program that begins in the initial configuration $\langle C_0 \parallel C_1, \sigma \rangle$ and that reaches configurations such that Processor 0's continuation is C_0' and Processor 1's continuation is C_1'.

When $|\Sigma'| = 1$, the computation is not racey. A not racey, multiprocessor computation (program), $\langle C_0 \parallel C_1, \sigma \rangle \xrightarrow{*} \langle C_0' \parallel C_1', \sigma' \rangle$, has the state-confluence property:

$$\forall\, C_0''\, C_1''\, \sigma''\ \langle C_0 \parallel C_1, \sigma \rangle \xrightarrow{*} \langle C_0'' \parallel C_1'', \sigma'' \rangle \Rightarrow \langle C_0'' \parallel C_1'', \sigma'' \rangle \xrightarrow{*} \langle C_0' \parallel C_1', \sigma' \rangle$$

When $|\Sigma'| > 1$, the computation is racey. Racey computations do not have the state-confluence property.

A multiprocessor program has a data race if two accesses conflict, at least one of them is a write, and they are not ordered[41]. In the case of two writes, the value observed after both writes depends on which was last; in the case of a write and read, the value read depends on whether the

read happens before the write or after it. A multiprocessor program with a data race is racey[2].

A data race is a sufficient condition for a computation to be racey. However, according to our definition of a racey computation, a data race is not necessary — a computation may be racey because of other attributes of the state and/or there may be mechanisms other than loads that alter memory.

The two computations:

$$\pi_1 = \langle C_1 \parallel C_2, \sigma \rangle \xrightarrow{*} \langle C_1' \parallel C_2', \sigma_1' \rangle$$
$$\pi_2 = \langle C_1 \parallel C_2, \sigma \rangle \xrightarrow{*} \langle C_1' \parallel C_2', \sigma_2' \rangle$$

are racey if $\sigma_1' \neq \sigma_2'$.

These computations are locally racey if $\sigma_1'.Processor_0.regs \neq \sigma_2'.Processor_0.regs \vee \sigma_1'.Processor_1.regs \neq \sigma_2'.Processor_1.regs$.

They are globally racey if $\sigma_1'.Mem \neq \sigma_2'.Mem$

Raceyness is often restricted to specific memory locations, registers, etc. For example, stating that π_1 and π_2 above are globally racey with respect to memory location x means $\sigma_1'.Mem(x) \neq \sigma_2'.Mem(x)$. In such a case, we say that the initial configuration is racey (or leads to computations that are racey) with respect to memory location x (or a processor's register, etc.).

The configuration $\langle store\ 3\ 2; C_0 \parallel load\ 0\ 3; C_1, \sigma \rangle$ is locally racey with respect to Processor 1's register 0. There are exactly two computations that lead to a configuration where Processor 0's continuation is C_0 and and Processor 1's continuation is C_1.

$$\begin{aligned}\pi_0 = \langle store\ 3\ 2; C_0 \parallel load\ 0\ 3; C_1, \sigma[Mem(3) = 1] \rangle \rightarrow \\ \langle C_0 \parallel load\ 0\ 3; C_1, \sigma_0' = \sigma[Mem(3) = 2] \rangle \rightarrow \\ \langle C_0 \parallel C_1, \sigma_0'' = \sigma[Mem(3) = 2, P_1.Reg(0) = 2] \rangle\end{aligned}$$

$$\begin{aligned}\pi_1 = \langle store\ 3\ 2; C_0 \parallel load\ 0\ 3; C_1, \sigma[Mem(3) = 1] \rangle \rightarrow \\ \langle store\ 3\ 2; C_0 \parallel; C_1, \sigma_1' = \sigma[Mem(3) = 1, P_1.Reg(0) = 1] \rangle \rightarrow \\ \langle C_0 \parallel C_1, \sigma_1'' = \sigma[Mem(3) = 2, P_1.Reg(0) = 1] \rangle\end{aligned}$$

As also depicted in Figure 3a, these two computations lead to different results:

$$\sigma_0''.\text{Processor1.regs}(0) = 1$$
$$\sigma_1''.\text{Processor1.regs}(0) = 2$$

Figure 3b depicts a not racey computation.

Definition 6 (Fairness Constraint).

[2] In particular, the computation delineated by the continuations where each processor executes the conflicting access defines a racey computation.

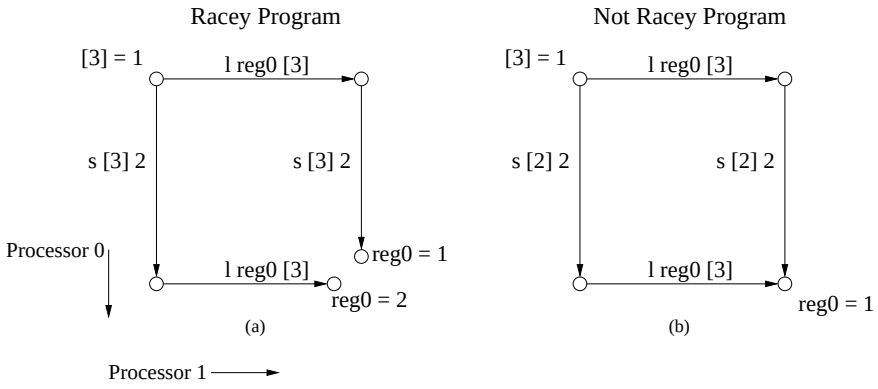

Fig. 3. Racey and Not Racey Computations. *A locally racey computation is shown in (a). A not racey computation is shown in (b). Not racey computations can be represented as a commutating diagram because the "final" configurations are equivalent.*

A fairness constraint is a configuration formula ϕ. Given a computation $\pi \in \Pi_P$ and fairness constraint ϕ. π satisfies ϕ iff the fairness constraint is true infinitely often along π (i.e., $\pi \models \Box\Diamond(\phi)$).

Definition 7 (Fair Execution).

Let $F_c = \{\phi_1, \ldots, \phi_n\}$ be a set of n fairness constraints. Let $\Pi = \{\pi_0, \ldots\}$ be the set of computations that describe all executions of program P. We call the set, $F = \{\pi \in \Pi | \pi \models \bigwedge_{F_c} \Box\Diamond(\phi_i)\}$, the fair executions of P with respect to F_c.

Interleaving semantics trade parallel/independent processor execution for a model that allows only one processor, at a time, to perform a step in the computation. In this model, it's possible to produce a computation where only one processor makes steps in the computation, even though this computation is not observable by any real system. Fairness constraints and the resulting fair execution paths are used to restrict verification to realizable computations.

Definition 8 (Roots).

The "Roots" of a program are a set of distinguished objects that are reachable (accessible via the normal rules of memory access).

The objects that comprise the roots of RCU "Classic" are the global RCU pointer, g and the Accessor's local RCU pointer g_1.

Definition 9 (Live Object).

An object is live if it is in the set of roots or is in the transitive closure of objects reachable from any object in the set of roots.

Definition 10 (Memory Safe).

A computation is memory safe if it does not dereference Nil and it does not access memory that has been freed. The two instructions that dereference pointers are `di` *and* `dr`*; if either dereferences Nil, the resulting continuation is $Error$. Also, if either accesses memory that has been freed, the resulting continuation is $Error$. Thus, computation π is memory safe it does not lead to the $Error$ continuation:*

$$\begin{array}{l} \textit{Memory Safe } \pi \Leftrightarrow \\ \quad \pi \models \neg\Diamond \left(Continuation_0 = Error \ \vee \ Continuation_1 = Error\right) \end{array}$$

Definition 11 (Variable Use/Def).

The use of a variable is the set of positions of those statements where the variable appears on the right-hand side of an expression.

The def of a variable is the set of positions of those statements where the variable appears on the left-hand side of an expression. Since statements are not labeled in our language, the position is the continuation such that definition is the first command in the sequence of commands that make up the continuation.

Definition 12 (Reaching Definition).

The reaching definition for a particular use of a variable is the set of definitions for the variable that "reach" that use — definition d of variable t reaches a statement u that "uses" the variable t if there is a sequence of configurations from the definition d to u such that there is no other configuration that defines t.

Definition 13 (RCU "Classic" Program).

$P_0 =$ `load 0 1;store 2 0;storei 2 2;storei 3 2;resched 1;resched 0;storei 0 0;halt`

$P_1 =$ `forever (load 0 3;resched 1)`

Initial configuration:

$\langle P_0 \parallel P_1, \sigma \rangle$

$$\sigma \ = \begin{cases} Mem(0) = 1, Mem(1) = 1, Mem(2) = 0, Mem(3) = 1 \\ ReadyList0 = \emptyset \\ ReadyList1 = \emptyset \\ Processor1.Regs(0) = Nil \end{cases}$$

P_0 is called the "mutator" program and it executes on Processor 0. Similarly, P_1 is called the "accessor" program and it executes on Processor 1.

We also define an instrumented version of the accessor program:

$I_1 =$ `forever (load 0 3;loadi 0 Nil;resched 1)`

The instrumented version differs only by the addition of a load immediate that "kills" the definition of register 0*. The initial configuration:*

$\langle P_0 \parallel I_1, \sigma \rangle$

is called Instrumented RCU "Classic".

The implementations of RCU make use of certain memory locations in particular ways:

$Mem(0) = 0 \Rightarrow Mem(1)$ *is free.*
$Mem(1)$ *is the "old" object.*
$Mem(2)$ *is the "new" object.*
$Mem(3)$ *is the global pointer,* g*.*

Section 7 gives an overview of RCU "Classic".

Definition 14 (Read-Side Critical Section).

The RCU Read-Side Critical Section is the code region in an RCU Accessor program where operations take place that involve RCU protected objects. In a traditional locking environment, one would find this region bounded by lock and unlock directives.

RCU Read-Side Critical Sections do not contain synchronizing events such as the resched command.

8.2 Assumptions

Assumption 1 (Fair Progress Constraint)

$$Continuation_0 = Halt$$

Let $\Pi = \{\pi_0, \ldots\}$ be the set of computations that describe execution of RCU (i.e., the initial configuration of each $\pi \in \Pi$ is $\langle P_0 \parallel P_1, \sigma \rangle$).

The set of fair executions of RCU, Π_{fair} with respect to the fair program constraint is given by

$$\Pi_{fair} = \{\pi \in \Pi \mid \pi \models \Box\Diamond (Continuation_0 = Halt)\}$$

Similarly, let $\Gamma = \{\gamma_0, \ldots\}$ be the set of computations that describe execution of Instrumented RCU (i.e., the initial configuration of each $\gamma \in \Gamma$ is $\langle P_0 \parallel I_1, \sigma \rangle$).

The set of fair executions of Instrumented RCU, Γ_{fair} with respect to the fair program constraint is given by

$$\Gamma_{fair} = \{\gamma \in \Gamma \mid \gamma \models \Box\Diamond (Continuation_0 = Halt)\}$$

8.3 Verification

In this section, we prove that RCU implementations that conform to particular restrictions are memory safe and do not collect live objects. We also show how RCU achieves synchronization without using synchronization primitives.

Our proof is structured as follows. First, we prove that every fair execution of RCU reaches a state such that Processor 0 is "blocked" waiting for Processor 1. In this state, Processor 1 can always allow Processor 0 to proceed. By instrumenting the code, it is easy to prove that the "old" object is not live in this state. We show that this state alway precedes collection; thus, instrumented RCU cannot collect live objects.

To make it easy to show that RCU is memory safe (doesn't access nil pointers or retrieve freed memory), we instrument the semantics to generate an error continuation if RCU accesses freed memory or attempts to dereference a nil pointer. We then prove that instrumented RCU cannot reach a configuration where the continuation is error.

We next show that the restriction on fair executions can be eliminated — thus proving that every execution of instrumented RCU using instrumented semantics doesn't collect live objects and is memory safe. Lastly, we introduce restrictions on the local RCU pointer's def/use chain that allow us to remove the instrumentation. This establishes that that RCU implementations that conform to these restrictions are memory safe and do not collect live objects.

8.3.1 Instrumented RCU does not collect live objects

Let,

$$
\begin{array}{l}
f \triangleq Continuation_0 = R \wedge \\
\quad Mem = \sigma.Mem[Mem(2) = 2, Mem(3) = 2] \wedge \\
\quad\quad \{(\sigma.Processor_0[regs(0) = 1], \texttt{resched 0;storei 0 0;halt})\} \subseteq \\
\quad\quad\quad ReadyList_1)
\end{array}
$$

be the configuration formula describing the configuration arising after the Mutator (Processor 0) executes the `resched 1` instruction. This is the configuration in which Processor 0 "blocks" waiting for Processor 1.

Lemma 1.

$\forall \gamma \in \Gamma_{fair}\ \gamma \models \Diamond\ f$

Proof:

By definition of sequencing, `cmds` *are executed in sequence.*

Since, for all fair executions, Processor 0 *"reaches" the* `halt` *instruction of* P_0*, it must be the case that the instructions located before* `halt` *were first executed. In particular, the continuation* `resched 1;resched 0;storei 0 0;halt`

must have been reached. And since only Processor 0 *updates memory,* $Mem(2) = 2,\ Mem(3) = 2$ *and Processor* 0*'s Register* 0 *is* 1.

By definition of the semantics of the `resched 1` *instruction, whenever Processor* 0 *executes* `resched 1`*,* $Continuation_0 = R$*. Thus,*

$$\forall \gamma \in \Gamma_{fair}\ \gamma \models \Diamond\ f$$

□

Let,

$$g \stackrel{\Delta}{=} f\ \wedge\ Continuation_1 = R \wedge \{(\sigma.Processor_1[regs(0) = \texttt{Nil}], \texttt{forever (load 0 3;loadi 0 Nil;resched 1))}\} \subseteq ReadyList_1)$$

be the configuration formula describing the configuration where Processor 0 is "blocked" waiting for Processor 1 and Processor 1 is able to continue either with the Accessor or with the Mutator. Note that Mutator immediately reschedules itself to run on Processor 0 — thereby "unblocking" Processor 0.

Lemma 2.

$\forall \gamma \in \Gamma_{fair}\ \gamma \models \Box(f \Rightarrow \Diamond g)$

Proof:

Processor 0 *cannot transition out of the configuration where its continuation is R unless its ReadyList is not empty.* $ReadyList_0$ *is populated by the mutator continuation that executes on Processor* 1 *(i.e.,* `resched 0;...`*). For that continuation to run, Processor* 1*'s accessor program must have executed its* `resched` *instruction.*

Since Processor 1 *executes independently of Processor* 0*, at the time Processor* 0 *executes* `resched 1` *and enters the R continuation, Processor* 1 *continuation can be any legal sequence; if it is R, the Lemma is proved. If it is not R, then since the accessor program loops, it will eventually execute the* `resched 1` *instruction causing it to enter the R continuation — and it will have tasks on its ReadyList.*

□

Lemma 3.

$(\Diamond p\ \wedge\ \Box(p \Rightarrow \Diamond q)) \Rightarrow \Diamond q$

Proof:

$(\Diamond p\ \wedge\ \Box(p \Rightarrow \Diamond q))$	*1. Assumption*
$\Diamond p$	*2. (1), Logic:* $a \wedge b \Rightarrow a$
$\Box(p \Rightarrow \Diamond q)$	*3. (1), Logic:* $a \wedge b \Rightarrow b$
p	*4. Definition of* $\Diamond$*, (2)*
$\Diamond q$	*5. Definition of* $\Box$*, (4), (3), modus ponens*

Discharging the assumption gives the lemma:

$$(\Diamond p \ \wedge \ \Box(p \Rightarrow \Diamond q)) \Rightarrow \Diamond q$$

□

Lemma 4.

$\forall \gamma \in \Gamma_{fair} \ \gamma \models \Diamond \ g$

Proof:

$\forall \gamma \in \Gamma_{fair} \ \gamma \models \Diamond \ f$ *Lemma 1*

$\forall \gamma \in \Gamma_{fair} \ \gamma \models \Box \ (f \Rightarrow \Diamond \ g)$ *Lemma 2*

$\forall \gamma \in \Gamma_{fair} \ \gamma \models \Diamond \ g$ *Lemma 1, Lemma 2, Lemma 3*

□

By Lemma 4, the configuration $\langle R \parallel R, \sigma'' \rangle$, where

$$\sigma'' = \left\{ \begin{array}{l} Processor_0 = \sigma.Processor_0[Regs(0) = 1] \\ Processor_1 = \sigma.Processor_0[Regs(0) = 0] \\ Mem = \sigma.Mem[Mem(2) = 2, Mem(3) = 2] \\ ReadyList_0 = \emptyset \\ ReadyList_1 = \{ \\ \quad (\sigma.Processor_0[regs(0) = 1], \\ \quad\quad \texttt{resched 0;storei 0 0;halt}), \\ \quad (\sigma.Processor_1[regs(0) = \texttt{Nil}], \\ \quad\quad \texttt{forever (load 0 3;loadi 0 Nil;resched 1)}) \\ \} \end{array} \right.$$

is inevitable: $\langle P_0 \parallel I_1, \sigma \rangle \xrightarrow{*} \langle R \parallel R, \sigma'' \rangle$.

Lemma 5.

live $x \Leftrightarrow$
$Mem(3) = x \ \vee$
$\exists (regs, c) \in ReadyList_1 (regs(0) = x \ \wedge \ c = \texttt{I}_1) \ \vee$
$(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \ \vee$
$(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \ \vee$
$(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x)$
where $x \in \texttt{loc}$.

Proof:

A RCU object is live (Definition 9) only if it can be reached via the global RCU pointer or the local RCU pointer (these two pointers are the only objects in the set of roots – Definition 8).

(1) *Prove "⇐" part:*

$(Mem(3) = x) \vee$
$(\exists(regs, c) \in ReadyList_1(regs(0) = x \ \wedge \ c = \texttt{I}_1) \vee$
$(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x)$
$\Rightarrow$ *live* x, *where* $x \in$ `loc`.

Since the global RCU pointer is memory location 3,

$$(1) \quad Mem(3) = x \Rightarrow \textit{live } x, \ x \in \texttt{loc}$$

The other way for an object to be live is via the local RCU pointer. Although the RCU local pointer is Processor 1*'s Register* 0*, Processor* 1*'s Register* 0 *is not always the local RCU pointer. When Processor* 1 *is executing the RCU Accessor program (*$\texttt{I}_1$*), Processor* 1*'s Register* 0 *is the RCU local pointer. When Processor* 1 *is not executing the RCU Accessor program, the RCU local pointer is on* $ReadyList_1$ *in the context portion of the context-continuation pair associated with the RCU Accessor Program. Thus, when Processor* 1 *is not executing the RCU Accessor Program,*

$$(2) \quad \exists(regs, c) \in ReadyList_1(regs(0) = x \wedge c = \texttt{I}_1) \Rightarrow \textit{live } x, \ x \in \texttt{loc}$$[3]

The following equivalence (obtained by applying the definition of the `forever` *instruction to the RCU Accessor program) is used to define the possible continuations of the RCU Accessor Program.*

$$\texttt{I}_1 \ = \ \texttt{load 0 3;loadi 0 Nil;resched 1;I}_1$$

This gives rise to the following cases:

(a) $(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \Rightarrow \textit{live } x$
(b) $(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \Rightarrow \textit{live } x$
(c) $(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \Rightarrow \textit{live } x$

$x \in$ `loc`

Case (c) *is unnecessary because the definition of register* 0 *is killed by the previous instruction (i.e., writing a* `Nil` *to Processor* 1*'s Register* 0 *ensures that Register* 0 *cannot point to an object). However, we will later consider real RCU (without instrumentation), so we we won't optimize by removing this case.*

[3] (2) means that if the accessor's register 0 "points" to x then x is live, even when the accessor is switched out.

(1), (2), and Cases (a),(b),(c) give:

$(Mem(3) = x) \vee$
$(\exists(regs, c) \in ReadyList_1(regs(0) = x \ \wedge \ c = \texttt{I}_1) \vee$
$(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x)$
$\Rightarrow$ *live* x, *where* $x \in$ `loc`.

this proves the "$\Leftarrow$" part.

(2) *Prove the "$\Rightarrow$" part:*

live $x \Rightarrow$
$Mem(3) = x \vee$
$\exists(regs, c) \in ReadyList_1(regs(0) = x \ \wedge \ c = \texttt{I}_1) \vee$
$(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x)$
where $x \in$ `loc`.

Proof:

By definition (Definitions 8, 9),

$$live\ x \Rightarrow RCU\ Global\ pointer = x \ \vee \ RCU\ Local\ pointer = x$$

In proving the "left" arrow portion of the proof, we showed that the RCU Local pointer is Processor 1*'s Register* 0 *when Processor* 1*'s continuation is the RCU Accessor program; otherwise the RCU Local pointer is on* $ReadyList_1$ *in the context portion of the context-continuation pair associated with the RCU Accessor Program.*

Since, the RCU global pointer is $Mem(3)$ *and because the case analysis performed in the "left" arrow portion is exhaustive, we can reverse the implications in (1),(2),and cases (a),(b),and (c); which proves the "right" part.*

Since we proved the left and right parts, the lemma follows:

live $x \Leftrightarrow$
$Mem(3) = x \vee$
$\exists(regs, c) \in ReadyList_1(regs(0) = x \ \wedge \ c = \texttt{I}_1) \vee$
$(Continuation_1 = \texttt{I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{loadi 0 Nil;resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x) \vee$
$(Continuation_1 = \texttt{resched 1;I}_1 \ \wedge \ Processor_1.regs(0) = x)$
where $x \in$ `loc`.

□

The Mutator collects the old object by "freeing" the memory associated with the old object. As described earlier, memory location 0 tracks the type-state of the old object (memory location 1). The `storei 0 0` statement "frees" the old object; if memory location 1 is *live* when the Mutator frees it, then a live object has been collected. This gives the equivalence[4]:

$$collectLive\ x \Leftrightarrow (Continuation_0 = \texttt{storei 0 0;halt} \ \wedge \ live\ x)$$

Lemma 6 (Fair executions of Instrumented RCU do not collect live objects).

$\forall \gamma \in \Gamma_{fair}\ \gamma \models \Box \neg (collectLive\ 1)$

Proof:

Every fair execution of Instrumented RCU passes through the configuration defined in Lemma 4. This configuration is characterized by both processors continuations being R. In this configuration, the global RCU pointer references the "new" object and the local RCU pointer references `Nil`*. Thus, in this "*$\langle R \parallel R, \sigma'' \rangle$*" configuration, the old object (memory location* 1*) is not live. Because the local RCU pointer is only defined by the global pointer, any future definition of the local pointer cannot reference the "old" object.*

Since every configuration where the Mutator's continuation equals `storei 0 0;halt` *is dominated by the "*$\langle R \parallel R, \sigma'' \rangle$*" configuration,*

$$\forall \gamma \in \Gamma_{fair}\ \gamma \not\models \Diamond (collectLive\ 1)$$

The lemma

$$\forall \gamma \in \Gamma_{fair}\ \gamma \models \Box \neg (collectLive\ 1)$$

follows from $\Box \Leftrightarrow \neg \Diamond \neg$ *in LTL.*

□

Theorem 1 (Instrumented RCU does not collect live objects).

$\forall \gamma \in \Gamma\ \gamma \models \Box \neg (collectLive\ 1)$

Proof:

The Fair Progress Constraint (Assumption 1) partitions the set of Instrumented RCU computations such that every computation is either in the set of fair executions or it is not:

$$\forall \gamma \in \Gamma\ (\gamma \in \Gamma_{fair} \ \vee \ \gamma \notin \Gamma_{fair})$$

Since every $\gamma \notin \Gamma_{fair}$ *does not collect storage*

[4] As discussed earlier, a future version of the semantics will provide allocate and free; collecting a live object becomes ($Continuation_0 =$ `free(x);_` $\wedge$ *live* x).

$$\forall \gamma \in \Gamma\ \gamma \models \text{`}collect\ live\ object\text{`} \Rightarrow \gamma \in \Gamma_{fair}$$

Thus,to prove the theorem that Instrumented RCU does not collect live objects, it is sufficient to prove that no fair execution of Instrumented RCU collects a live object, that is

$$\forall \gamma_1 \in \Gamma_{fair}\ \gamma_1 \models \Box \neg (collectLive\ 1) \Rightarrow \\ (\forall \gamma_2 \in \Gamma\ \gamma_2 \models \Box \neg (collectLive\ 1))$$

Since the antecedent is given by Lemma 6, Modus Ponens and renaming gives the theorem:

$$\forall \gamma \in \Gamma\ \gamma \models \Box \neg (collectLive\ 1)$$

□

8.3.2 Instrumented RCU with restrictions is memory safe

A continuation is a sequence. Each command in the sequence has an implicit position. Labeling a continuation gives an implicit position an explicit name. For simplicity, the first command of the initial continuation of a multiprocessor program is labeled ℓ_0, the second ℓ_1, and so on. For commands comprising the body of a `forever` loop, only the body is labeled. For example, the instrumented RCU Accessor continuation ($\mathtt{I}_1$) is labeled:

$$\mathtt{I}_1 = \ell_0\mathtt{:load\ 0\ 3};\ \ell_1\mathtt{:loadi\ 0\ Nil};\ \ell_2\mathtt{:resched\ 1};\ \mathtt{I}_1$$

Let $\ell_0 \prec \ell_1$ mean that the program statement located at position ℓ_0 is statically located before the program statement located at position ℓ_1. Given a continuation C and a command c, $Pos\ C\ c$ gives the position (label) of c within C:

$$Pos\ C\ c\ \ell \Leftrightarrow C = \cdots\ \ell\mathtt{:}c\ \cdots$$

$\mathtt{I}_1$'s Read-Side Critical Section begins at ℓ_0 and ends at ℓ_1.

Applications of RCU need to dereference the RCU local pointer. Showing that RCU is memory safe (Definition 10) requires showing that dereferencing the RCU local pointer does not lead to `Error` as Processor 1's continuation. To ensure that dereferencing the RCU local pointer is safe, we give the following restrictions:

Restriction 1

Local RCU pointer use occurs only within the Read-Side Critical Section.

$$\forall \ell\ Pos\ \mathtt{I}_1\ \mathtt{dr\ 0}\ \ell\ \Rightarrow\ (\ell \succ \ell_0\ \wedge\ \ell \prec \ell_1)$$

Restriction 2

For each use of the local RCU pointer, every reaching definition is statically located before the use.

$$\forall \ell\ Pos\ \mathtt{I}_1\ \mathtt{dr\ 0}\ \ell \Rightarrow \forall \ell'\ RDef\ Processor1.regs(0)\ \ell\ \ell' \Rightarrow \ell' \prec \ell$$

Restriction 3

For each use of the local RCU pointer, every reaching definition uses only the global RCU pointer.

$$\forall \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell \Rightarrow \forall \ell'\ RDef\ Processor1.regs(0)\ \ell\ \ell' \Rightarrow usesOnly\ Mem(3)\ \ell'$$

Lemma 7 ($\neg$ Restriction 1 $\Rightarrow$ $\neg$ Memory Safe).

$$\begin{array}{l}(\exists \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell\ \wedge\ (\ell \prec \ell_0\ \vee\ \ell \succ \ell_1))\ \Rightarrow \\ \quad \exists \gamma \in \Gamma\ \gamma \models \Diamond\, (Continuation_1 = Error)\end{array}$$

Proof:

(1) $$\begin{array}{l}\exists \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell\ \wedge\ \ell \succ \ell_1\ \Rightarrow \\ \quad \exists \gamma \in \Gamma\ \gamma \models \Diamond\, (Continuation_1 = Error)\end{array}$$

Since the command at ℓ_1 assigns Nil to the local RCU pointer, placing the `dr 0` *instruction immediately after ℓ_1 ensures the transition $\langle dr, \sigma\rangle \rightarrow \langle Error, \sigma\rangle$.*

(2) $$\begin{array}{l}\exists \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell\ \wedge\ \ell \prec \ell_0\ \Rightarrow \\ \quad \exists \gamma \in \Gamma\ \gamma \models \Diamond\, (Continuation_1 = Error)\end{array}$$

Since the initial value of the local RCU pointer is Nil (Definition 13), dereferencing it before defining it ensures the transition $\langle dr, \sigma\rangle \rightarrow \langle Error, \sigma\rangle$.

Thus,

$$\begin{array}{l}(\exists \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell\ \wedge\ (\ell \prec \ell_0\ \vee\ \ell \succ \ell_1))\ \Rightarrow \\ \quad \exists \gamma \in \Gamma\ \gamma \models \Diamond\, (Continuation_1 = Error)\end{array}$$

□

Lemma 8 ($\neg$ Restriction 2 $\Rightarrow$ $\neg$Memory Safe).

$$\begin{array}{l}(\exists \ell\ Pos\ \texttt{I}_1\ \texttt{dr 0}\ \ell\ \wedge \\ \quad \exists \ell'\ RDef\ Processor1.regs(0)\ \ell\ \ell'\ \wedge \\ \quad\quad \ell' \succ \ell) \Rightarrow \\ \quad\quad\quad \exists \gamma \in \Gamma\ \gamma \models \Diamond\, (Continuation_1 = Error)\end{array}$$

Proof:

A definition located after a use of the local RCU pointer that reaches that use must be located before the `resched 1` *instruction. Locations before the* `resched 1` *instruction can be partitioned into two regions: Positions before the local RCU pointer is assigned Nil (ℓ_1) and positions after it is assigned Nil.*

Definitions located after ℓ_1 can acquire an object via the global RCU pointer that is "freed" by the RCU mutator before the accessor's continuation transitions from R to $\mathtt{I}_1$.

Definitions before ℓ_1 are "killed" by the definition at ℓ_1 that assigns Nil to the local RCU pointer.

In either case, the transition $\langle dr, \sigma \rangle \rightarrow \langle Error, \sigma \rangle$ is assured, giving

$$\begin{array}{l}(\exists \ell \; Pos \; \mathtt{I}_1 \; \mathtt{dr} \; \mathtt{0} \; \ell \wedge \\ \quad \exists \ell' \; RDef \; Processor1.regs(0) \; \ell \; \ell' \wedge \\ \quad\quad \ell' \succ \ell) \Rightarrow \\ \quad\quad\quad \exists \gamma \in \Gamma \; \gamma \models \Diamond \, (Continuation_1 = Error)\end{array}$$

□

Lemma 9 (Restrictions 1, 2, 3 are necessary for Instrumented RCU to be Memory Safe).

$$Memory\ Safe \Rightarrow R1 \wedge R2 \wedge R3$$

Proof:

Contrapositive:

$$\neg\, R1 \vee \neg\, R2 \vee \neg\, R3 \Rightarrow \neg\, Memory\ Safe$$

Rewriting using definition: "$\Rightarrow$" $\stackrel{\Delta}{=}$ $(a \Rightarrow b \Leftrightarrow \neg\, a \vee b)$:

$$(R1 \wedge R2 \wedge R3) \vee (\neg\, Memory\ Safe)$$

Distributing the "and" over the "or":

$$(R1 \vee \neg\, Memory\ Safe) \wedge (R2 \vee \neg\, Memory\ Safe) \wedge (R3 \vee \neg\, Memory\ Safe)$$

Rewriting using definition of "$\Rightarrow$":

$$\begin{array}{l}(\neg\, R1 \Rightarrow \neg\, Memory\ Safe) \wedge \\ (\neg\, R2 \Rightarrow \neg\, Memory\ Safe) \wedge \\ (\neg\, R3 \Rightarrow \neg\, Memory\ Safe)\end{array}$$

This gives the equivalence:

$$(Memory\ Safe \Rightarrow R1 \wedge R2 \wedge R3) \Leftrightarrow \begin{pmatrix} \neg\, R1 \Rightarrow \neg\, Memory\ Safe \wedge \\ \neg\, R2 \Rightarrow \neg\, Memory\ Safe \wedge \\ \neg\, R3 \Rightarrow \neg\, Memory\ Safe \end{pmatrix}$$

By Lemma 7: $\neg\, R1 \Rightarrow \neg\, Memory\ Safe$

By Lemma 8: $\neg\, R2 \Rightarrow \neg\, Memory\ Safe$

If the local RCU pointer is defined to reference objects other than those to which the global RCU pointer refers, then we cannot assert that RCU is memory safe. So,

$\neg\, R3 \Rightarrow \neg\, \mathit{Memory\ Safe}$

Thus,

$$(\neg\, R1 \Rightarrow \neg\, \mathit{Memory\ Safe}) \wedge$$
$$(\neg\, R2 \Rightarrow \neg\, \mathit{Memory\ Safe}) \wedge$$
$$(\neg\, R3 \Rightarrow \neg\, \mathit{Memory\ Safe})$$

Then, by the above equivalence:

$$(\mathit{Memory\ Safe} \Rightarrow R1 \wedge R2 \wedge R3)$$

□

Lemma 10 (Restrictions 1, 2, 3 are sufficient for Instrumented RCU to be Memory Safe).

$$R1 \wedge R2 \wedge R3 \Rightarrow \mathit{Memory\ Safe}$$

Proof:

Contrapositive

$$\neg\, \mathit{Memory\ Safe} \Rightarrow \neg\, (R1 \wedge R2 \wedge R3)$$

Distribute $\neg$ *over conjunction in consequent:*

$$\neg\, \mathit{Memory\ Safe} \Rightarrow (\neg\, R1 \ \vee\ \neg\, R2 \ \vee\ \neg\, R3)$$

Rewrite, expanding definitions

$$\begin{array}{l}
\exists \gamma \in \Gamma\ \gamma \models \Diamond\, (\mathit{Continuation}_1 = \mathit{Error}) \Rightarrow \\
\quad (\exists \ell\ \mathit{Pos}\ \mathtt{I}_1\ \texttt{dr 0}\ \ell\ \Rightarrow\ \ell \prec \ell_0\ \vee\ \ell \succ \ell_1) \vee \\
\quad (\exists \ell\ \mathit{Pos}\ \mathtt{I}_1\ \texttt{dr 0}\ \ell \Rightarrow \\
\qquad \exists \ell'\ \mathit{RDef}\ \mathit{Processor}1.\mathit{regs}(0)\ \ell\ \ell' \Rightarrow \ell' \succ \ell) \vee \\
\quad (\exists \ell\ \mathit{Pos}\ \mathtt{I}_1\ \texttt{dr 0}\ \ell \Rightarrow \\
\qquad \exists \ell'\ \mathit{RDef}\ \mathit{Processor}1.\mathit{regs}(0)\ \ell\ \ell' \Rightarrow \\
\qquad\quad \neg\, \mathit{usesOnly}\ \mathit{Mem}(3)\ \ell')
\end{array}$$

By definition, $\pi \models \Diamond\, (\mathit{Continuation}_1 = \mathit{Error})$ *implies that the* `dr 0` *command transitioned to "Error"; this can only happen if the local RCU pointer was* Nil *when dereferenced or the object to which it pointed was "Freed".*

A dereference operation occurs either inside the Read-Side Critical Section or outside it. If the dereference operation occurs outside of the critical region, $\mathit{Continuation}_1 = \mathit{Error}$ *and Restriction* 1 *is violated.*

If the dereference operation occurs within the critical section, then the error continuation results only if the dereference operation attempts to dereference Nil *or the operation attempts to dereference a memory location that has been* Freed*. Dereferencing* Nil *can only occur if there is*

no reaching definition for that use that is before the use. Dereferencing a Freed location can only occur if there is a reaching definition that is located after the use. Both of these violate Restriction 2.

This shows that it is not possible to reach the "Error" continuation if the accessor meets Restrictions 1 *and* 2 *and every definition of the local RCU uses only the global RCU pointer. Thus, if Restrictions* 1 *and* 2 *are met, then if the accessor reaches the "Error" continuation, it must be the case that some reaching definition for the local RCU pointer does not reference the global RCU pointer. This violates Restriction* 3.

Thus,

$$(\neg \textit{Memory Safe} \Rightarrow \neg R1) \vee$$
$$(\neg \textit{Memory Safe} \Rightarrow \neg R2) \vee$$
$$(\neg \textit{Memory Safe} \Rightarrow \neg R3)$$

Which gives the lemma

$$R1 \wedge R2 \wedge R3 \Rightarrow \textit{Memory Safe}$$

□.

Theorem 2 (Restrictions 1, 2, 3 are necessary and sufficient for Instrumented RCU to be Memory Safe).

$$\textit{Memory Safe} \Leftrightarrow R1 \wedge R2 \wedge R3$$

Proof:

$\textit{Memory Safe} \Rightarrow R1 \wedge R2 \wedge R3$	*Lemma* 9
$R1 \wedge R2 \wedge R3 \Rightarrow \textit{Memory Safe}$	*Lemma* 10
$\textit{Memory Safe} \Leftrightarrow R1 \wedge R2 \wedge R3$	*Definition of* $\Leftrightarrow$

□.

8.3.3 Uninstrumented RCU has the live object and (with restrictions) memory safety properties

Let $\mathcal{L}$ be the uninstrumented (processor) semantics and $\mathcal{L}'$ be the instrumented semantics. As noted in Sections 5 and 6, the difference between the instrumented and uninstrumented syntax and semantics are `Nil` as a `val` and the `dr` and `di` instructions transitions to `Error`. Note that storing `Nil` to a memory location (or register) is a different transition than storing `digits` to a memory location (or register).

Recall that a computation is a sequence of configurations (Definition 4) such that the transition from one configuration to the next is determined by the application of a semantic rule. The semantics of a language is defined by the set

semantic rules that define the transitions — each semantic rule is a potential transition in the execution of some program.

Let $T = \{t_0, \ldots, t_n\}$ be the set of transitions for $\mathcal{L}$ (uninstrumented semantics). Let the set of transitions $T' = \{t_0, \ldots, t_n, \ldots, t_m\}$ be those for the instrumented semantics, $\mathcal{L}'$. $T \subset T' \Rightarrow \mathcal{L} \subset \mathcal{L}'$ and, in fact, T' differs from T only in that T' has the *extra* transitions $t_{n+1}, \ldots, t_m$ — that is, $T' = T \cup \{t_{n+1}, \ldots, t_m\}$.

Let P be the uninstrumented RCU "Classic" program and P' be the instrumented RCU "Classic" program. As described in Definition 13, the instrumented RCU "Classic" program adds the instruction, `load 0 Nil`, at the end of the RCU Read-side Critical Section.

$\mathcal{L}'(P') = \{h'_0, \ldots\}$ is the set of all executions of instrumented RCU (P') with respect to the instrumented semantics, $\mathcal{L}'$. Similarly, $\mathcal{L}(P) = \{h_0, \ldots\}$ is the set of all executions of uninstrumented RCU with respect to the uninstrumented semantics, $\mathcal{L}$. Since $\mathcal{L}$ and $\mathcal{L}'$ differ only with respect to transitions $\{t_{n+1}, \ldots, t_m\}$, if no computation in the $\mathcal{L}'$ executions of P' uses those transitions, then $\mathcal{L}'(P') = \mathcal{L}(P)$.

Lemma 11.

$$\begin{gathered}\forall t \in \{t_{n+1}, \ldots, t_m\}, \forall h' \in \mathcal{L}'(P') \\ h' \not\models \langle c, \sigma\rangle \xrightarrow{t} \langle c', \sigma'\rangle \Rightarrow \\ \mathcal{L}'(P') = \mathcal{L}(P)\end{gathered}$$

Theorem 1 shows that an instrumented version of RCU executing an instrumented semantics does not collect live objects. Theorem 2 shows that an instrumented version of RCU executing an instrumented semantics is memory safe if and only if certain restrictions are obeyed. By instrumenting the code and the semantics, these properties were stated simply and the properties easily proved.

$$\forall \pi' \in \mathcal{L}'(P')\ \pi' \models \mathcal{Q}$$

To prove that the uninstrumented code executing the uninstrumented semantics has the same properties, identify restrictions R such that

$$\begin{gathered}R(P') \Rightarrow \mathcal{L}' = \mathcal{L} \\ \wedge\ R(P') \Rightarrow P' = P\end{gathered}$$

Then,

$$R(P) \Rightarrow \forall \pi \in \mathcal{L}(P)\ \pi \models \mathcal{Q}$$

Lemma 12.

$$\begin{gathered}(\forall \pi' \in \mathcal{L}'(P')\ \pi' \models \mathcal{Q} \wedge \\ (R(P') \Rightarrow \mathcal{L}' = \mathcal{L}\ \wedge\ R(P') \Rightarrow P' = P)) \Rightarrow \\ R(P) \Rightarrow \forall \pi \in \mathcal{L}(P)\ \pi \models \mathcal{Q}\end{gathered}$$

Proof:

Rewrite and renaming.

□

Lemma 13.

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow \mathcal{L} \ = \ \mathcal{L}'$$

Proof:

By Lemma 10, we have that implementations of RCU that satisfy $R1$*,* $R2$*,* $R3$*, cannot make transitions to the* error *continuation; thus, those transitions are not realizable. The transitions to the* error *continuation are what distinguishes the instrumented semantics from the uninstrumented instructions.*

Then by Lemma 11, when the transitions to the error *continuation cannot be realized by a program* P*, the set of computations defining the execution of* P *using the instrumented semantics,* $\mathcal{L}'(P)$ *is equivalent to the set of computations from using the uninstrumented semantics,* $\mathcal{L}(P)$*. This gives the lemma:*

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow \mathcal{L} \ = \ \mathcal{L}'$$

□

Lemma 14.

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow P \ = \ P'$$

Proof:

Figure 4 illustrates the define/usage chains for the local RCU pointer when the constraints identified by restrictions $R1$*,* $R2$*, and* $R3$ *are satisfied.*

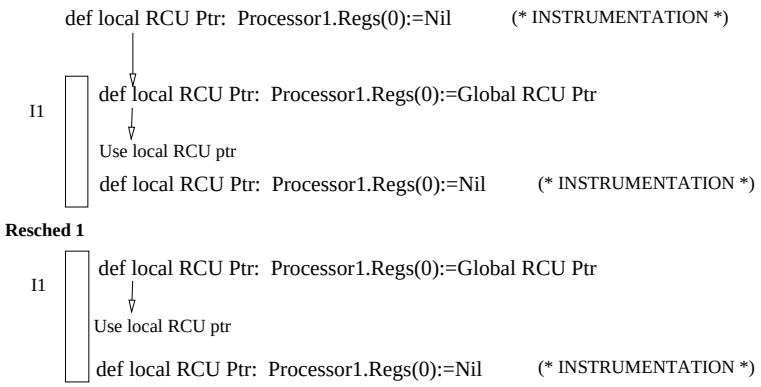

Fig. 4. $R(P) \Rightarrow P = P'$*. Definitions provided by the instrumented code are killed by definitions in the uninstrumented code. Between definitions there are no intervening usages; thus, the instrumented definitions can be eliminated as dead code.*

It is easy to see that instrumented definitions for the local RCU pointer can be eliminated as dead code[5] *since every "instrumented" definition is followed by another "uninstrumented" definition without an intervening usage[34]. Thus,*

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow P \ = \ P'$$

□

Lemma 15 (Fair executions of RCU implementations that satisfy restrictions $R1$, $R2$, and $R3$ do not collect live objects).

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow \forall \pi \in L(P)_{fair} \ \pi \models \Box \neg (collectLive \ 1)$$

Proof:

$R1 \ \wedge \ R2 \ \wedge \ R3$	*Assumption*
$\mathcal{L}(P) = \mathcal{L}'(P')$	*Assumption*, *Lemma* 13, *Lemma* 14
$\forall \gamma \in \mathcal{L}'(P')_{fair} \ \gamma \models \Box \neg (collectLive \ 1)$	*Lemma* 6
$\forall \pi \in \mathcal{L}(P)_{fair} \ \pi \models \Box \neg (collectLive \ 1)$	*Subst. "=" and renaming*

Discharge assumption:

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow \forall \pi \in L(P)_{fair} \ \pi \models \Box \neg (collectLive \ 1)$$

□

Theorem 3 (RCU implementations that satisfy restrictions $R1$, $R2$, and $R3$ do not collect live objects).

$$R1 \ \wedge \ R2 \ \wedge \ R3 \Rightarrow \forall \pi \in \Pi \ \pi \models \Box \neg (collectLive \ 1)$$

Proof:

The Fair Progress Constraint (Assumption 1) partitions the set of RCU computations such that every computation is either in the set of fair executions or it is not:

$$\forall \pi \in \Pi \ (\pi \in \Pi_{fair} \ \vee \ \pi \notin \Pi_{fair})$$

Since every $\pi \notin \Pi_{fair}$ does not collect storage

$$\forall \pi \in \Pi \ \pi \models \text{`}collect \ live \ object\text{`} \Rightarrow \pi \in \Pi_{fair}$$

Thus,to prove the theorem that RCU does not collect live objects, it is sufficient to prove that no fair execution of RCU collects a live object, that is

$$\forall \pi_1 \in \Pi_{fair} \ \pi_1 \models \Box \neg (collectLive \ 1) \Rightarrow \\ (\forall \pi_2 \in \Pi \ \pi_2 \models \Box \neg (collectLive \ 1))$$

Assuming $R1 \ \wedge \ R2 \ \wedge \ R3$, then the antecedent is given by Lemma 15; Modus Ponens and renaming gives:

[5] When translated to SSA form, definitions without a following use show up as assignments to variables that are never used.

$$\forall \pi \in \Pi\ \pi \models \Box \neg (collectLive\ 1)$$

Discharging the assumption gives the theorem:

$$R1\ \wedge\ R2\ \wedge\ R3 \Rightarrow \forall \pi \in \Pi\ \pi \models \Box \neg (collectLive\ 1)$$

□

Theorem 4 (RCU implementations that satisfy restrictions $R1$, $R2$, and $R3$ are Memory Safe).

$$R1\ \wedge\ R2\ \wedge\ R3 \Rightarrow \forall \pi \in \mathcal{L}(P)\ \pi \models \textit{Memory Safe}$$

Proof:

$R1\ \wedge\ R2\ \wedge\ R3$	*Assumption*
$\mathcal{L}(P) = \mathcal{L}'(P')$	*Assumption*, *Lemma* 13, *Lemma* 14
$\forall \gamma \in \mathcal{L}'(P')\ \gamma \models$ *Memory Safe*	*Modus Ponens*, *Assumption*, *Lemma* 10
$\forall \pi \in \mathcal{L}(P)\ \pi \models$ *Memory Safe*	*Subst. "=" and renaming*

Discharge assumption:

$$R1\ \wedge\ R2\ \wedge\ R3 \Rightarrow \forall \pi \in \mathcal{L}(P)\ \pi \models \textit{Memory Safe}$$

□

9 Conclusions

RCU has resisted rigorous analysis because existing semantic based approaches under approximate multiprocessor behavior and the algorithm relies on abstruse interactions with the operating system that aren't reconciled with language semantics. This work tackles the latter by incorporating the relevant operating system context switch behavior in a minimal language capable of expressing RCU. We define the formal semantics of this language using a small-step operational semantics that maps language constructs to configuration transformations. We express the live object and memory safety properties in linear temporal logic and use the sets of traces comprised of sequences of configurations that are created by simulated execution of the RCU program via the operational semantics as models to determine the validity of these properties. We instrument both the code and the language semantics to simplify the verification effort — we prove that instrumented RCU executing instrumented semantics does not collect live objects and is memory safe. We then introduce restrictions on the definition and use of the local RCU pointers that allow us to eliminate the instrumentation; thus, RCU implementations that conform to these restrictions do not collect live objects and are memory safe.

Our work advances the state-of-the-art by capturing the real behavior of an operating system context switch in a way that exposes the intermediate state of a system where one thread has been switched out and the next is yet to be switched

in. Our semantics captures this with the "R" configuration. We show that this intermediate state is used by RCU to achieve synchronization without the use of synchronization primitives because the operating system blocks execution until a particular event occurs.

We formalize restrictions for the definition and use of the RCU local pointer. These restrictions can help certify RCU implementations and are readily accommodated by static analysis tools.

This work demonstrates that operational verification techniques are useful for proving liveness and safety properties of concurrent, reactive programs. We give, for the first time, proofs that RCU does not collect live objects and that it is memory safe.

Our proofs that RCU does not collect live objects and is memory safe use a novel strategy. We instrument the code to make it simple to specify and prove the properties. We next introduce restrictions on the program code that allows us to remove the instrumentation; thus, code conforming to the restrictions has the properties. This technique proved especially valuable in showing that RCU is memory safe.

We confirmed our intuition that an operational semantics based on state transformation is easy for engineers to understand — as part of the review process, we invited experts in RCU multiprocessor programming to look at our RCU implementation and the language semantics. They readily understood the RCU implementation and were able to follow the machine language semantics. They confirmed that our implementation was "correct" and that our semantics were realistic[6].

10 Future work

Our processor model and language included only those "features" necessary to prove the live object and memory safety properties of RCU "Classic" executing on a sequentially consistent multiprocessor. However, modern processors are not sequentially consistent and deployed RCU implementations protect recursive data structures such as a linked list. RCU linked lists are especially interesting because they satisfy a safe-traversal property: RCU accessors traversing a linked list always reach a next node or the end of the list. Other non-blocking synchronization techniques such as software transactional memory[47, 13, 12], hazard pointers[33], and lock-free approaches[14, 7, 8] do not satisfy this property.

To develop appropriate machine abstractions for weakly consistent architectures, we are studying how modern processors implement strong serialization properties akin to acquire/release semantics[10, 11] and in particular how locks and memory barriers ensure the visibility of the global RCU pointer. At this stage, we are confident in the description of the weakest but still useful memory model[9].

[6] This tells why it is important to have a specification.

A natural next step is to extend our operational verification approach to RCU linked list implementations that execute on weakly (the weakest) consistent memory architectures.

References

1. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Proceedings of Formal Underpinnings of Java Workshop, OOPSLA '98*, 1998.
2. T. Axford. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. Wiley Series in Parallel Computing. Wiley, 1990.
3. B. A. Bowen and R. J. A. Buhr. *The Logical Design of Multiple-Microprocessor Systems*. Prentice-Hall, 1980.
4. D. Comer. *Operating System Design – The XINU Approach*. Prentice-Hall, 1984.
5. M. Compton. Stenning's protocol implemented in udp and verified in isabelle. In *CATS '05: Proceedings of the 2005 Australasian symposium on Theory of computing*, pages 21–30, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
6. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, pages 996–1072. Elsevier Science Publishers / MIT Press, 1990.
7. K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University, September 2003.
8. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computers*, 25(2), May 2007.
9. M. Frigo. The weakest reasonable memory model. Master's thesis, Massachutes Institute of Technology, 1997.
10. K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Stanford University, Dept of EE and CS, December 1995. http://www.hpl.hp.com/techreports/compag-dec/WRL-95-9.pdf.
11. K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.
12. T. Harris, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Conference on Principles and Pracitce of Parallel Programming*, Chicago, Illinois, USA, June 2005.
13. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.
14. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
15. G. Holzman. *Spin Model Checker*. Wiley, 2004.
16. P. Howard, J. Walpole, P. McKenney, and J. Triplett. The case for relativistic programming. Technical report, Portland State University, January 2009.
17. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.
18. L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28:690–691, 1979.

19. L. Lamport. What good is temporal logic? In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress*, pages 657–686, Amsterdam, 1983. North Holland.
20. H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. In *Theorem Proving in Higher Order Logics (TPHOLS 04*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200. Springer-Verlag, 2004.
21. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems — Specification*. Springer-Verlag, 1992.
22. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems — Safety properties*. Springer-Verlag, 1995.
23. P. McKenney. Hierarchical RCU. *Linux Weekly News*, November 2008.
24. P. McKenney and J. Walpole. What is RCU, fundamentally? *Linux Weekly News*, December 2007.
25. P. McKenney and J. Walpole. What is RCU? part 2: Usage. *Linux Weekly News*, December 2007.
26. P. McKenney and J. Walpole. What is RCU? part 3: The RCU API. *Linux Weekly News*, January 2008.
27. P. E. McKenney. Read-Copy Udate Usage. http://www.rdrop.com/users/paulmck/RCU/linuxusage.html — this site is updated quarterly.
28. P. E. McKenney. Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels. *Ph.D. Dissertation; OGI*, 2004. http://www.rdrop.com/users/paulmck/RCU.
29. P. E. McKenney. Using Promela and Spin to verify parallel algorithms. Available: http://lwn.net/Articles/243851/ [Viewed September 8, 2007], August 2007.
30. P. E. McKenney and S. Rostedt. Integrating and validating dynticks and preemptable rcu. Available: http://lwn.net/Articles/279077/ [Viewed April 24, 2008], April 2008.
31. J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 106–113, New York, NY, USA, 1991. ACM Press.
32. J. Meseguer. Rewriting as a unified model of concurrency. In *OOPSLA/ECOOP '90: Proceedings of the workshop on Object-based concurrent programming*, New York, NY, USA, 1991. Association for Computing Machinery.
33. M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.
34. S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
35. H. R. Nielson and F. Nielson. *Semantics With Applications*. John Wiley & Sons, 1992. Revised 1999 – `http://www.www.daimi.au.dk/~hrn`.
36. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.
37. G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, September 1976.
38. G. D. Plotkin. Structural operational semantics. *Lecture Notes DAIMI FN-19*, 1981.
39. G. D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.
40. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977.

41. W. Pugh, J. Mason, and S. V. Adve. The java memory model. *POPL*, pages 378–391, January 2005.
42. T. Ridge. Operational reasoning for concurrent caml programs and weak memory models. In *TPHOLs*, pages 278–293, 2007.
43. S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–391, New York, NY, USA, 2009. ACM.
44. D. Sarma and K. Thomas. Read Copy Udate HOWTO. *Linux Documentation*, 2001. http://lse.sourceforge.net/locking/rcu/HOWTO/index.hmtl.
45. S. Schacht. Proving properties of actor programs using temporal logic. In *In G. Agha and F. De Cindo (Eds.), Proc. of the workshop on object-oriented programming and models of concurrency*, 1995.
46. F. B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. Springer-Verlag, 1997.
47. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.