Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms

Mathieu Desnoyers, Paul E. McKenney, Michel R. Dagenais

***Abstract*—Modeling parallel algorithms at the architecture level permits to explore side-effects of weak ordering performed by modern processors. Formal verification of such models with model-checking can ensure that algorithm guarantees will hold even in the presence of the most aggressive compiler and processor optimizations.**

This paper proposes a virtual architecture to model the effects of such optimizations. It first presents the *OoOmem* framework to model out-of-order memory accesses. It then presents the *OoOisched* framework to model the effects of out-of-order instruction scheduling.

These two frameworks are explained and tested using weakly-ordered memory interaction scenarios known to be affected by weak ordering. Then, modeling of user-level RCU (Read-Copy Update) synchronization algorithms is presented. It uses the virtual architecture proposed to verify that the RCU guarantees are indeed respected.

***Index Terms*—C.0.d Modeling of computer architecture < 0. General < C. Computer System Organization, C.1.2.g Parallel processors < C.1.2 Multiple Data Stream Architectures (Multiprocessors) < C.1 Processor Architectures < C. Computer System Organization, D.1.3. Concurrent Programming < D.1 Programming Techniques < D. Software/Software Engineering, D.2.4.d Formal Methods < D.2.4 Software/Program Verification < D.2 Software Engineering < D. Software/Software Engineering, D.2.4.e Model Checking < D.2.4 Software/Program Verification < D.2 Software Engineering < D. Software/Software Engineering, D.4.1.f Synchronization < D.4.1 Process Management < D.4 Operating Systems < D. Software/Software Engineering D.4.5.f Verifcation < D.4.5 Reliability < D.4 Operating Systems < D. Software/Software Engineering <,**

I. Introduction

FORMAL verification of synchronization primitives for shared memory multiprocessor architectures is undoubtedly useful due to the architecture and context dependency of bug occurrence. An algorithm can appear bug-free when used in a large set of test cases. However, testing is unable to certify that compiler optimizations done for a different invocation of a synchronization primitive will work as expected. Portability is also hard to certify with testing, because it would involve testing the primitives on all architecture variants.

Our principal motivation is to create detailed architecture-level models taking into account weak instruction scheduling and memory access ordering able to verify the user-space RCU (Read-Copy Update) implementations presented in (1). The complexity level of these algorithms makes formal verification well worthwhile.

This paper first depicts the modeling challenges, then summarizes the LTL model-checking principles and introduces to modeling of parallel algorithms. This is followed by a presentation of the frameworks created to accurately model real architectures. Finally, the RCU library modeling and its verification are presented.

II. Modeling Challenges

Modeling multi-core systems for formal verification of parallel algorithm implementations brings interesting challenges. These are caused by the architecture and compiler ordering semantics. These challenges come from the presence of:

- compiler-level optimizations,
- execution of nested signal handlers,

on architectures with the following characteristics:

- shared-memory multiprocessor,
- weak memory-ordering,
- pipelined and superscalar,
- out-of-order instruction scheduling.

All these challenges are a direct result of our desire to model algorithms for production use. If this was instead meant to be only used for prototyping, we might be strongly tempted to assume a non-optimizing compiler and almost inexistent sequentially consistent machines to avoid dealing with optimization, out-of-order execution and out-of-order memory access effects. We might also be tempted to assume signal handlers out of existence as a simplification.

Concurrent algorithms have been modeled on weakly ordered systems in the past (2), and interrupts (similar to signals) have been modeled as well (3). However, the methods used in this past work to model weakly-ordered systems can result in combinatorial explosion of the model. Models specifically covering the x86 (4), as well as PowerPC and ARM (5) architectures for parallel algorithm verification has also been proposed in the past. In this paper, we present a more general approach that allows the model to more closely follow the architecture behavior than the previous RCU models and to take into account the weakest ordering amongst multiple architectures, therefore modeling weak-ordering effects more accurately.

To further reduce the computational requirements of the validation process, we approximate the properties of the actual hardware, but in all cases modeling weaker ordering than the actual hardware provides. This weaker-ordering approximation

Manuscript received July X, 2009; revised Month Y, 2009

M. Desnoyers (mathieu.desnoyers@polymtl.ca) and M. R. Dagenais (michel.dagenais@polymtl.ca) are with the Computer and Software Engineering Department, Ecole Polytechnique de Montreal.

Paul E. McKenney (paulmck@linux.vnet.ibm.com) is with the IBM Linux Technology Center.

ensures that any algorithm passing our validation will run correctly on conforming hardware.

We propose a model representing the interprocessor interactions, which we call our *virtual architecture*. In this architecture, each Promela process represents a processor. To model `CISC` architectures, complex instructions are divided into micro-operations, which are then represented as atomic Promela statements. It results in a `RISC` architecture, which allows to easily detail micro-operations dependencies. Through this article, micro-operation and instruction will be used as synonyms, given those apply to our RISC virtual architecture.

The models proposed here specifically use a data flow representation of each processor, where each node repesents a micro-operation and where each arc represents a data or control dependency between micro-operations. This permits to model accurately all possible micro-operation interleavings as seen from the point of view of their visible effect outside of the processor. We model the interactions between processors by creating a cache-memory interaction model.

III. Modeling and Model-Checking

This section summarizes the principles of `LTL` model-checking and presents an introduction to modeling of parallel algorithms. It constitutes the background on which is constructed the rest of this article.

A. LTL Model-Checking

The verification is carried out using Promela (6), a special-purpose modeling language that performs a full state-space search. This in turn allows all possible execution histories to be examined for specified classes of errors, including race conditions and livelock conditions.

The equivalence between data flow analysis and model-checking of abstract interpretations has been used (7; 8) to model simple sequential programs. Data flow analysis has also been used to verify properties of concurrent programs (9).

We represent our model in Promela and perform verification of properties expressed as `LTL` (Linear Temporal Logic) formulas[1]. The model description expresses atomic statements executed by one or more processes. The Spin verifier (10; 11) transforms the model into a Büchi (12) automaton. The `LTL` formulas are transformed in the negation of never-claims suited for verification of the model. The Spin verifier visits all atomic statements required to validate the `LTL` claims in all execution orders allowed by the model. One sequence used to visit atomic statements in a particular order forms a path.

`LTL` formalism allows basic logical operators within predicates:

- $\Rightarrow$: logical implication,
- $\Longleftrightarrow$: equivalence,
- $\wedge$: conjunction,
- $\vee$: disjunction,
- $\neg$: negation.

To reason over the future of paths, temporal operators can be applied to predicates:

- **G**: Temporal operator *always* (a predicate will always hold),
- **F**: Temporal operator *eventually* (a predicate must eventually become true),
- **U**: Strong until (will hold until another predicate is true).

Model checking permits to explore all execution scenarios required to verify a specific `LTL` property. Unlike simulation-based approaches, the model-checker only needs to generate the states required to verify the property, rather than performing an exhaustive state-space exploration. Each Promela process being represented as an automaton, we can represent the complete state-space generated by parallel processes by performing the product of these automatons.

One major limitation is that `LTL` model-checking is pspace-complete. Reasoning about specific predicates to verify, allows limiting the state-space to the subset required to verify the predicates. This is why Spin enhances state-space exploration with lossless compression techniques based on the characteristics of the claims validated. For instance, *Partial Order Reduction* (13) permits to merge states for which the partial order does not affect the property to verify.

B. Introduction to Parallel Algorithm Modeling

As an introductory example, let us consider the verification of the busy-waiting lock primitives, usually known as *spinlock*, present in the PowerPC and Intel architectures of Linux kernel 2.6.30.

The spinlock implementation found in the PowerPC architecture is relatively straightforward: it consists of two states, either 0 or non-zero. It uses the "`lwarx`" (Load Word and Reserve Indexed) and "`stwcx.`" (Store Word Conditional Indexed) instructions to atomically compare and update the lock value. The store only succeeds if the memory location has not been updated since the load.

As an example, a Promela model of this locking primitive is presented in Figure 1. Line 1 contains the lock variable definition, followed by a data access reference count defined in Line 2. Lines 4–17 contain the spin lock primitive. The `inline` function in Promela is close to that of the C language, except that such functions in Promela have type-free arguments, are not permitted to contain declarations, and do not return any value. Lines 6–16 contain the busy-waiting loop `do ... od`, stopped only by the `break` statement on Line 13 if the variable `lock` is 0. The `skip` statement on Line 10 is an empty statement. It has no effect other than permitting to follow the Promela grammar. Line 7 begins with "`:: 1 ->`", which indicates a condition always fulfilled. This lets the statements following the "`->`" execute unconditionally. Line 7 ends with a very important keyword: `atomic`. It precedes a sequence of statements, contained within brackets, which is considered as indivisible. They are therefore executed in a single execution step, similarly to an atomic instruction executed by a processor. Lines 19–22 contain the unlock function. Lines 24–33 contain the body of the processes, which takes a spinlock, increments and decrements the reference count, and releases the lock in an infinite loop. Two instances

[1]See the Promela reference for equivalent `LTL` symbolism http://spinroot.com/spin/Man/ltl.html

```
 1 byte lock = 0;
 2 byte refcount = 0;
 3
 4 inline spin_lock(lock)
 5 {
 6   do
 7   :: 1 -> atomic {
 8       if
 9       :: lock ->
10           skip;
11       :: else ->
12           lock = 1;
13           break;
14       fi;
15     }
16   od;
17 }
18
19 inline spin_unlock(lock)
20 {
21   lock = 0;
22 }
23
24 proctype proc_X()
25 {
26   do
27   :: 1 ->
28        spin_lock(lock);
29        refcount = refcount + 1;
30        refcount = refcount - 1;
31        spin_unlock(lock);
32   od;
33 }
34
35 init
36 {
37   run proc_X();
38   run proc_X();
39 }
```

Fig. 1. Promela Model for PowerPC Spinlock

of the process are run upon initialization by `init` at Lines 35–39.

This Promela code is represented by the diagram found in Figure 2. Each node represents a Promela statement. A name is added to most nodes to make interpretation easier. Each node contains a Promela statement. Arrows connecting the nodes represent how the model-checker can move between nodes. Some require conditions to be active, e.g. `(lock == 1)`, to allow moving to the target node. The `STEP++` statements on the arrows represent that the execution counter is incremented. A concurrent process may run between different steps, but not while `STEP` stays invariant. The latter scenario happens in the `ATOMIC` box, which represents the atomic sequence of statements.

Safety of this locking primitive is successfully verified by the Spin model-checker by verifying that the reference count value is never higher than 1. This is performed by prepending `#define refcount_gt_one (refcount > 1)` to the model and by using the following `LTL` formula. PowerPC spinlock safety is verified by the `LTL` claim:

$$\mathbf{G}\ (\neg\ \text{refcount\_gt\_one}\ )$$

However, one major downside of this spinlock implementation is its lacks of fairness. A CPU always acquiring the same spinlock in a loop could effectively starve other CPUs. This can be verified using the following `LTL` formula:

$$\mathbf{G}\ (\ \mathbf{F}\ (\neg\ \text{np\_}\ ))$$

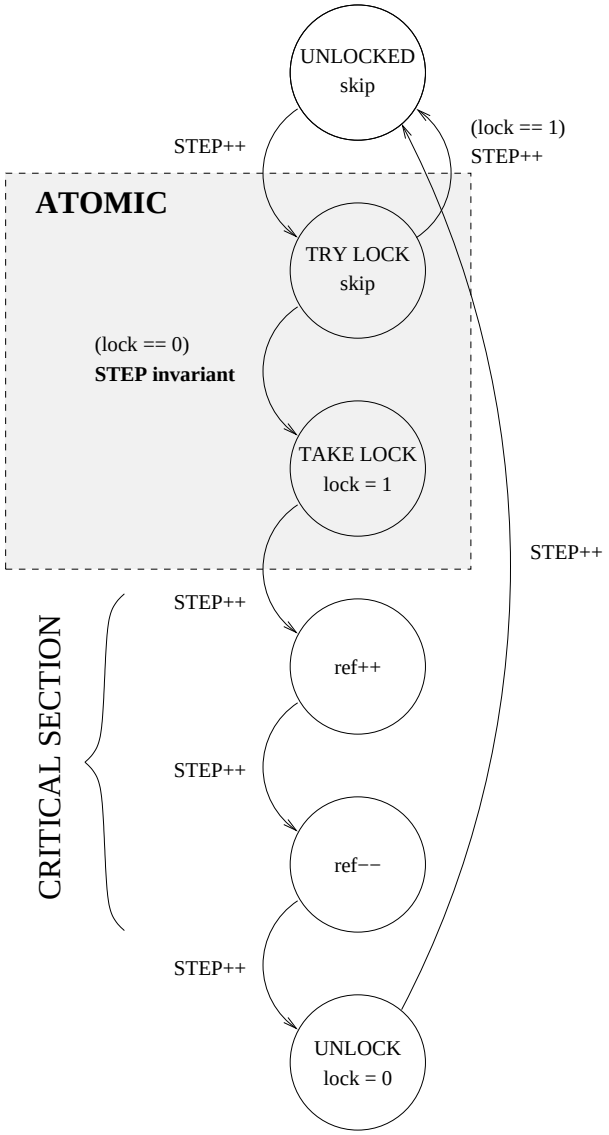

Fig. 2. Diagram Representation of PowerPC Spinlock Model

The keyword `np_` has a special meaning: it is true if all system states do not correspond to progress states. We modify the code from Figure 1 to insert such progress states: this involves separating the process body in two different definitions to add a `progress:` keyword within the infinite loop in one of them. Using the Spin verifier *weak fairness* option lets it detect non-progress cycles involving more than one process. This corresponds to starvation of a process by one or more other processes.

Ticket spinlocks used for the Intel spinlock implementation found in Linux correct the fairness problem identified in the PowerPC architecture. The Promela implementation is omitted due to space considerations, but the state diagram is presented at Figure 3. The new elements added to this graphs are the `LOW_HALF()` and `HIGH_HALF()` primitives, which select half lower and upper bits of the lock, respectively.

The Spin model-checker verifies that this model is safe and fair, under certain conditions. Changing the number of bits available for the low and high halves of the ticket lock as

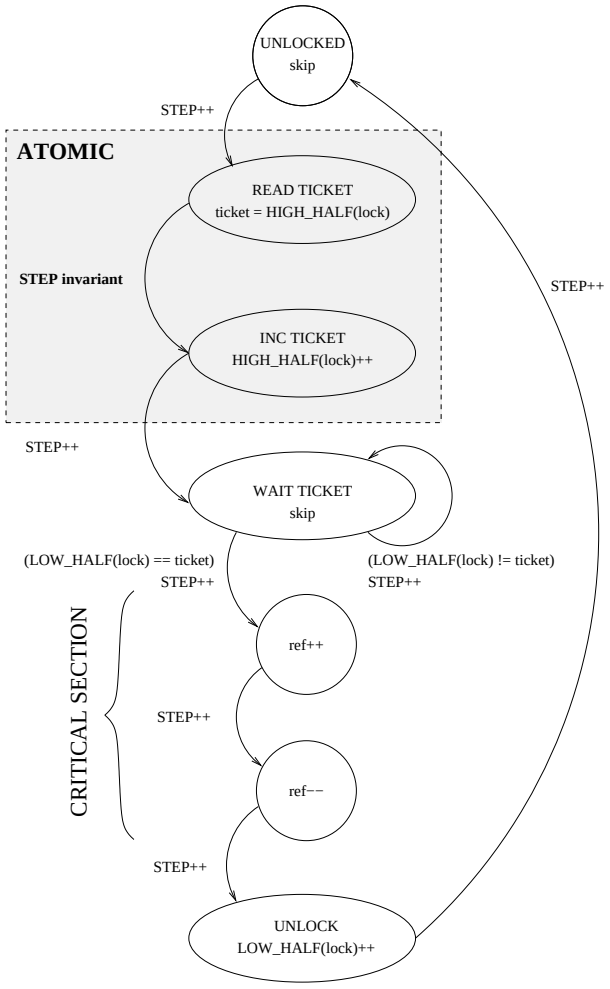

Fig. 3. Diagram Representation of Intel Ticket Spinlock Model

well as the number of processes shows that fairness is only ensured when the number of processes fits in the number of bits available for each half.

IV. Weakly-Ordered Memory Framework

A. Architecture

Modeling out-of-order memory accesses performed by processors at a level consistent with their hardware implementation is important to enable accurate modeling of side-effects that can be caused by missing memory barriers in synchronization algorithms. The bugs within this category are hard to reproduce, mainly because they are dependent on the architecture, execution context and timing between the processors. Therefore, testing the implementation might not be sufficient to certify the absence of bugs.

To model an algorithm including the effects of the memory barriers required on various architectures, or more importantly lack thereof, we choose to create a virtual architecture performing the most aggressive memory reordering. The Alpha 21264 seems to be an especially interesting architecture with respect to reordering, given its ability to reorder dependent loads (14; 15). It also reorders loads after stores, stores after loads, loads after loads and stores after stores. Even atomic operations can be reordered with respect to loads and stores.

The Alpha architecture can reorder dependent loads in addition to other memory accesses due to the design of its caches. These are are divided into banks, which can each communicate with main memory. If the channel between a bank and memory is saturated, the updates of the less busy channels could reach their destination before loads or stores initiated earlier. Such extremely weak ordering can therefore cause any sequence of loads and stores to different addresses, including dependent loads, to be perceived in a different order from the point of view of another processor. Indeed, when memory content changes, in-cache view updates are not guaranteed to reach the cache-lines in the same order the main memory stores appear.

We name the weak memory ordering part of our virtual architecture *OoOmem*, where *OoO* stands for *Out-of-Order*. It models the exchanges between CPU cache-lines and main memory. To model the worse possible case, each variable belongs to a different cache-line and there is only one cache-line per bank. These cache-lines are therefore free to be updated (or not) with main memory between each instruction execution.

Each CPU is modeled as a Promela process. Each variable is represented with a main memory entry, a per-CPU entry and a per-CPU dirty flag. Operations and accesses to these memory entries are performed through Promela macros created as part of the *OoOmem* framework to facilitate manipulation of the variables. We call these the “primitives” of the *OoOmem* framework. For each cache-local variable, the `ooo_mem()` primitive models the case where it is updated as well as the case where it is not. This primitive must be called between each cache-line access. This causes all possible interleaving of exchanges between caches and memory to appear in the set of generated execution traces.

Explicit memory ordering of loads and stores can be respectively forced by using the primitives `smp_rmb()` and `smp_wmb()`. These memory barriers respectively send the cache-local stores to memory and ensure that all in-memory data is loaded into the local cache. The per-variable, per-CPU dirty flag makes sure that a given CPU fetches data that it has recently written from its emulated cache rather than fetching stale data from main memory.

The primitive `write_cached_var()` updates the cache-local version of a variable with a new value and sets the dirty flag for this variable on the local CPU. The dirty flag will let `ooo_mem()` and `smp_wmb()` subsequently commit this change to main memory. In addition, this ensures that neither `ooo_mem()` nor `smp_rmb()` overwrite the cache-local version before it has been committed to memory.

The read-side equivalent is `read_cached_var()`, which loads a cache-local variable into the local process variables.

Modeling other architectures such as the Intel, PowerPC and Sparc processor families, which do not permit to reorder dependent loads, only requires replacing the conditional load part of the `ooo_mem()` primitive by a call to `smp_rmb()`. As a result, the local cache is unconditionally updated by loading all main memory variables into the non-dirty local cache-

lines. The effects of independent loads reordering done by these architectures is modeled by the *Out-of-order Instruction Scheduling Model* presented at Section V.

B. Testing

In order to validate the accuracy of the framework, we use architectural litmus tests for which results are known. One such litmus test involves processor A performing consecutive updates to a pair of variables, while processor B concurrently reads these same variables in reverse order. We expect that if the ordering is correct, whenever processor B sees the updated second variable, it must eventually read the updated first variable. This is expressed in the following pseudo-code and `LTL` formula:

Pseudo-code modeled:

```
alpha = 0;
beta = 0;

Processor A              Processor B
alpha = 1;               x = beta;
smp_wmb();               smp_rmb();
beta = 1;                y = alpha;
```

`LTL` claim to satisfy:

$$\mathbf{G}\ \big(x = 1 \Rightarrow \mathbf{F}\ (y = 1)\big)$$

This model is verified successfully by the Spin verifier. Error-injection is performed to ensure that the verifier would appropriately generate the erroneous execution traces if the required memory barriers were missing. This is performed by removing the `smp_wmb()` from Processor A or `smp_rmb()` from Processor B. Verifying these two altered models shows the expected errors and execution traces: variables being either stored to or loaded from memory in the wrong order fails to verify the `LTL` claim.

V. Out-of-Order Instruction Scheduling Framework

Although the *OoOmem* framework presented earlier represents exchanges between cache and memory accurately, it does not reproduce all reordering performed at the processor level regarding out-of-order micro-operation (`RISC` instruction) scheduling. This section explains how we model these effects.

A. Architecture

Superscalar pipelined architectures leverage instruction-level parallelism by allowing multiple instructions to start concurrently and by reordering instruction completion. It can, more aggressively, reorder the sequence in which independent instructions are issued. Speculative execution can also cause execution of instructions before their result is proven to be needed. Such out-of-order instruction execution can be seen on the Alpha 21264 (15).

Our virtual architecture framework for out-of-order instruction scheduling therefore encompasses all possible instruction scheduling which can be done by either compiler optimizations (lifting, combining reads, re-loading a variable to diminish register pressure, etc.) or the processor. The weakest scheduling possible is bounded by the dependencies between instructions. In order to let the verifier explore all possible execution orders, our virtual processor framework, *OoOisched*, provides:

- an infinite number of registers,
- a pipeline with an infinite number of stages,
- a superscalar architecture able to fetch and execute an infinite number of instructions concurrently,
- and the ability to perform speculative instruction execution when they have no side-effect on cache.

As in the *OoOmem* framework, one Promela process represents one processor. A key element of this framework is to have a compact instruction execution state and dependency representation. We choose to use a per-processor set of tokens to represent the dependencies with a single token per instruction. Tokens are produced by executing instructions and typically cleared only at the end of the instruction sequence. The conditions required to activate an instruction are represented by a set of tokens. Each token can be represented by a single bit.

As an example, the dependencies of the test-case presented in Section V-B are modeled in the Promela listing in Figure 4 and illustrated in Figure 5.

An instruction scheduling loop tests for every instructions dependency constraints to execute them. Execution of instruction is non-deterministic: when the dependencies of multiple instructions are met, any one of them may fire, but does not have to. It therefore explores all the possible execution orderings fitting within the dependency constraints. It proceeds until the end of the loop, which consists in executing the last instruction of the sequence. This last instruction clears all the tokens and breaks the instruction scheduling loop. When the bit allocated for an instruction is enabled, it inhibits its execution and enables its dependent instructions. The state of each CPU's execution is kept in a per-process data structure containing the current execution state tokens.

In the Promela model presented in Figure 4, the macro `CONSUME_TOKENS(tokens, enable, inhibit)` is used as trigger to execute an instruction. The parameter *tokens* is the token container of the current processor. The scheduler is allowed to execute an instruction only if all the *enable* tokens are active and all the *inhibit* tokens are inactive. Its role is to check for pre-conditions for instruction execution, but does *not* clear any token.

The macro `PRODUCE_TOKENS(tokens, prod)` adds tokens identified by *prod* to *tokens*. It is typically used at the end an instruction execution by producing its own token. Finally, `CLEAR_TOKENS(tokens, clear)` clears all the specified CPU tokens. It is typically used after the last instruction of the scheduling loop, but can also be used to partially clear the token set to produce loops.

The diagram representing the Promela model in Figure 5 represents each instruction by a node. White arrows represent unmet dependencies and black arrows correspond to dependencies met. Colored nodes are those currently candidate for execution: all their dependencies are met, which means that all tokens they consume are enabled, and all the tokens that

inhibit their execution are cleared. The column on the left represent the tokens associated with each instruction.

We choose this representation of the data and control flow rather than more classic token-based models like Petri networks (16) or coloured Petri networks (17) to allow easy injection of faults in the model. This would be cumbersome to do with a classic representation where one instruction would produce a token that would be later consumed by a following instruction. For instance, removing a read barrier or write barrier from the model would require to completely modify the dependency graph of the following instructions to make sure they now depend on prior instructions. Failure to do so would create an artificial synchronization barrier which would not model the error injection correctly. Since the token model provides the complete list of instruction dependencies, errors can be injected by enabling the token corresponding to the instructions to disable before entering the instruction scheduling loop. The effect is the inhibition of the instruction and satisfaction of all dependencies normally met when this instruction is executed.

Given our virtual architecture models all possible sequences of code execution allowed by the data dependencies, only a few specific issues must be addressed to make sure compiler optimizations are taken into account. In our framework, a temporary per-process variable, corresponding to a processor register, should never be updated concurrently by multiple instructions. SSA (Static Single Assignment) (18) is an intermediate representation typically used in compilers where each variable is assigned exactly once. Using such representation for registers would ensure to have no more than a single instruction using a temporary register, but this would cost additional state-space. Given this resource is limited, we re-use registers outside of their liveness region. There are only two cases where we expect the compiler to re-use the result of loads. The first case is when the compiler is instructed to perform a single `volatile` access to load the variable to a register. The second case is when an explicit compiler barrier is added between the register assignment (load from cache) and register use. In all other cases, re-use of loaded variables will be taken into account by performing the two loads next to each other due to speculative execution (prefetching) support in the scheduler.

One limitation of this framework is that it adds an artificial compiler barrier and core synchronization between consecutive instruction scheduler executions. This would not take into account side-effects caused by scheduler execution within a loop. This is caused by the instruction's inability to cross the artificial synchronization generated by the last instruction executed at the end of the scheduler loop. This last instruction is required to clear all tokens before the next execution. Such effect can be modeled by unrolling the loop.

Because the token container is already occupied by the outermost execution of the scheduler, recursion is also not handled by the framework. A supplementary container could be used to model the nested execution. However, using a different instruction scheduler for the nested context would fail to appropriately model interleaving of instructions between different nesting levels. Therefore, nested calls must be expanded into the caller site.

B. Testing

Before introducing the more complex `RCU` model, we present a test model for the *OoOisched* framework. This model is based on both the *OoOisched* and *OoOmem* frameworks. It models an execution involving two processors and two memory locations. In this model, Processor A successively writes to `alpha` and reads `beta`. Processor B successively writes to `beta` and then reads `alpha`. This verifies that at least one processor reads the updated variable. This is shown in the following pseudo-code. Dependency constraints applied on instructions executed by Processor A are illustrated by Figure 5.

Pseudo-code modeled:

```
alpha = 0;
beta = 0;
x = 1;
y = 1;

Processor A              Processor B
alpha = 1;               beta = 1;
smp_mb();                smp_mb();
x = beta;                y = alpha;
```

`LTL` claim to satisfy:

$$\mathbf{G}\ (x = 1 \vee y = 1))$$

This model is successfully verified by the Spin verifier. Error-injection is performed to ensure that the verifier would appropriately generate the erroneous execution traces if the required memory barriers were missing. This is performed by either:

- completely removing the `smp_mb()`,
- removing only the `smp_rmb()` part of the barrier,
- removing only the `smp_wmb()` part of the barrier,
- removing the implicit core synchronization provided by the `smp_mb()` semantics, which leaves the reads and the writes free to be reordered.

Verifying these altered models shows the expected errors and execution traces, where the read or write instructions being reordered fails to verify the `LTL` claims.

Removing core synchronization from the model presented in Section V-B permits verifying its behavior when injecting errors. The diagram presented in Figure 6 shows a snapshot of instruction execution with core synchronization removed. It shows that two instructions are candidate for execution: either `alpha = 1` or `x = beta`. It this case, the store and load can be performed in any order by the instruction scheduler.

As an example of the result of an error-injection, we present an execution trace generated by the Spin model-checker. We use the test model presented in Figure 4 with core synchronization removed. This partial execution trace excludes the empty execution and `else`-statements for conciseness. Some statements are also folded.

Lines 6–8 show the instruction scheduler from processor B scheduling the two first instructions of this processor: production of the initial token and error-injection by producing the

```
 1 #define PA_PROD_NONE    (1 << 0)
 2 #define PA_WRITE        (1 << 1)
 3 #define PA_WMB          (1 << 2)
 4 #define PA_SYNC_CORE    (1 << 3)
 5 #define PA_RMB          (1 << 4)
 6 #define PA_READ         (1 << 5)
 7
 8 byte pa_tokens;
 9
10 active proctype processor_A()
11 {
12   PRODUCE_TOKENS(pa_tokens, PA_PROD_NONE);
13
14   do
15   :: CONSUME_TOKENS(pa_tokens,
16                     PA_PROD_NONE, PA_WRITE) ->
17       ooo_mem();
18       WRITE_CACHED_VAR(alpha, 1);
19       ooo_mem();
20       PRODUCE_TOKENS(pa_tokens, PA_WRITE);
21   :: CONSUME_TOKENS(pa_tokens,
22                     PA_WRITE, PA_WMB) ->
23       smp_wmb();
24       PRODUCE_TOKENS(pa_tokens, PA_WMB);
25   :: CONSUME_TOKENS(pa_tokens,
26                     PA_WRITE | PA_WMB,
27                     PA_SYNC_CORE) ->
28       PRODUCE_TOKENS(pa_tokens, PA_SYNC_CORE);
29   :: CONSUME_TOKENS(pa_tokens,
30                     PA_SYNC_CORE, PA_RMB) ->
31       smp_rmb();
32       PRODUCE_TOKENS(pa_tokens, PA_RMB);
33   :: CONSUME_TOKENS(pa_tokens,
34                     PA_SYNC_CORE | PA_RMB,
35                     PA_READ) ->
36       ooo_mem();
37       pa_read = READ_CACHED_VAR(beta);
38       ooo_mem();
39       PRODUCE_TOKENS(pa_tokens, PA_READ);
40   :: CONSUME_TOKENS(pa_tokens,
41                     PA_PROD_NONE | PA_WRITE |
42                     PA_WMB | PA_SYNC_CORE |
43                     PA_RMB | PA_READ, 0) ->
44       CLEAR_TOKENS(pa_tokens,
45                    PA_PROD_NONE | PA_WRITE |
46                    PA_WMB | PA_SYNC_CORE |
47                    PA_RMB | PA_READ);
48       break;
49   od;
50 }
```

Fig. 4. OoO Instruction Scheduling and Memory Frameworks Promela Test Code, Processor A

core synchronization token ahead of the instruction scheduler execution.

```
 6:    proc  1 (CPU_B) line 252 "mem.spin" (state 2)
             [pb_tokens = (pb_tokens|(1<<PA_PROD_NONE))]
 8:    proc  1 (CPU_B) line 261 "mem.spin" (state 3)
             [pb_tokens = (pb_tokens|(1<<PA_SYNC_CORE))]
```

Line 10 presents the trigger permitting activation of the write instruction.

```
10:    proc  1 (CPU_B) line 265 "mem.spin" (state 4)
             [(((!((pb_tokens&(1<<PA_WRITE)))
              && ((pb_tokens&(1<<PA_PROD_NONE))
                  ==(1<<PA_PROD_NONE))))]
```

Lines 19–21 show processor B updating its alpha and beta cache-view from memory. Processor B performs a random cache update. In this execution trail, it loads alpha and beta into its local cache.

```
19:    proc  1 (CPU_B) line 125 "mem.spin" (state 30)
             [(!(cache_dirty_alpha[_pid]))]
19:    proc  1 (CPU_B) line 125 "mem.spin" (state 31)
               [cached_alpha[_pid] = mem_alpha]
21:    proc  1 (CPU_B) line 126 "mem.spin" (state 41)
               [(!(cache_dirty_beta[_pid]))]
21:    proc  1 (CPU_B) line 126 "mem.spin" (state 42)
```

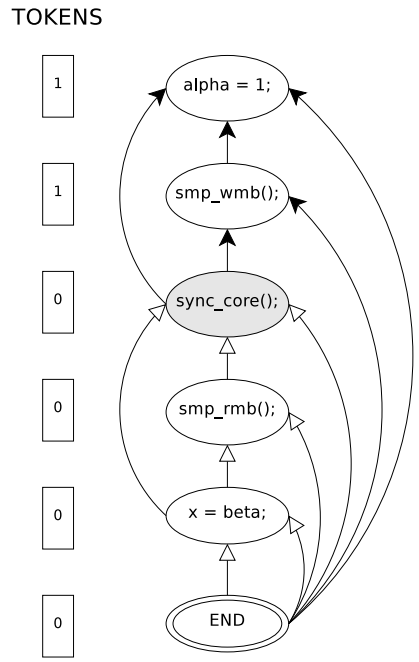

Fig. 5. Instruction Dependencies of Out-of-Order Instruction Scheduling and Memory Framework Test

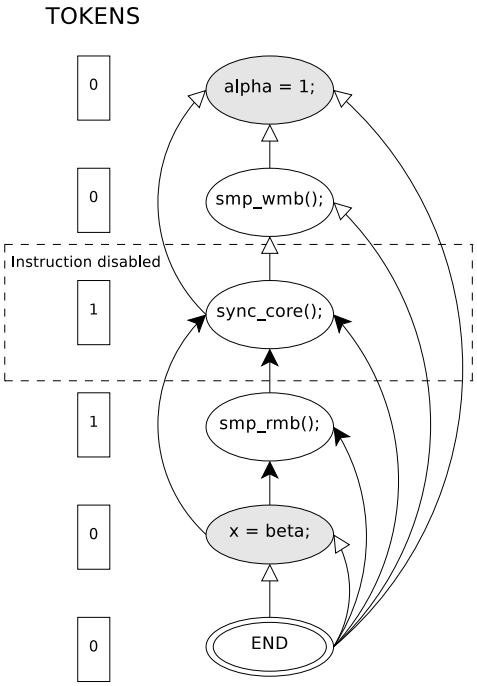

Fig. 6. Instruction Dependencies of Out-of-Order Instruction Scheduling and Memory Framework Test (Error Injection)

```
                [cached_beta[_pid] = mem_beta]
```

At Line 23, processor B writes to its cache view of beta and sets the matching dirty flag.

```
 23:    proc  1 (CPU_B) line 267 "mem.spin" (state 53)
                [cached_beta[_pid] = 1]
 23:    proc  1 (CPU_B) line 267 "mem.spin" (state 54)
                [cache_dirty_beta[_pid] = 1]
```

A succinct high-level summary of the execution trail follows:

```
 CPU B: writes to in-cache beta
 CPU A: writes to in-cache alpha
 CPU A: smp_rmb()
 CPU B: smp_wmb()
 CPU B: smp_rmb()
 CPU B: reads alpha from its cache
 CPU A: smp_wmb()
 CPU A: reads beta from its cache
```

At the bottom of the execution trail, the state for which the LTL condition did not hold is shown:

```
spin: trail ends after 161 steps
#processes: 1
                mem_alpha = 1
                cached_alpha[0] = 1
                cached_alpha[1] = 0
                cache_dirty_alpha[0] = 0
                cache_dirty_alpha[1] = 0
                mem_beta = 1
                cached_beta[0] = 1
                cached_beta[1] = 1
                cache_dirty_beta[0] = 0
                cache_dirty_beta[1] = 0
                x = 0
                y = 0
                pa_tokens = 31
                pb_tokens = 63
161:    proc  0 (CPU_A) line 218 "mem.spin" (state 239)
2 processes created
```

At that point, both pa_read and pb_read contain 0. By examining the execution trace, we understand that this behavior is made possible by letting processor A execute its read memory barrier before the write memory barrier. This is allowed because the removed core synchronization permits reordering these unrelated types of barriers.

Therefore, even given the known model limitations regarding loops and nesting, the instruction scheduling and weakly-ordered memory architecture models are sufficient to model RCU algorithms, as is shown in Section VI.

VI. Read-Copy Update Algorithm Modeling

Read-Copy Update (RCU) is a synchronization primitive allowing multiple readers of a data structure to execute concurrently with extremely low-overhead (1). Its main characteristic is to provide linear read-side scalability as the number of processor increases. It performs this by allowing multiple copies of a data structure to exist at the same time. In a period of time called *grace period*, each processor is allowed to see a different copy of the data structure. The RCU synchronization guarantees specify a lower bound to the duration of the grace period, after which no further references to old copies exist, so that the underlying memory becomes reclaimable.

The main motivation for validating the RCU algorithms is their complexity level. These algorithms are parallel and imply inconsistent views between processors at a specific point in time. Also, because RCU's read-side primitives contain no standard mutual exclusion primitives, memory ordering must be performed by these same RCU primitives.

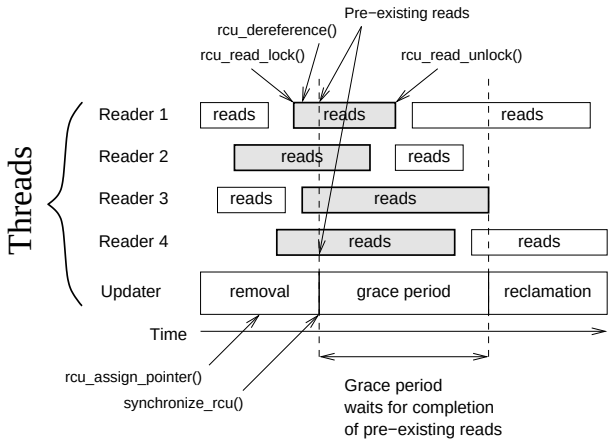

Fig. 7. Schematic of RCU Grace Period and Read-Side Critical Sections

This section presents the model we created to verify if the grace-period and publication guarantees are satisfied by two RCU synchronization algorithms proposed in paper (1): *General-Purpose RCU* and *Low-Overhead RCU Via Signal Handling*. It also verifies that both updater and readers always progress. We first describe the general RCU model, which is subsequently derived into a signal-based memory barrier model.

A schematic for the high-level structure of an RCU-based algorithm is shown in Figure 7. An RCU grace period is informally defined as any time period such that all RCU read-side critical sections in existence at the beginning of that period have completed before its end.

Here, each box labeled "Reads" is an RCU read-side critical section that begins with rcu_read_lock() and ends with rcu_read_unlock(). Each row of RCU read-side critical sections denotes a separate thread, for a total of four read-side threads. The two boxes at the bottom left and right of the figure denote a fifth thread, this one performing an RCU update.

This RCU update is split into two phases, a removal phase denoted by the lower left-hand box and a reclamation phase denoted by the lower right-hand box. These two phases must be separated by a grace period, which is determined by the duration of the synchronize_rcu() execution. During the removal phase, the RCU update removes elements from the data structure (possibly inserting some as well) by issuing an rcu_assign_pointer() or equivalent pointer-replacement primitive. These removed data elements will not be accessible to RCU read-side critical sections starting after the removal phase ends, but might still be accessed by RCU read-side critical sections initiated during the removal phase. However, by the end of the RCU grace period, all of the RCU read-side critical sections that might be accessing the newly removed data elements are guaranteed to have completed, courtesy of the definition of *grace period*. Therefore, the grace-period guarantee ensures that the reclamation phase, beginning after the grace period ends, can safely free the data elements removed previously.

The *publication guarantee* ensures that data accessed by

the read-side through the rcu_dereference() primitive (always executed between rcu_read_lock() and rcu_read_unlock()) will see changes made by the write-side before publication of the RCU pointer by rcu_assign_pointer.

The model which abstracts the RCU algorithm has one updater and one reader process. It is based on the *OoOmem* and *OoOisched* frameworks to verify that it satisfies the above-mentioned guarantees when executed on a weakly-ordered processor and memory architecture. In addition to global and per-thread RCU synchronization variables, the data structures required are a pointer to the current RCU data and an *arena*: a pool of free memory used by the memory allocator. All these data structures are modeled with the *OoOmem* framework.

The updater process performs two loops which first update data in a newly allocated arena entry from the *OoOmem* model and then publishes a pointer to the new entry into another shared *OoOmem* variable using a modeled rcu_assign_pointer() primitive. At the same time the updater stores the new pointer, the updater loads the previous value into its local registers. It is reclaimed after a grace period passes. The modeled synchronize_rcu() primitive is used to wait for such grace period to reach a quiescent state. After that point, the arena entry corresponding to the old pointer can be poisoned.

Memory reclamation is modeled by writing *poison* data into the arena entry. In a valid model, the read-side should never see such poison value in the memory location it reads. This method is used to detect inappropriate use of reclaimed memory.

The updater loops do not need to be unrolled because the synchronize_rcu() primitive contains full memory barriers. Poisoning alone could spill in the next loop and overlap with stores to the newly allocated arena entry, but late-arriving poisoning stores are not relevant to the characteristics we validate.

The reader is modeled as one process entering two read-side critical sections. Each consists of a lock and unlock pair, between which are placed a rcu_dereference() and a read of the corresponding arena entry. The first critical section holds two nested read locks. The second critical section holds a single nesting-level read lock. Given the outermost code of successive read lock/unlock can spill on each other, such spilling caused by reordering, prefetching and optimizations is modeled by those two successive critical sections. The data read within each critical section is saved to the globally visible data_read_first and data_read_second variables to be used for verification.

The guarantees provided by the RCU algorithm are verified with the LTL formula presented in (1), which makes sure the reader never loads a poisoned arena entry. The Spin model-checker checks for states which do not verify this claim. It generates an error and presents a counter-example consisting of a faulty execution trace when the claim is not satisfied.

$$\mathbf{G} \left(\begin{array}{c} data\_read\_first \neq POISON \\ \wedge \\ data\_read\_second \neq POISON \end{array} \right) \quad (1)$$

To minimize the risk of modeling errors, we augment our models with error-injection regression tests. For each of the characteristic we need to validate, we create a model alteration which is known not to satisfy the characteristic. This permits to not only verify that the guarantees are sufficient to ensure model correctness, but also that the modeled guarantees are actually required, e.g. if a predicate is not satisfied, an error will occur.

Remove `smp_mb()`**:** The first error injected is to remove the memory barriers surrounding synchronize_rcu(). This is known not to meet the grace period guarantee, as it would let the pointer update spill over the whole grace period into the following quiescent state. This would therefore let poisoning occur before the pointer is updated.

Remove `smp_wmb()`**:** The second error we inject is to remove the write memory-barrier from the rcu_assign_pointer() primitive. This is known not to meet the publication guarantee, because the pointer could then be published before the newly allocated arena entry is populated with non-poisoned information.

Remove `smp_rmb()`**:** The third error is injected by removing the read memory-barrier from the rcu_dereference() primitive. On Alpha (and *only* on Alpha), this is known not to meet the publication guarantee because the reader's cache could be populated with the new pointer before the arena entry is updated. We therefore expect the reader to see poisoned data.

Single grace-period phase: The fourth error-injection test consists in altering synchronize_rcu() to only perform a single grace-period phase. This is expected not to meet the grace period guarantee by allowing a race condition between a reader and two consecutive updates.

The reader code is modeled in an infinite loop to verify updater's progress when facing a steady flow of readers. The reader and updater progress are tested in two different runs:

- For reader progress, a single progress statement is added between each reader loop execution.
- On the updater-side, progress statements are added in each update loop and an infinite loop containing a progress statement is added at the end of the updater's process execution.

The weak fairness Spin option ensures that non-progress errors are flagged only for cycles containing at least one statement from each process, but not containing any special *progress* labels.

Remove `smp_mb()` **from busy-loop:** The fifth error injected is to remove the memory barrier placed in the updater's busy loop waiting for a grace period phase to complete. This injects an updater progress error by allowing the updater's cache to never read the eventually updated reader nesting counter. On real systems, the bounded size of the buffers between the CPU, cache and memory interconnects ensures that the remote nesting counter is updated, but given our virtual architecture model assumes infinitely-sized buffers, an explicit memory barrier must be placed in the busy-loops to ensure data is being read.

The signal-based memory barrier is modeled as a derivation of the general model by changing the updater-side memory barriers for a primitive which sends a signal and waits for memory barrier execution from the reader-side, all this between two memory barriers. On the read-side, the memory barriers are modeled by verifying, between each instruction execution, if the execution status tokens appears to be in

sequential execution order. If execution appears in sequential order between two instructions, the reader process chooses to either ignore any memory barrier request or to service any number of memory barrier requests by issuing memory barriers and informing the updater-side of the completion.

Due to the added complexity and therefore state-space size explosion, modeling of read-side in signal handers nesting over the updater and reader thread is performed using a different model which assumes a sequentially ordered architecture with the *OoOmem* weakly-ordered memory framework. Given that signal handlers have the property to order the core execution before and after they execute, it allows using the safety and progress characteristics proven with the *OoOisched* framework as lemma.

A. State-Space Compression

Given that the state-space required to perform the verification can increase quickly, the following state-space compression techniques were used to perform the verifications.

Running the Spin model-checker to verify specific `LTL` formulas transformed into *never claims* permits checking for safety while performing *Partial Order Reduction* (13). This model-checking approach discards relative statement ordering which does not matter for the property to verify. This reduces the state-space size tremendously with a very small performance impact, while preserving the safety and liveness properties of `LTL`.

Accepting a small performance impact (perceived slowdown of a factor 1.3 on our models), the `COLLAPSE` compression (11) can be used to reduce the state-space required to perform verification by separating the state into subcomponents. The compression comes from the fact that one state configuration for a specific process tends to reoccur in different global data and other process states. It uses separate descriptors as key to encode and search global data objects and data objects belonging to each process. Each time the same process state is encountered, it can be encoded with the same per-process state descriptor instead of saving the whole state, which saves precious state-space. A global state descriptor, used to identify the overall state, therefore consists of a state vector made of the global and per-process descriptors. This lossless compression preserves the complete state-space.

Another possible lossless compression technique, the `DMA` (Deterministic Finite Automaton) (11), can further diminish the state-space size by leveraging the high degree of similarity between the different states. It represents the state-space using an encoding similar to `BDD`s (Binary Decision Diagrams) (19; 20). However, this compression technique incurs a prohibitive performance impact. Our tests on large models show that state-space exploration is about 10 times slower. Given that the non-compressed execution of some verifications already takes about 24 hours, the computation time required for `DMA` compression is considered to be beyond our available computation time resources.

B. RCU Model-Checking Results

This section presents the Spin model-checker results for three models: the general purpose user-space `RCU` model, the signal-based `RCU` model and the modeling of signal-handler read-side. Test run results are presented along with the resources required to perform the verification. These verifications are performed on a Intel Core2 Xeon 2.0 GHz with 16 GB RAM.

The only test result we really care about is whether the verification succeeds or fails. For the unaltered model and for progress verifications, the `LTL` claim or progress property are expected to hold. Such successful verification are denoted as `PASS`. For each error-injection run presented in Section VI, the expected result is that the model-checker should detect the injected error, denoted as `INJECT`. The notation `FNEG` would indicate that the model-checker was blind to an injected error. This constitutes a "false negative". These are not errors as such, as not every bug necessarily results in a failure on every architectures modeled. Finally, the notation `FAIL` indicates that the model-checker detected a bug in the algorithm.

Additional information about the time and memory required to run these verifications is only provided to show the amount of computational resources needed for such verification. The only requirement is that execution time and memory used fit within our available resource limits.

Tables I and II present the result of the general-purpose `RCU` model-checking using the Alpha and Intel/PowerPC virtual architectures, respectively. The safety and progress verifications are successful, and all error-injections generate expected errors, except one: on the Intel/PowerPC architecture, no error is generated when removing the `smp_rmb()` on the read-side. This shows that no read barrier is required on these architectures due to the fact that dependent loads are not reordered.

TABLE I
GENERAL-PURPOSE `RCU` VERIFICATION RESULTS FOR THE ALPHA ARCHITECTURE

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	1.06	2h33m
Remove `smp_mb()`	INJECT	1.07	1h06m
Remove `smp_wmb()`	INJECT	0.96	1h14m
Remove `smp_rmb()`	INJECT	0.52	9m
Single grace-period phase	INJECT	0.66	26m
Reader progress	PASS	1.76	10h38m
Updater progress	PASS	1.76	9h23m
Remove loop `smp_mb()`	INJECT	0.47	1m

Table III presents the verification result of the signal-based `RCU` model for the Alpha virtual architecture. This verification ensures signal-based memory barriers provide the memory ordering guarantees and that no livelock nor deadlock can occur. Progress verification requires to use the `COLLAPSE` Spin option to compress the state-space size. It takes 3.5 days to complete the updater progress verification. To reduce the required CPU time, the reader progress and the updater progress error-injection are performed on a simplified read-side model with only a single, non-nested, critical section. Updater progress has also been verified using this simpler model and resulted in a successful progress verification.

TABLE II
GENERAL-PURPOSE RCU VERIFICATION RESULTS FOR THE INTEL/POWERPC ARCHITECTURES

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	0.82	4m
Remove `smp_mb()`	INJECT	1.96	16m
Remove `smp_wmb()`	INJECT	0.79	5m
Remove `smp_rmb()`	FNEG	0.82	6m
Single grace-period phase	INJECT	0.61	1m
Reader progress	PASS	1.28	23m
Updater progress	PASS	1.28	23m
Remove loop `smp_mb()`	INJECT	0.47	0m

TABLE III
SIGNAL-BASED RCU VERIFICATION RESULTS FOR THE ALPHA ARCHITECTURE

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	5.21	1d14h18m
Remove `smp_mb()`	INJECT	0.59	13m
Remove `smp_wmb()`	INJECT	5.17	1d14h33m
Remove `smp_rmb()`	INJECT	1.20	2h43m
Single grace-period phase	INJECT	3.09	5h45m
Reader progress	PASS	0.89	2h02m
Updater progress	PASS	11.73	4d15h13m
Remove loop `smp_mb()`	INJECT	0.472	1m

As in the Alpha signal-based RCU verification, Intel/PowerPC model verification require to use the COLLAPSE compression to fit in the available memory. Here we notice that the test execution time for each progress verification is approximately 4 hours and uses about 10 GB of memory. As in the general purpose RCU model, the `smp_rmb()` removal does not cause any error on Intel/PowerPC because the architecture does not reorder dependent loads.

TABLE IV
SIGNAL-BASED RCU VERIFICATION RESULTS FOR THE INTEL/POWERPC ARCHITECTURES

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	6.98	1h43m
Remove `smp_mb()`	INJECT	1.18	4m
Remove `smp_wmb()`	INJECT	6.98	1h42m
Remove `smp_rmb()`	FNEG	6.98	1h45m
Single grace-period phase	INJECT	4.28	16m
Reader progress	PASS	10.08	4h18m
Updater progress	PASS	9.88	4h18m
Remove loop `smp_mb()`	INJECT	0.58	3m

Modeling of read-side signal handler nested over a reader thread is presented in Table V. This model executes a read-side critical section in a signal handler interrupting a reader thread. We proceed to this verification to model a read-side critical section in a signal handler, which generates execution traces where a nested signal handler could deadlock with an interrupted process, if the signal handler would busy-loop waiting for the interrupted process.

TABLE V
GENERAL-PURPOSE RCU SIGNAL-HANDLER READER NESTED OVER READER VERIFICATION (NO INSTRUCTION SCHEDULING)

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	4.35	10m
Remove `smp_mb()`	INJECT	1.60	5m
Remove `smp_wmb()`	INJECT	0.78	2m
Remove `smp_rmb()`	INJECT	1.60	2m
Single grace-period phase	INJECT	0.57	0m
Reader progress	PASS	9.21	1h56m
Updater progress	PASS	9.15	1h03m
Remove loop `smp_mb()`	INJECT	0.51	0m

Table VI is the results obtained by modeling an interrupting read-side signal handler critical section nested over the updater thread. It presents an interesting result: given all read-side critical sections are contained within signal handlers nested over the updater, no memory barrier is required to ensure correctness because no cache-line exchange is required. In fact, only a single process is executing.

TABLE VI
GENERAL-PURPOSE RCU SIGNAL-HANDLER READER NESTED OVER UPDATER VERIFICATION (NO INSTRUCTION SCHEDULING)

Model Regression Test	PASS/FAIL INJECT/FNEG	Memory (GB)	Time
Unaltered model (safety)	PASS	0.47	0m
Remove `smp_mb()`	FNEG	0.48	1m
Remove `smp_wmb()`	FNEG	0.47	1m
Remove `smp_rmb()`	FNEG	0.47	0m
Single grace-period phase	FNEG	0.47	0m
Reader progress	PASS	0.47	1m
Updater progress	PASS	0.47	1m
Remove loop `smp_mb()`	FNEG	0.47	1m

For each of the unaltered models checked, model coverage includes all of the RCU model lines, but excludes some *OoOmem* model operations which are not useful in some contexts. For instance, the *OoOmem* "random" store to memory will never be executed if a process never writes into a given variable. Error injection runs do not need to visit all the state space because they stop after the first error encountered. Therefore, these self-testing runs do not need to provide complete coverage.

C. RCU Verification Discussion

Results presented in Section VI-B demonstrate that we were able to successfully verify the RCU algorithm models in various execution scenarios with affordable computation resources. The error-injection tests further demonstrate that the model is able to detect defects that do not respect the RCU guarantees.

In these tests, the number of updater has been limited to one given we protect updater critical sections using a mutual

exclusion primitive already expected to be valid. The number of reader is also fixed to one because the updater waits, in turn, for each reader one after the other. The algorithm therefore does not contain any reader-reader data or control dependency.

As expected, model of the read-side signal handler nested over a RCU reader succeeds because the RCU read-side is executed with $O(1)$ computational complexity, which implies that it never busy-loops.

The simplified read-side in signal handler model does not perform instruction execution reordering. Given the proof provided by the previous verifications, the nested signal handler execution can be modeled as being serialized with the rest of the interrupted code because the operating system is called before and after the signal handler. The *OoOmem* model is however still used to appropriately take the out-of-order memory effects into account. This models the Alpha virtual architecture, which is a superset of the Intel/PowerPC virtual architecture, given it allows weaker memory ordering.

Verification of interrupting read-side signal handler critical section nested over the updater thread interestingly shows that the read-side signal handler can nest over the updater without causing progress error (no livelock nor deadlock). The single grace-period phase test shows no error. This can be explained by the fact that the execution trace which requires two grace-period phases involves the reader seeing two updater updates. This execution trace is impossible here because the updater is being interrupted by the nested read-side signal.

Error-injection tests have been very useful to ensure model completeness. For instance, trying the test-case presented in Section V-B on the *OoOmem* model showed its limitations. Changing the `smp_mb()` into consecutive `smp_rmb()` and `smp_wmb()` (which are free to be reordered) did not produce the expected error. This showed that we needed to model out-of-order instruction scheduling to properly represent this class of CPU instruction reordering effects, effectively leading to the creation of the *OoOisched* model.

Another example where error-injection has been useful happened during the *OoOisched*-based RCU model creation. The *OoOisched* framework being based on a instruction scheduling loop, we can only use the model coverage information provided by Spin as indication that statements have been reached at least once, but it tells nothing about the execution orders visited. Instruction dependency implementation errors, which inhibited execution of some instructions incorrectly, were identified with the help of these error-injection tests.

We also created a model for uniprocessor execution of the RCU algorithm. The code generated for this model has the particularity that all memory barriers are replaced by compiler barriers, except `smp_read_barrier_depends()`, on Alpha, which is completely removed. In this model, a single processor cache is used by both the reader and the writer processes. No communication is required with main memory, given all accesses are going through the locally cached variables. Therefore, out-of-order memory updates are disabled. The results of the tests, not presented here for conciseness, show that simply using compiler barriers suffice to provide RCU safe against thread preemption on a uniprocessor system.

VII. Framework Discussion

Compared to models used previously for RCU verification, the proposed framework covers more micro-architecture side-effects. This includes, for instance, effects of data prefetch. Moreover, the state-space size required by our framework has been shown to be manageable on current computers when modeling complex synchronization algorithms such as RCU. This shows that it should be applicable to other parallel algorithms with similar complexity level.

One of the major improvements of this modeling framework is to allow a more regular description of algorithms. It removes the need to account for low-level architecture side-effects directly in the algorithm model by providing artefacts which encapsulate the architecture behavior. This framework therefore minimizes the risk of modeling error.

Due to its ability to model the weakest ordering possible, altering the framework to model memory barriers specific to architectures such as Alpha, Intel, PowerPC and even Sparc is straightforward. Modeling specific architectures can by done by creating the synchronization instructions implemented in a given architecture and modifying the behavior of the cache-memory synchronization to match the architecture behavior. For instance, the PowerPC "`lwsync`" instruction[2] can be modeled as two instructions. The first instruction needed is a `smp_rmb()` which depend on all prior loads, and upon which depends all following loads and stores. The second instruction is a `smp_wmb()`, which depends on all prior stores, but upon which only the following stores depend. Such flexibility in modeling the low-level synchronization primitives become very handy to model the Sparc "`membar`" primitive, which permits to only order either, some or all of:

- stores vs stores,
- stores vs loads,
- loads vs loads or
- loads vs stores.

In the case of the RCU algorithm model, we only need full memory barriers.

We are aware of one recently proposed compiler optimization not handled by our model. Value-speculative optimizations (21; 22) performed by the compiler could cause dependent loads to be performed out-of-order if the first data to read is speculated, which would permit to read dependent data in the wrong order. These dependency-breaking optimizations are outside of the proposed model scope. Work in progress for upcoming versions of the C++ standard include compiler mechanisms designed to selectively suppress value speculation (23; 24; 25).

VIII. Conclusion

To accurately model the low-level multiprocessor interactions at the architecture-level, we created a virtual architecture performing the most aggressive optimizations still meeting the instruction inter-dependencies. Memory access ordering is expressed by modeling a processor cache with extremely weak

[2] `lwsync` - Lightweight synchronization: Orders loads with respect to subsequent loads and stores. Orders stores with respect to other stores. Does not order stores with respect to subsequent loads.

ordering. A model of instruction dependencies deals with the effects of out-of-order instruction execution.

Formal verification of both general-purpose RCU and signal-based RCU has been performed on this virtual architecture, therefore modeling the effects of out-of-order instruction execution and out-of-order memory accesses. The high complexity-level of these RCU algorithms caused by the high degree of parallelism and extremely relaxed consistency semantics can easily overwhelm human conception. This is why validation at the lowest level of interprocessor interaction is needed to certify that these algorithms perform the expected synchronization.

Future work in this area could involve modeling value-speculative compiler optimizations, to enable detection of ordering problems which can occur when dependent memory accesses can be reordered dependency-breaking compiler optimizations.

Modeling these algorithms on this virtual architecture lets us demonstrate that all invocations of this algorithm primitives will behave appropriately and that porting it to yet unforeseen architectures will work as expected.

ACKNOWLEDGEMENTS

We owe thanks to Nicolas Gorse, Etienne Bergeron and Alexandre Desnoyers for reviewing this paper, to Maged Michael and Alan Stern for many illuminating discussions, and to Kathy Bennett for her support of this effort.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851. This work is funded by Google, Natural Sciences and Engineering Research Council of Canada, Ericsson and Defence Research and Development Canada.

LEGAL STATEMENT

REFERENCES

[1] M. Desnoyers, P. E. McKenney, A. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *to appear*.

[2] P. E. McKenney, "Using Promela and Spin to verify parallel algorithms," August 2007, available: http://lwn.net/Articles/243851/.

[3] P. E. McKenney and S. Rostedt, "Integrating and validating dynticks and preemptable RCU," April 2008.

[4] *A Better x86 Memory Model: x86-TSO*, 2009.

[5] *The semantics of power and ARM multiprocessor machine code*. New York, NY, USA: ACM, 2008.

[6] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[7] D. A. Schmidt, "Data flow analysis is model checking of abstract interpretations," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, California, 1998, pp. 38–48.

[8] B. Steffen, "Data flow analysis as model checking," in *Lectures Notes in Computer Sciences. Theoretical Aspects of Computer Software: TACS*, vol. 256, 1991, pp. 346–364.

[9] M. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," in *In Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, 1994, pp. 62–75.

[10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. Springer-Verlag, 2001.

[11] G. J. Holzmann, "The SPIN model checker," *IEEE Transactions on Software Engineering*, 1997.

[12] J. van Leeuwen, Ed., *Handbook of theoretical computer science (vol. B): formal models and semantics*. Cambridge, MA, USA: MIT Press, 1990.

[13] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8(1), pp. 39–64, 1996.

[14] P. E. McKenney, "Memory ordering in modern microprocessors, part II," *Linux Journal*, July 2005.

[15] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19(2), pp. 24–36, March 1999.

[16] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9(3), pp. 223–252, 1977.

[17] K. Jensen, "Coloured petri nets," *Petri Nets: Central Models and Their Properties*, vol. 254, pp. 248–299, 1987.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[19] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35(8), pp. 677–691, 1986.

[20] ——, "Symbolic boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[21] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value speculation scheduling for high performance processors," *SIGPLAN Not.*, vol. 33, no. 11, pp. 262–271, 1998.

[22] C. ying Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Software-only value speculation scheduling," Tech. Rep., 1998.

[23] P. E. McKenney, "C++ data-dependency ordering: Atomics," http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2359.html, 2007.

[24] ——, "C++ data-dependency ordering: Memory model," http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2360.html, 2007.

[25] ——, "C++ data-dependency ordering: Function annotation," http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2361.html, 2007.