Stateless Model Checking for POWER

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson(✉)

Department of Information Technology, Uppsala University, Uppsala, Sweden
carl.leonardsson@it.uu.se

Abstract. We present the first framework for efficient application of stateless model checking (SMC) to programs running under the relaxed memory model of POWER. The framework combines several contributions. The first contribution is that we develop a scheme for systematically deriving operational execution models from existing axiomatic ones. The scheme is such that the derived execution models are well suited for efficient SMC. We apply our scheme to the axiomatic model of POWER from [8]. Our main contribution is a technique for efficient SMC, called *Relaxed Stateless Model Checking* (RSMC), which systematically explores the possible inequivalent executions of a program. RSMC is suitable for execution models obtained using our scheme. We prove that RSMC is sound and optimal for the POWER memory model, in the sense that each complete program behavior is explored exactly once. We show the feasibility of our technique by providing an implementation for programs written in C/pthreads.

1 Introduction

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which parallel threads can interact. To make matters worse, current shared-memory multicore processors, such as Intel's x86, IBM's POWER, and ARM, [9,28,29,45], achieve higher performance by implementing *relaxed memory models* that allow threads to interact in even subtler ways than by interleaving of their instructions, as would be the case in the model of *sequential consistency* (SC) [32]. Under the relaxed memory model of POWER, loads and stores to different memory locations may be reordered by the hardware, and the accesses may even be observed in different orders on different processor cores.

Stateless model checking (SMC) [25] is one successful technique for verifying concurrent programs. It detects violations of correctness by systematically exploring the set of possible program executions. Given a concurrent program which is terminating and threadwisely deterministic (e.g., by fixing any input data to avoid data-nondeterminism), a special runtime scheduler drives the SMC exploration by controlling decisions that may affect subsequent computations, so that the exploration covers all possible executions. The technique is automatic, has no false positives, can be applied directly to the program source code, and

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 134–156, 2016.
DOI: 10.1007/978-3-319-41540-6_8

can easily reproduce detected bugs. SMC has been successfully implemented in tools, such as VeriSoft [26], Chess [37], Concuerror [17], rInspect [49], and Nidhugg [1].

However, SMC suffers from the state-space explosion problem, and must therefore be equipped with techniques to reduce the number of explored executions. The most prominent one is *partial order reduction* [18,24,39,47], adapted to SMC as *dynamic partial order reduction* (DPOR) [2,23,40,43]. DPOR addresses state-space explosion caused by the many possible ways to schedule concurrent threads. DPOR retains full behavior coverage, while reducing the number of explored executions by exploiting that two schedules which induce the same order between conflicting instructions will induce equivalent executions. DPOR has been adapted to the memory models TSO and PSO [1,49], by introducing auxiliary threads that induce the reorderings allowed by TSO and PSO, and using DPOR to counteract the resulting increase in thread schedulings.

In spite of impressive progress in SMC techniques for SC, TSO, and PSO, there is so far no effective technique for SMC under more relaxed models, such as POWER. A major reason is that POWER allows more aggressive reorderings of instructions within each thread, as well as looser synchronization between threads, making it significantly more complex than SC, TSO, and PSO. Therefore, existing SMC techniques for SC, TSO, and PSO can not be easily extended to POWER.

In this paper, we present the first SMC algorithm for programs running under the POWER relaxed memory model. The technique is both sound, in the sense that it guarantees to explore each programmer-observable behavior at least once, and optimal, in the sense that it does not explore the same complete behavior twice. Our technique combines solutions to several major challenges.

The first challenge is to design an execution model for POWER that is suitable for SMC. Existing execution models fall into two categories. Operational models, such as [12,21,41,42], define behaviors as resulting from sequences of small steps of an abstract processor. Basing SMC on such a model would induce large numbers of executions with equivalent programmer-observable behavior, and it would be difficult to prevent redundant exploration, even if DPOR techniques are employed. Axiomatic models, such as [7,8,36], avoid such redundancy by being defined in terms of an abstract representation of programmer-observable behavior, due to Shasha and Snir [44], here called *Shasha-Snir traces*. However, being axiomatic, they judge whether an execution is allowed only after it has been completed. Directly basing SMC on such a model would lead to much wasted exploration of unallowed executions. To address this challenge, we have therefore developed a scheme for systematically deriving execution models that are suitable for SMC. Our scheme derives an execution model, in the form of a labeled transition system, from an existing axiomatic model, defined in terms of Shasha-Snir traces. Its states are partially constructed Shasha-Snir traces. Each transition adds ("commits") an instruction to the state, and also equips the instruction with a parameter that determines how it is inserted into the Shasha-Snir trace. The parameter of a load is the store from which it reads its value. The parameter of a store is its position in the coherence order of stores to the same memory location. The order in which instructions are added must respect various dependencies between instructions, such that each instruction makes sense at the time when it is added. For example, when adding a store or a load instruction, earlier instructions that are needed to compute which memory address it accesses must already have been added. Our execution model therefore takes as input a partial order, called *commit-before*, which constrains the order in which instructions can be added. The commit-before order should be tuned to suit the given axiomatic memory model. We define a condition of *validity* for commit-before orders, under which our derived execution model is equivalent to the original axiomatic one, in that they generate the same sets of Shasha-Snir traces. We use our scheme to derive an execution model for POWER, equivalent to the axiomatic model of [8].

Having designed a suitable execution model, we address our main challenge, which is to design an effective SMC algorithm that explores all Shasha-Snir traces that can be generated by the execution model. We address this challenge by a novel exploration technique, called *Relaxed Stateless Model Checking* (RSMC). RSMC is suitable for execution models, in which each instruction can be executed in many ways with different effects on the program state, such as those derived using our execution model scheme. The exploration by RSMC combines two mechanisms: (i) RSMC considers instructions one-by-one, respecting the commit-before order, and explores the effects of each possible way in which the instruction can be executed. (ii) RSMC monitors the generated execution for data races from loads to subsequent stores, and initiates alternative explorations where instructions are reordered. We define the property *deadlock freedom* of execution models, meaning intuitively that no run will block before being complete. We prove that RSMC is sound for deadlock free execution models, and that our execution model for POWER is indeed deadlock free. We also prove that RSMC is optimal for POWER, in the sense that it explores each *complete* Shasha-Snir trace exactly once. Similar to sleep set blocking for classical SMC/DPOR, it may happen for RSMC that superfluous *incomplete* Shasha-Snir traces are explored. Our experiments indicate, however, that this is rare.

To demonstrate the usefulness of our framework, we have implemented RSMC in the stateless model checker Nidhugg [33]. For test cases written in C with pthreads, it explores all Shasha-Snir traces allowed under the POWER memory model, up to some bounded length. We evaluate our implementation on several challenging benchmarks. The results show that RSMC efficiently explores the Shasha-Snir traces of a program, since (i) on most benchmarks, our implementation performs no superfluous exploration (as discussed above), and (ii) the running times correlate to the number of Shasha-Snir traces of the program. We show the competitiveness of our implementation by comparing with an existing state of the art analysis tool for POWER: goto-instrument [5].

Outline. The next section presents our derivation of execution models. Section 3 presents our RSMC algorithm, and Sect. 4 presents our implementation and experiments. Proofs of all theorems, and formal definitions, are provided in our technical report [4]. Our implementation is available at [33].

2 Execution Model for Relaxed Memory Models

POWER — A Brief Glimpse. The programmer-observable behavior of POWER multiprocessors emerges from a combination of many features, including out-of-order and speculative execution, various buffers, and caches. POWER provides significantly weaker ordering guarantees than, e.g., SC and TSO.

We consider programs consisting of a number of threads, each of which runs a deterministic code, built as a sequence of assembly instructions. The grammar of our assumed language is given in Fig. 1. The threads access a shared memory, which is a mapping from addresses to values. A program may start by declaring named global variables with specific initial values. Instructions include register assignments and conditional branches with the usual semantics. A load 'r:=[a]' loads the value from the memory address given by the arithmetic expression a into the register r. A store '[a_0]:=a_1' stores the value of the expression a_1 to the memory location addressed by the evaluation of a_0. For a global variable x, we use x as syntactic sugar for [&x], where &x is the address of x. The instructions sync, lwsync, isync are fences (or memory barriers), which are special instructions preventing some memory ordering relaxations. Each instruction is given a label, which is assumed to be unique.

As an example, consider the program in Fig. 2. It consists of two threads P and Q, and has two zero-initialized memory locations x and y. The thread P loads the value of x, and stores that value plus one to y. The thread Q is similar, but always stores the value 1, regardless of the loaded value. Under the SC or TSO memory models, at least one of the loads L0 and L2 is guaranteed to load the initial value 0 from memory. However, under POWER the order between the load L2 and the store L3 is not maintained. Then it is possible for P to load the value 1 into r_0, and for Q to load 2 into r_1. Inserting a sync between L2 and L3 would prevent such a behavior.

Axiomatic Memory Models. Axiomatic memory models, of the form in [8], operate on an abstract representation of observable program behavior, introduced by Shasha and Snir [44], here called *traces*. A trace is a directed graph,

```
   ⟨prog⟩ ::= ⟨varinit⟩* ⟨thrd⟩+
⟨varinit⟩ ::= ⟨var⟩ '=' ℤ
   ⟨thrd⟩ :=  'thread' ⟨tid⟩ ':' ⟨linstr⟩+
 ⟨linstr⟩ ::= ⟨label⟩ ':' ⟨instr⟩ ';'
  ⟨instr⟩ ::= ⟨reg⟩ ':=' ⟨expr⟩ |            // register assignment
              'if' ⟨expr⟩ 'goto' ⟨label⟩ | // conditional branch
              ⟨reg⟩ ':=' '[' ⟨expr⟩ ']' |            // memory load
              '[' ⟨expr⟩ ']' ':=' ⟨expr⟩ |          // memory store
              'sync' | 'lwsync' | 'isync'                 // fences
   ⟨expr⟩ ::= (arithmetic expression over literals and registers)
```

Fig. 1. The grammar of concurrent programs

```
x = 0         y = 0
thread P:     thread Q:
L0: r0 := x;  L2: r1 := y;
L1: y := r0+1; L3: x := 1;
```

$$\begin{tikzcd}
\texttt{L0: } r_0 := x \arrow[d, "\text{po,data}"'] & & \texttt{L2: } r_1 := y \arrow[d, "\text{po}"] \\
\texttt{L1: } y := r_0\text{+1} \arrow[urr, bend right, "\text{rf}"'] & & \texttt{L3: } x := 1 \arrow[ull, bend left, "\text{rf}"]
\end{tikzcd}$$

Fig. 2. Left: An example program: LB + data. Right: A trace of the same program.

Event	Parameter	Semantic Meaning
L3: x := 1	0	First in coherence order for x
L0: r_0 := x	L3	Read value 1 from L3
L1: y := r_0+1	0	First in coherence order for y
L2: r_1 := y	L1	Read value 2 from L1

Fig. 3. The run L3[0].L0[L3].L1[0].L2[L1], of the program in Fig. 2 (left), leading to the complete state corresponding to the trace given in Fig. 2 (right). Here we use the labels L0–L3 as shorthands for the corresponding events.

in which vertices are executed instructions (called *events*), and edges capture dependencies between them. More precisely, a *trace* π is a quadruple $(E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ where E is a set of *events*, and po, co, and rf are relations over E[1]. An *event* is a tuple (t, n, l) where t is an identifier for the executing thread, l is the unique label of the instruction, and n is a natural number which disambiguates instructions. Let $\mathbb{E}$ denote the set of all possible events. For an event $e = (\mathsf{t}, n, l)$, let $\mathsf{tid}(e)$ denote t and let $\mathsf{instr}(e)$ denote the instruction labelled l in the program code. The relation po (for "program order") totally orders all events executed by the same thread. The relation co (for "coherence order") totally orders all stores to the same memory location. The relation rf (for "read-from") contains the pairs (e, e') such that e is a store and e' is a load which gets its value from e. For simplicity, we assume that the initial value of each memory address x is assigned by a special *initializer instruction* init_x, which is first in the coherence order for that address. A trace is a *complete trace of the program* $\mathcal{P}$ if the program order over the committed events of each thread makes up a path from the first instruction in the code of the thread, to the last instruction, respecting the evaluation of conditional branches. Figure 2 shows the complete trace corresponding to the behavior described in the beginning of this section, in which each thread loads the value stored by the other thread.

An axiomatic memory model M (following the framework [8]) is defined as a predicate M over traces π, such that $\mathrm{M}(\pi)$ holds precisely when π is an allowed trace under the model. Deciding whether $\mathrm{M}(\pi)$ holds involves checking (i) that the trace is internally consistent, defined in the natural way (e.g., the relation co relates precisely events that access the same memory location), and (ii) that various combinations of relations that are derived from the trace are acyclic or irreflexive. Which specific relations need to be acyclic depends on the memory model.

1 [8] uses the term "execution" to denote what we call "trace".

We define the *axiomatic semantics under* M as a mapping from programs $\mathcal{P}$ to their denotations $[\![\mathcal{P}]\!]^{\mathsf{Ax}}_{\mathrm{M}}$, where $[\![\mathcal{P}]\!]^{\mathsf{Ax}}_{\mathrm{M}}$ is the set of complete traces π of $\mathcal{P}$ such that $\mathrm{M}(\pi)$ holds. In the following, we assume that the axiomatic memory model for POWER, here denoted M^{POWER}, is defined as in [8]. The interested reader is encouraged to read the details in [8], but the high-level understanding given above should be enough to understand the remainder of this text.

Deriving an Execution Model. Let an axiomatic model M be given, in the style of [8]. We will derive an equivalent execution model in the form of a transition system.

States. States of our execution model are traces, augmented with a set of fetched events. A state σ is a tuple of the form $(\lambda, F, E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ where $\lambda(\mathsf{t})$ is a label in the code of t for each thread t, $F \subseteq \mathbb{E}$ is a set of events, and $(E, \mathsf{po}|_E, \mathsf{co}, \mathsf{rf})$ is a trace such that $E \subseteq F$. (Here $\mathsf{po}|_E$ is the restriction of po to E.) For a state $\sigma = (\lambda, F, E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, we let $\mathsf{exec}(\sigma)$ denote the trace $(E, \mathsf{po}|_E, \mathsf{co}, \mathsf{rf})$. Intuitively, F is the set of all currently fetched events and E is the set of events that have been committed. The function λ gives the label of the next instruction to fetch for each thread. The relation po is the program order between all fetched events. The relations co and rf are defined for committed events (i.e., events in E) only. The set of all possible states is denoted $\mathbb{S}$. The initial state $\sigma_0 \in \mathbb{S}$ is defined as $\sigma_0 = (\lambda_0, E_0, E_0, \varnothing, \varnothing, \varnothing)$ where λ_0 is the function providing the initial label of each thread, and E_0 is the set of all initializer events.

Commit-Before. The order in which events can be committed – effectively a linearization of the trace – is restricted by a *commit-before order*. It is a parameter of our execution model which can be tuned to suit the given axiomatic model. Formally, a commit-before order is defined by a *commit-before function* cb, which associates with each state $\sigma = (\lambda, F, E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, a *commit-before order* $\mathsf{cb}_\sigma \subseteq F \times F$, which is a partial order on the set of fetched events. For each state σ, the commit-before order cb_σ induces a predicate $\mathit{enabled}_\sigma$ over the set of fetched events $e \in F$ such that $\mathit{enabled}_\sigma(e)$ holds if and only if $e \notin E$ and the set $\{e' \in F \,|\, (e', e) \in \mathsf{cb}_\sigma\}$ is included in E. Intuitively, e can be committed only if all the events it depends on have already been committed. Later in this section, we define requirements on commit-before functions, which are necessary for the execution model and for the RSMC algorithm respectively.

Transitions. The transition relation between states is given by a set of rules, in Fig. 4. The function $\mathrm{val}_\sigma(e, a)$ denotes the value taken by the arithmetic expression a, when evaluated at the event e in the state σ. The value is computed in the natural way, respecting data-flow. (Formal definition in the technical report [4].) For example, in the state σ corresponding to the trace given in Fig. 2, where e is the event corresponding to label L1, we would have $\mathrm{val}_\sigma(e, \mathsf{r}_0\mathsf{+1}) = 2$. The function $\mathrm{address}_\sigma(e)$ associates with each load or store event e the memory location accessed. For a label l, let $\lambda_{\mathsf{next}}(l)$ denote the next label following l in the program code. Finally, for a state σ with coherence order co and a store e to some memory location x, we let $\mathit{extend}_\sigma(e)$ denote the set of coherence orders co' which result from inserting e anywhere in the total order of stores to x in co.

$$\frac{\begin{array}{c} F_{\mathsf{t}} = \{e'' \in F | \mathsf{tid}(e'') = \mathsf{t}\} \quad e = (\mathsf{t}, |F_{\mathsf{t}}|, \lambda(\mathsf{t})) \\ \nexists e', a, l \;.\; e' \in F \setminus E \wedge \mathsf{tid}(e') = \mathsf{t} \wedge \mathsf{instr}(e') = (\texttt{if}\ a\ \texttt{goto}\ l) \end{array}}{\sigma \xrightarrow{FLB} (\lambda[\mathsf{t} \hookleftarrow \lambda_{\mathsf{next}}(\lambda(\mathsf{t}))], F \cup \{e\}, E, \mathsf{po} \cup (F_{\mathsf{t}} \times \{e\}), \mathsf{co}, \mathsf{rf})}\text{FETCH}$$

$$\frac{\begin{array}{c} \mathsf{instr}(e) = (\texttt{if}\ a\ \texttt{goto}\ l) \quad \mathsf{t} = \mathsf{tid}(e) \\ \mathrm{val}_{\sigma}(e, a) \in \mathbb{Z} \setminus \{0\} \quad \mathit{enabled}_{\sigma}(e) \end{array}}{\sigma \xrightarrow{FLB} (\lambda[\mathsf{t} \hookleftarrow l], F, E \cup \{e\}, \mathsf{po}, \mathsf{co}, \mathsf{rf})}\text{BRT} \qquad \frac{\begin{array}{c} \mathsf{instr}(e) \in \{\mathsf{sync}, \mathsf{lwsync}, \mathsf{isync}, \mathsf{r}\texttt{:=}a\} \\ \mathit{enabled}_{\sigma}(e) \end{array}}{\sigma \xrightarrow{FLB} (\lambda, F, E \cup \{e\}, \mathsf{po}, \mathsf{co}, \mathsf{rf})}\text{LOC}$$

$$\frac{\begin{array}{c} \mathsf{instr}(e) = (\texttt{if}\ a\ \texttt{goto}\ l) \\ \mathrm{val}_{\sigma}(e, a) = 0 \quad \mathit{enabled}_{\sigma}(e) \end{array}}{\sigma \xrightarrow{FLB} (\lambda, F, E \cup \{e\}, \mathsf{po}, \mathsf{co}, \mathsf{rf})}\text{BRF} \qquad \frac{\begin{array}{c} \mathsf{instr}(e) = (\texttt{[}a\texttt{]:=}a') \quad \mathit{enabled}_{\sigma}(e) \quad \mathrm{M}(\mathsf{exec}(\sigma')) \\ \sigma' = (\lambda, F, E \cup \{e\}, \mathsf{po}, \mathsf{co}', \mathsf{rf}) \quad \mathsf{co}' \in \mathit{extend}_{\sigma}(e) \end{array}}{\sigma \xrightarrow{e[\mathit{position}_{\mathsf{co}'}(e)]} \sigma'}\text{ST}$$

$$\frac{\begin{array}{c} \mathsf{instr}(e) = (\mathsf{r}\texttt{:=[}a\texttt{]}) \quad \mathit{enabled}_{\sigma}(e) \quad e_w \in E \quad \mathsf{instr}(e_w) = (\texttt{[}a'\texttt{]:=}a'') \\ \mathrm{address}_{\sigma}(e_w) = \mathrm{address}_{\sigma}(e) \quad \sigma' = (\lambda, F, E \cup \{e\}, \mathsf{po}, \mathsf{co}, \mathsf{rf} \cup \{(e_w, e)\}) \quad \mathrm{M}(\mathsf{exec}(\sigma')) \end{array}}{\sigma \xrightarrow{e[e_w]} \sigma'}\text{LD}$$

Fig. 4. Execution model of programs under the memory model M. Here $\sigma = (\lambda, F, E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$.

For each such order co', we let $\mathit{position}_{\mathsf{co}'}(e)$ denote the position of e in the total order: I.e. $\mathit{position}_{\mathsf{co}'}(e)$ is the number of (non-initializer) events e' which precede e in co'.

The intuition behind the rules in Fig. 4 is that events are committed non-deterministically out of order, but respecting the constraints induced by the commit-before order. When a memory access (load or store) is committed, a non-deterministic choice is made about its effect. If the event is a store, it is non-deterministically inserted somewhere in the coherence order. If the event is a load, we non-deterministically pick the store from which to read. Thus, when committed, each memory access event e is parameterized by a choice p: the coherence position for a store, and the source store for a load. We call $e[p]$ a *parameterized event*, and let $\mathbb{P}$ denote the set of all possible parameterized events. A transition committing a memory access is only enabled if the resulting state is allowed by the memory model M. Transitions are labelled with FLB when an event is fetched or a local event is committed, or with $e[p]$ when a memory access event e is committed with parameter p.

We illustrate this intuition for the program in Fig. 2 (left). The trace in Fig. 2 (right) can be produced by committing the instructions (events) in the order `L3`, `L0`, `L1`, `L2`. For the load `L0`, we can then choose the already performed `L3` as the store from which it reads, and for the load `L2`, we can choose to read from the store `L1`. Each of the two stores `L3` and `L1` can only be inserted at one place in their respective coherence orders, since the program has only one store to each memory location. We show the resulting sequence of committed events in Fig. 3:

the first column shows the sequence of events in the order they are committed, the second column is the parameter assigned to the event, and the third column explains the parameter. Note that other traces can be obtained by choosing different values of parameters. For instance, the load `L2` can also read from the initial value, which would generate a different trace.

Next we explain each of the rules: The rule FETCH allows to fetch the next instruction according to the control flow of the program code. The first two requirements identify the next instruction. To fetch an event, all preceding branch events must already be committed. Therefore events are never fetched along a control flow path that is not taken. We point out that this restriction does not prevent our execution model from capturing the observable effects of speculative execution (formally ensured by Theorem 1).

The rules LOC, BRT and BRF describe how to commit non-memory access events.

When a store event is committed by the ST rule, it is inserted non-deterministically at some position $n = \mathit{position}_{\mathsf{co}'}(e)$ in the coherence order. The guard $\mathrm{M}(\mathsf{exec}(\sigma'))$ ensures that the resulting state is allowed by the axiomatic memory model.

The rule LD describes how to commit a load event e. It is similar to the ST rule. For a load we non-deterministically choose a source store e_w, from which the value can be read. As before, the guard $\mathrm{M}(\mathsf{exec}(\sigma'))$ ensures that the resulting state is allowed.

Given two states $\sigma, \sigma' \in \mathbb{S}$, we use $\sigma \xrightarrow{FLB(max)} \sigma'$ to denote that $\sigma \xrightarrow{FLB}{}^{*} \sigma'$ and there is no state $\sigma'' \in \mathbb{S}$ with $\sigma' \xrightarrow{FLB} \sigma''$. A *run* τ from some state σ is a sequence of parameterized events $e_1[p_1].e_2[p_2].\cdots.e_k[p_k]$ such that $\sigma \xrightarrow{FLB(max)} \sigma_1 \xrightarrow{e_1[p_1]} \sigma_1' \xrightarrow{FLB(max)} \cdots \xrightarrow{e_k[p_k]} \sigma_k' \xrightarrow{FLB(max)} \sigma_{k+1}$ for some states $\sigma_1, \sigma_1', \ldots, \sigma_k', \sigma_{k+1} \in \mathbb{S}$. We write $e[p] \in \tau$ to denote that the parameterized event $e[p]$ appears in τ. Observe that the sequence τ leads to a uniquely determined state σ_{k+1}, which we denote $\tau(\sigma)$. A run τ, from the initial state σ_0, is *complete* iff the reached trace $\mathsf{exec}\tau(\sigma_0)$ is complete. Figure 3 shows an example complete run of the program in Fig. 2 (left).

In summary, our execution model represents a program $\mathcal{P}$ as a labeled transition system $TS^{\mathcal{P}}_{\mathrm{M},\mathsf{cb}} = (\mathbb{S}, \sigma_0, \longrightarrow)$, where $\mathbb{S}$ is the set of states, σ_0 is the initial state, and $\longrightarrow \subseteq \mathbb{S} \times (\mathbb{P} \cup \{FLB\}) \times \mathbb{S}$ is the transition relation. We define the *execution semantics under* M *and* cb as a mapping, which maps each program $\mathcal{P}$ to its denotation $[\![\mathcal{P}]\!]^{\mathsf{Ex}}_{\mathrm{M},\mathsf{cb}}$, which is the set of complete runs τ induced by $TS^{\mathcal{P}}_{\mathrm{M},\mathsf{cb}}$.

Validity and Deadlock Freedom. Here, we define validity and deadlock freedom for memory models and commit-before functions. Validity is necessary for the correct operation of our execution model (Theorem 1). Deadlock freedom is necessary for soundness of the RSMC algorithm (Theorem 4). First, we introduce some auxiliary notions.

We say that a state $\sigma' = (\lambda', F', E', \mathsf{po}', \mathsf{co}', \mathsf{rf}')$ is a *cb-extension* of a state $\sigma = (\lambda, F, E, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, denoted $\sigma \leq_{\mathsf{cb}} \sigma'$,if σ' can be obtained from σ by fetching in program order or committing events in cb order. Formally $\sigma \leq_{\mathsf{cb}} \sigma'$ if $\mathsf{po} = \mathsf{po}'|_F$, $\mathsf{co} = \mathsf{co}'|_E$, $\mathsf{rf} = \mathsf{rf}'|_E$, F is a po'-closed subset of F', and E is a $\mathsf{cb}_{\sigma'}$-closed subset of E'. More precisely, the condition on F means that for any events $e, e' \in F'$, we have $[e' \in F \wedge (e, e') \in \mathsf{po}'] \Rightarrow e \in F$. The condition on E is analogous.

We say that cb is *monotonic* w.r.t. M if whenever $\sigma \leq_{\mathsf{cb}} \sigma'$, then (i) $\mathrm{M}(\mathsf{exec}(\sigma')) \Rightarrow \mathrm{M}(\mathsf{exec}(\sigma))$, (ii) $\mathsf{cb}_\sigma \subseteq \mathsf{cb}_{\sigma'}$, and (iii) for all $e \in F$ such that either $e \in E$ or $\left(enabled_\sigma(e) \wedge e \notin E'\right)$, we have $(e', e) \in \mathsf{cb}_\sigma \Leftrightarrow (e', e) \in \mathsf{cb}_{\sigma'}$ for all $e' \in F'$. Conditions (i) and (ii) are natural monotonicity requirements on M and cb. Condition (iii) says that while an event is committed or enabled, its cb-predecessors do not change.

A state σ induces a number of relations over its fetched (possibly committed) events. Following [8], we let addr_σ, data_σ, ctrl_σ, denote respectively address dependency, data dependency and control dependency. Similarly, $\mathsf{po\text{-}loc}_\sigma$ is the subset of po that relates memory accesses to the same memory location. Lastly, sync_σ and lwsync_σ relate events that are separated in program order by respectively a sync or lwsync. The formal definitions can be found in [8], and in our technical report [4]. We can now define a weakest reasonable commit-before function cb^0, capturing natural dependencies:

$$\mathsf{cb}^0_\sigma = (\mathsf{addr}_\sigma \cup \mathsf{data}_\sigma \cup \mathsf{ctrl}_\sigma \cup \mathsf{rf})^+,$$

where R^+ denotes the transitive (but not reflexive) closure of R.

We say that a commit-before function cb is *valid* w.r.t. a memory model M if cb is monotonic w.r.t. M, and for all states σ such that $\mathrm{M}(\mathsf{exec}(\sigma))$ we have that cb_σ is acyclic and $\mathsf{cb}^0_\sigma \subseteq \mathsf{cb}_\sigma$.

Theorem 1 (Equivalence with Axiomatic Model). Let cb be a commit-before function valid w.r.t. a memory model M. Then $[\![\mathcal{P}]\!]^{\mathsf{Ax}}_{\mathrm{M}} = \{\mathsf{exec}(\tau(\sigma_0)) \mid \tau \in [\![\mathcal{P}]\!]^{\mathsf{Ex}}_{\mathrm{M},\mathsf{cb}}\}$. $\square$

Program		Blocked run τ
`x = 0`	`y = 0`	L3[0]
`thread P:`	`thread Q:`	L5[0]
`L0: r0:=y;`	`L3: x:=3;`	L0[L5]
`L1: x:=r0;`	`L4: sync;`	L2[0]
`L2: x:=2;`	`L5: y:=1;`	`(L1 blocked)`

Blocked state σ

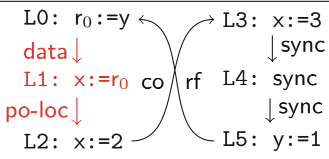


Fig. 5. If the weak commit-before function cb^0 is used, the POWER semantics may deadlock. When the program above (left) is executed according to the run τ (center) we reach a state σ (right) where L0, L2, L3–L5 are successfully committed. However, any attempt to commit L1 will close a cycle in the relation $\mathsf{co}; \mathsf{sync}_\sigma; \mathsf{rf}; \mathsf{data}_\sigma; \mathsf{po\text{-}loc}_\sigma$, which is forbidden under POWER. This blocking behavior is prevented when the stronger commit-before function $\mathsf{cb}^{\mathsf{power}}$ is used, since it requires L1 and L2 to be committed in program order.

The commit-before function cb^0 is valid w.r.t. M^{POWER}, implying (by Theorem 1) that $[\![\mathcal{P}]\!]^{\mathsf{Ex}}_{\mathrm{M}^{POWER},\mathsf{cb}^0}$ is a faithful execution model for POWER. However, cb^0 is not strong enough to prevent blocking runs in the execution model for POWER. I.e., it is possible, with cb^0, to create an incomplete run, which cannot be completed. Any such blocking is undesirable for SMC, since it corresponds to wasted exploration. Figure 5 shows an example of how the POWER semantics may deadlock when based on cb^0.

We say that a memory model M and a commit before function cb are *deadlock free* if for all runs τ from σ_0 and memory access events e such that $\mathit{enabled}_{\tau(\sigma_0)}(e)$ there exists a parameter p such that $\tau.e[p]$ is a run from σ_0. I.e., it is impossible to reach a state where some event is enabled, but has no parameter with which it can be committed.

Commit-Before Order for POWER. We will now define a stronger commit before function for POWER, which is both valid and deadlock free:

$$\mathsf{cb}^{\mathsf{power}}_{\sigma} = (\mathsf{cb}^0_{\sigma} \cup (\mathsf{addr}_{\sigma};\mathsf{po}) \cup \mathsf{po\text{-}loc}_{\sigma} \cup \mathsf{sync}_{\sigma} \cup \mathsf{lwsync}_{\sigma})^{+}$$

Theorem 2. $\mathsf{cb}^{\mathsf{power}}$ is valid w.r.t. M^{POWER}.

Theorem 3. M^{POWER} and $\mathsf{cb}^{\mathsf{power}}$ are deadlock free.

3 The RSMC Algorithm

Having derived an execution model, we address the challenge of defining an SMC algorithm, which explores all allowed traces of a program in an efficient manner. Since each trace can be generated by many equivalent runs, we must, just as in standard SMC for SC, develop techniques for reducing the number of explored runs, while still guaranteeing coverage of all traces. Our RSMC algorithm is designed to do this in the context of semantics like the one defined above, in which instructions can be committed with several different parameters, each yielding different results.

Our exploration technique basically combines two mechanisms:

(i) In each state, RSMC considers an instruction e, whose cb-predecessors have already been committed. For each possible parameter value p of e in the current state, RSMC extends the state by $e[p]$ and continues the exploration recursively.

(ii) RSMC monitors generated runs to detect read-write conflicts (or "races"), i.e., the occurrence of a load and a subsequent store to the same memory location, such that the load would be able to read from the store if they were committed in the reverse order. For each such conflict, RSMC starts an alternative exploration, in which the load is preceded by the store, so that the load can read from the store.

Mechanism (ii) is analogous to the detection and reversal of races in conventional DPOR, with the difference that RSMC need only detect conflicts in which a load is followed by a store. A race where a load follows a store does not induce reordering by mechanism (ii). This is because our execution model allows the load to read from any of the already committed stores to the same memory location, without any reordering.

Instruction	Parameter	Semantic Meaning
`L0: r0 := x`	init_x	(read initial value)
`L1: y := r0+1`	0	(first in coherence of y)
`L2: r1 := y`	init_y	(read initial value)
`L3: x := 1`	0	(first in coherence of x)

Fig. 6. The first explored run of the program in Fig. 2

We illustrate the basic idea of RSMC on the program in Fig. 2 (left). As usual in SMC, we start by running the program under an arbitrary schedule, subject to the constraints imposed by the commit-before order `cb`. For each instruction, we explore the effects of each parameter value which is allowed by the memory model. Let us assume that we initially explore the instructions in the order `L0`, `L1`, `L2`, `L3`. For this schedule, there is only one possible parameter for `L0`, `L1`, and `L3`, whereas `L2` can read either from the initial value or from `L1`. Let us assume that it reads the initial value. This gives us the first run, shown in Fig. 6. The second run is produced by changing the parameter for `L2`, and let it read the value 1 written by `L1`.

During the exploration of the first two runs, the RSMC algorithm also detects a race between the load `L0` and the store `L3`. An important observation is that `L3` is not ordered after `L0` by the commit-before order, implying that their order can be reversed. Reversing the order between `L0` and `L3` would allow `L0` to read from `L3`. Therefore, RSMC initiates an exploration where the load `L0` is preceded by `L3` and reads from it. (If `L3` would have been preceded by other events that enable `L3`, these would be executed before `L3`.) After the sequence `L3[0].L0[L3]`, RSMC is free to choose the order in which the remaining instructions are considered. Assume that the order `L1`, `L2` is chosen. In this case, the load `L2` can read from either the initial value or from `L1`. In the latter case, we obtain the run in Fig. 3, corresponding to the trace in Fig. 2 (right).

After this, there are no more unexplored parameter choices, and so the RSMC algorithm terminates, having explored four runs corresponding to the four possible traces.

In the following section, we will provide a more detailed look at the RSMC algorithm, and see formally how this exploration is carried out.

3.1 Algorithm Description

In this section, we present our algorithm, RSMC, for SMC under POWER. We prove soundness of RSMC, and optimality w.r.t. explored *complete* traces.

The RSMC algorithm is shown in Fig. 7. It uses the recursive procedure **Explore**, which takes parameters τ and σ such that $\sigma = \tau(\sigma_0)$. **Explore** will explore all states that can be reached by complete runs extending τ.

First, on line 1, we fetch instructions and commit all local instructions as far as possible from σ. The order of these operations makes no difference. Then we turn to memory accesses. If the run is not yet terminated, we select an enabled event e on line 2.

```
// P[$e$] holds a run
// preceding the load event $e$.
global P = $\lambda e.\langle\rangle$
// Q[$e$] holds a set of continuations
// leading to the execution of the
// load event $e$ after P[$e$].
global Q = $\lambda e.\varnothing$

Explore($\tau, \sigma$)
     // Fetch & commit local greedily.
 1: while($\exists\sigma'.\sigma \xrightarrow{FLB} \sigma'$){$\sigma$ := $\sigma'$;}
     // Find committable memory access $e$.
 2: if($\exists e.enabled_\sigma(e)$){
 3:   if($e$ is a store){
        // Explore all ways to execute $e$.
 4:     S := $\{(n,\sigma')|\sigma \xrightarrow{e[n]} \sigma'\}$;
 5:     for($(n,\sigma') \in$ S){
 6:       Explore($\tau.e[n], \sigma'$);
 7:     }
 8:     DetectRace($\tau, \sigma, e$);
 9:   }else{ // $e$ is a load
10:     P[$e$] := $\tau$;
        // Explore all ways to execute $e$.
11:     S := $\{(e_w,\sigma')|\sigma \xrightarrow{e[e_w]} \sigma'\}$;
12:     for($(e_w,\sigma') \in$ S){
13:       Explore($\tau.e[e_w], \sigma'$);
14:     }
        // Handle R -> W races.
15:     explored = $\varnothing$;
16:     while($\exists\tau' \in$ Q[$e$]\explored){
17:       explored := explored$\cup\{\tau'\}$;
18:       Traverse($\tau, \sigma, \tau'$);
19:     }
20:   }
21: }
```

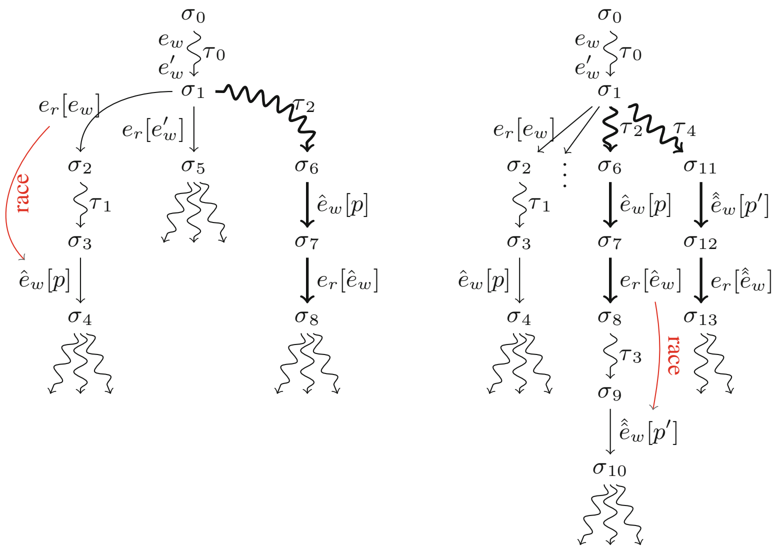
```
DetectRace($\tau, \sigma, e$)
 1: for( $e_r[e_w] \in \tau$ s.t.
          $e_r$ is a load $\wedge (e_r, e) \notin \mathsf{cb}_\sigma$
          $\wedge\ \mathsf{address}_\sigma(e_r) = \mathsf{address}_\sigma(e)$ ){
      // Compute postfix after P[$e_r$].
 2:   $\tau'$ := the $\tau'$ s.t. $\tau =$ P[$e_r$].$\tau'$;
      // Remove events not cb-before $e$.
 3:   $\tau''$ := $\mathsf{normalize}(\mathsf{cut}(\tau', e, \sigma), \mathsf{cb}_\sigma)$;
      // Construct new continuation.
 4:   $\tau'''$ := $\tau''.e[*].e_r[e]$;
      // Add to Q, to explore later.
 5:   Q[$e_r$] := Q[$e_r$]$\cup\{\tau'''\}$;
 6: }
```

```
Traverse($\tau, \sigma, \tau'$)
 1: if($\tau' = \langle\rangle$){
 2:   Explore($\tau, \sigma$);
 3: }else{
      // Fetch & commit local greedily.
 4:   while($\exists\sigma'.\sigma \xrightarrow{FLB} \sigma'$){$\sigma$ := $\sigma'$;}
 5:   $e[p].\tau''$ := $\tau'$; // Get first event.
 6:   if($p = *$){
        // Explore all ways to execute $e$.
 7:     S := $\{(n,\sigma')|\sigma \xrightarrow{e[n]} \sigma'\}$;
 8:     for($(n,\sigma') \in$ S){
 9:       Traverse($\tau.e[n], \sigma', \tau''$);
10:     }
11:   }else if($\exists\sigma'.\sigma \xrightarrow{e[p]} \sigma'$){
12:     Traverse($\tau.e[p], \sigma', \tau''$);
13:   }else{
        // Only happens when the final
        // load in $\tau'$ does not accept its
        // parameter. Stop exploring.
14:   }
15: }
```

Fig. 7. An algorithm to explore all traces of a given program. The initial call is **Explore**($\langle\rangle, \sigma_0$).

If the chosen event e is a store (lines 3–8), we first collect, on line 4, all parameters for e which are allowed by the memory model. For each of them, we recursively explore all of its continuations on line 6. I.e., for each coherence position n that is allowed for e by the memory model, we explore the continuation of τ obtained by committing $e[n]$. Finally, we call **DetectRace**. We will return shortly to a discourse of that mechanism.

If e is a load (lines 9–20), we proceed in a similar manner. Line 10 is related to **DetectRace**, and discussed later. On line 11 we compute all allowed parameters for the load e. They are (some of the) stores in τ which access the same address as e. On line 13, we make one recursive call to **Explore** per allowed parameter. The structure of this exploration is illustrated in the two branches from σ_1 to σ_2 and σ_5 in Fig. 8(a).



(a) A new branch $\tau_2.\hat{e}_w[*].e_r[\hat{e}_w]$ is added to $\mathtt{Q}[e_r]$ and later explored, starting from σ_1. τ_2 is a restriction of τ_1, containing only events that are cb_{σ_4}-before $\hat{e}_w$.

(b) Another read-write race is detected, starting from the leaf of a branch explored by **Traverse**. The new branch $\tau_4.\hat{\hat{e}}_w[*].e_r[\hat{\hat{e}}_w]$ is added at σ_1, not at σ_7.

Fig. 8. How **Explore** applies event parameters, and introduces new branches. Thin arrows indicate exploration performed directly by **Explore**. Bold arrows indicate traversal by **Traverse**.

Notice in the above that both for stores and loads, the available parameters are determined entirely by τ, i.e. by the events that precede e in the run. In the case of stores, the parameters are coherence positions between the earlier stores occurring in τ. In the case of loads, the parameters are the earlier stores occurring in τ. For stores, this way of exploring is sufficient. But for loads it is necessary to also consider parameters which appear later than the load in a run. Consider the example in Fig. 8(a). During the recursive exploration of a run from σ_0 to σ_4 we encounter a new store $\hat{e}_w$, which is in a race with e_r. If the load e_r and the store $\hat{e}_w$ access the same memory location, and e_r does not precede $\hat{e}_w$ in the cb-order, they could appear in the opposite order in a run (with $\hat{e}_w$ preceding e_r), and $\hat{e}_w$ could be an allowed parameter for the load e_r.

This read-write race is detected on line 1 in the function **DetectRace**, when it is called from line 8 in **Explore** when the store $\hat{e}_w$ is being explored. We must then ensure that some run is explored where $\hat{e}_w$ is committed before e_r so that $\hat{e}_w$ can be considered as a parameter for e_r. Such a run must include all events that are before $\hat{e}_w$ in cb-order, so that $\hat{e}_w$ can be committed. We construct τ_2, which is a template for a new run, including precisely the events in τ_1 which are cb-before the store $\hat{e}_w$. The run template τ_2 can be explored from the state σ_1 (the state where e_r was previously committed) and will then lead to a state where $\hat{e}_w$ can be committed. The run template τ_2 is computed from the complete run in **DetectRace** on lines 2 and 3. This is done by first removing (at line 2) the prefix τ_0 which precedes e_r (stored in $\mathtt{P[}e_r\mathtt{]}$ on line 10 in **Explore**). Thereafter (at line 3) events that are not cb-before $\hat{e}_w$ are removed using the function cut (here, $\mathsf{cut}(\tau, e, \sigma)$ restricts τ to the events which are cb_σ-before e), and the resulting run is normalized. The function $\mathsf{normalize}$ normalizes a run by imposing a predefined order on the events which are not ordered by cb. This is done to avoid unnecessarily exploring two equivalent run templates. The run template $\tau_2.\hat{e}_w[*].e_r[\hat{e}_w]$ is then stored on line 5 in the set $\mathtt{Q[}e_r\mathtt{]}$, to ensure that it is explored later. Here we use the special pseudo-parameter $*$ to indicate that every allowed parameter for $\hat{e}_w$ should be explored (See lines 6–10 in **Traverse**).

All of the run templates collected in $\mathtt{Q[}e_r\mathtt{]}$ are explored from the same call to **Explore**(τ_0, σ_1) where e_r was originally committed. This is done on lines 15–19. The new branch is shown in Fig. 8(a) in the run from σ_0 to σ_8. Notice on line 18 that the new branch is explored by the function **Traverse**, rather than by **Explore** itself. This has the effect that τ_2 is traversed, with each event using the parameter given in τ_2, until $e_r[\hat{e}_w]$ is committed. The traversal by **Traverse** is marked with bold arrows in Fig. 8. If the memory model does not allow e_r to be committed with the parameter $\hat{e}_w$, then the exploration of this branch terminates on line 13 in **Traverse**. Otherwise, the exploration continues using **Explore**, as soon as e_r has been committed (line 2 in **Traverse**).

Let us now consider the situation in Fig. 8(b) in the run from σ_0 to σ_{10}. Here $\tau_2.\hat{e}_w[*].e_r[\hat{e}_w]$, is explored as described above. Then **Explore** continues the exploration, and a read-write race is discovered from e_r to $\hat{\hat{e}}_w$. From earlier DPOR algorithms such as e.g. [23], one might expect that this case is handled by exploring a new branch of the form $\tau_2.\hat{e}_w[p].\tau_3'.\hat{\hat{e}}_w[p'].e_r[\hat{\hat{e}}_w]$, where e_r is simply delayed after σ_7 until $\hat{\hat{e}}_w$ has been committed. Our algorithm handles the case differently, as shown in the run from σ_0 to σ_{13}. Notice that $\mathtt{P[}e_r\mathtt{]}$ can be used to identify the position in the run where e_r was last committed by **Explore** (as opposed to by **Traverse**), i.e., σ_1 in Fig. 8(b). We start the new branch from that position (σ_1), rather than from the position where e_r was committed when the race was detected (i.e., σ_7). The new branch τ_4 is constructed when the race is detected on lines 2 and 3 in **DetectRace**, by restricting the sub-run $\tau_2.\hat{e}_w[p].e_r[\hat{e}_w].\tau_3$ to events that cb-precede the store $\hat{\hat{e}}_w$.

The reason for returning all the way up to σ_1, rather than starting the new branch at σ_7, is to avoid exploring multiple runs corresponding to the same trace. This could otherwise happen when the same race is detected in multiple runs. To see this happen, let us consider the program given in Fig. 9. A part of its exploration tree is given in Fig. 10. In the interest of brevity, when describing the exploration of the program runs, we will ignore some runs which would be explored by the algorithm, but which have no impact on the point of the example. Throughout this example, we will use the labels `L0`, `L1`, and `L2` to identify the events corresponding to the labelled instructions. We assume that in the first run to be explored (the path from σ_0 to σ_3 in Fig. 10), the load at `L0` is committed first (loading the initial value of `x`), then the stores at `L1` and `L2`. There are two read-write races in this run, from `L0` to `L1` and to `L2`. When the races are detected, the branches `L1[*].L0[L1]` and `L2[*].L0[L2]` will be added to `Q[L0]`. These branches are later explored, and appear in Fig. 10 as the paths from σ_0 to σ_6 and from σ_0 to σ_9 respectively. In the run ending in σ_9, we discover the race from `L0` to `L1` again. This indicates that a run should be explored where `L0` reads from `L1`. If we were to continue exploration from σ_7 by delaying `L0` until `L1` has been committed, we would follow the path from σ_7 to σ_{11} in Fig. 10. In σ_{11}, we have successfully reversed the race between `L0` and `L1`. However, the trace of σ_{11} turns out to be identical to the one we already explored in σ_6. Hence, by exploring in this manner, we would end up exploring redundant runs. The **Explore** algorithm avoids this redundancy by exploring in the different manner described above: When the race from `L0` to `L1` is discovered at σ_9, we consider the entire sub-run `L2[0].L0[L2].L1[1]` from σ_0, and construct the new sub-run `L1[*].L0[L1]` by removing all events that are not cb-before `L1`, generalizing the parameter to `L1`, and by appending `L0[L1]` to the result. The new branch `L1[*].L2[L1]` is added to `Q[L0]`. But `Q[L0]` already contains the branch `L1[*].L2[L1]` which was added at the beginning of the exploration. And since it has already been explored (it has already been added to the set `explored` at line 17) we avoid exploring it again.

```
thread P:     thread Q:     thread R:
L0: r := x    L1: x := 1    L2: x := 2
```

Fig. 9. A small program where one thread P loads from x, and two threads Q and R store to x.

Soundness and Optimality. We first establish soundness of the RSMC algorithm in Fig. 7 for the POWER memory model, in the sense that it guarantees to explore all Shasha-Snir traces of a program. We thereafter establish that RSMC is optimal, in the sense that it will never explore the same complete trace twice.

Theorem 4 (Soundness). Assume that cb is valid w.r.t. M, and that M and cb are deadlock free. Then, for each $\pi \in [\![\mathcal{P}]\!]^{\mathsf{Ax}}_{\mathrm{M}}$, the evaluation of a call to **Explore**$(\langle\rangle, \sigma_0)$ will contain a recursive call to **Explore**(τ, σ) for some τ, σ such that $\mathsf{exec}(\sigma) = \pi$. □

Corollary 1. RSMC is sound for POWER using M^{POWER} and $\mathsf{cb}^{\mathsf{power}}$.

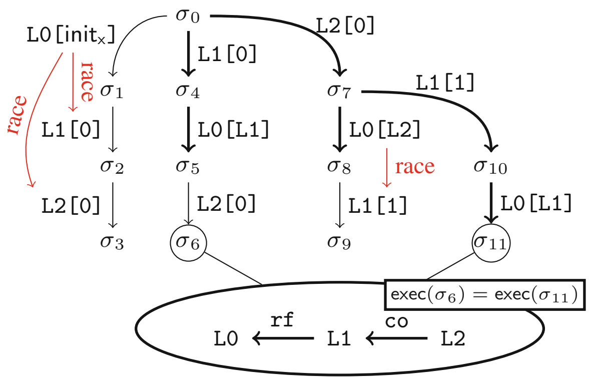


Fig. 10. Part of a faulty exploration tree for the program above, containing redundant branches. The branches ending in σ_6 and σ_{11} correspond to the same trace. The RSMC algorithm avoids this redundancy by the mechanism where all branches for read-write races from the same load e_r are collected in one set `Q`[e_r].

The proof of Theorem 4 involves showing that if an allowed trace exists, then the races detected in previously explored runs are sufficient to trigger the later exploration of a run corresponding to that trace.

Theorem 5 (Optimality for POWER). Assume that $\mathrm{M} = \mathrm{M}^{POWER}$ and $\mathsf{cb} = \mathsf{cb}^{\mathsf{power}}$. Let $\pi \in [\![\mathcal{P}]\!]^{\mathsf{Ax}}_{\mathrm{M}}$. Then during the evaluation of a call to **Explore**$(\langle\rangle, \sigma_0)$, there will be exactly one call **Explore**(τ, σ) such that $\mathsf{exec}(\sigma) = \pi$. $\square$

While the RSMC algorithm is optimal in the sense that it explores precisely one complete run per Shasha-Snir trace, it may initiate explorations that block before reaching a complete trace (similarly to sleep set blocking in classical DPOR). Such blocking may arise when the RSMC algorithm detects a read-write race and adds a branch to `Q`, which upon traversal turns out to be not allowed under the memory model. Our experiments in Sect. 4 indicate that the effect of such blocking is almost negligible, without any blocking in most benchmarks, and otherwise at most 10 % of explored runs.

4 Experimental Results

In order to evaluate the efficiency of our approach, we have implemented it as a part of the open source tool Nidhugg [33], for stateless model checking of C/pthreads programs under the relaxed memory. It operates under the restrictions that (i) all executions are bounded by loop unrolling, and (ii) the analysis runs on a given compilation of the target C code. The implementation uses RSMC to explore all allowed program behaviors under POWER, and detects any assertion violation that can occur. We validated our implementation by successfully running all 8070 relevant litmus tests published with [8].

Table 1. A comparison of running times (in seconds) for our implementation Nidhugg and goto-instrument. The *F* column indicates whether fences have been inserted code to regain safety. The *LB* column indicates whether the tools were instructed to unroll loops up to a certain bound. A *t/o* entry means that the tool failed to complete within 900 s. An asterisk (*) means that the tool found a safety violation. A struck out entry means that the tool gave the wrong answer regarding the safety of the benchmark. The superior running time for each benchmark is given in bold font. The *SS* column indicates the number of complete traces explored by Nidhugg before detecting an error, exploring all traces, or timing out. The *B* (for "blocking") column indicates the number of incomplete runs that Nidhugg started to explore, but that turned out to be invalid.

Tool running time (s), and trace count						
	goto-instrument			Nidhugg		
	F	LB	Time	Time	SS	B
dcl_singleton		7	*0.40	***0.13**	3	0
dcl_singleton	y	7	5.05	**0.19**	7	0
dekker		10	*229.39	***0.11**	5	0
dekker	y	10	t/o	**0.76**	246	0
fib_false			***1.86**	t/o	109171	0
fib_false_join			***0.84**	*35.46	11938	0
fib_true			**7.05**	t/o	109122	0
fib_true_join			**8.92**	57.67	19404	0
indexer		5	68.16	**1.57**	19	0
lamport		8	*635.45	***0.12**	3	0
lamport	y	8	t/o	**0.20**	50	2
parker		5	~~1.20~~	***0.13**	5	0
parker	y	5	**1.24**	7.44	1126	0
peterson			*0.24	***0.11**	3	0
peterson	y		0.19	**0.11**	10	1
pgsql		8	*161.05	***0.11**	2	0
pgsql	y	8	t/o	**0.58**	16	0
pgsql_bnd			t/o	***0.11**	2	0
pgsql_bnd	y		t/o	t/o	36211	0
stack_safe			**13.84**	73.86	1005	0
stack_unsafe			***1.03**	*3.32	20	0
szymanski			*1.02	***0.11**	17	0
szymanski	y		304.87	**0.31**	226	0

The main goals of our experimental evaluation are (i) to show the feasibility and competitiveness of our approach, in particular to show for which programs it performs well, (ii) to compare with goto-instrument, which to our knowledge is the only other tool analyzing C/pthreads programs under POWER[2], and (iii) to show the effectiveness of our approach in terms of wasted exploration effort.

Table 1 shows running times for Nidhugg and goto-instrument for several benchmarks in C/pthreads. All benchmarks were run on an 3.07 GHz Intel Core i7 CPU with 6 GB RAM. We use goto-instrument version 5.1 with cbmc version 5.1 as backend.

We note here that the comparison of running time is mainly relevant for the benchmarks where *no* error is detected (errors are indicated with a * in Table 1). This is because when an error is detected, a tool may terminate its analysis without searching the remaining part of the search space (i.e., the remaining runs in our case). Therefore the time consumption in such cases, is determined by whether the search strategy was lucky or not. This also explains why in e.g. the dekker benchmark, fewer Shasha-Snir traces are explored in the version *without* fences, than in the version *with* fences.

Comparison with goto-instrument. goto-instrument employs code-to-code transformation in order to allow verification tools for SC to work for more relaxed memory models such as TSO, PSO and POWER [5]. The results in Table 1 show that our technique is competitive. In many cases Nidhugg significantly outperforms goto-instrument. The benchmarks for which goto-instrument performs better than Nidhugg, have in common that goto-instrument reports that no trace may contain a cycle which indicates non-SC behavior. This allows goto-instrument to avoid expensive program instrumentation to capture the extra program behaviors caused by memory consistency relaxation. While this treatment is very beneficial in some cases (e.g. for stack_* which is data race free and hence has no non-SC executions), it also leads to false negatives in cases like parker, when goto-instrument fails to detect Shasha Snir-cycles that cause safety violations. In contrast, our technique is precise, and will never miss any behaviors caused by the memory consistency violation within the execution length bound.

We remark that our approach is restricted to thread-wisely deterministic programs with fixed input data, whereas the bounded model-checking used as a backend (CBMC) for goto-instrument can handle both concurrency and data nondeterminism.

Efficiency of Our Approach. While our RSMC algorithm explores precisely one complete run per Shasha-Snir trace, it may additionally start to explore runs that then turn out to block before completing, as described in Sect. 3. The SS and B columns of Table 1 indicate that the effect of such blocking is almost negligible, with no blocking in most benchmarks, and at most 10 % of the runs.

A costly aspect of our approach is that every time a new event is committed in a trace, Nidhugg will check which of its possible parameters are allowed by the axiomatic memory model. This check is implemented as a search for particular

[2] The cbmc tool previously supported POWER [6], but has withdrawn support in later versions.

cycles in a graph over the committed events. The cost is alleviated by the fact that RSMC is optimal, and avoids exploring unnecessary traces.

To illustrate this tradeoff, we present the small program in Fig. 11. The first three lines of each thread implement the classical Dekker idiom. It is impossible for both threads to read the value 0 in the same execution. This property is used to implement a critical section, containing the lines `L4`–`L13` and `M4`–`M13`. However, if the fences at `L1` and `M1` are removed, the mutual exclusion property can be violated, and the critical sections may execute in an interleaved manner. The program *with* fences has only three allowed Shasha-Snir traces, corresponding to the different observable orderings of the first three instructions of both threads. *Without* the fences, the number rises to 184759, due to the many possible interleavings of the repeated stores to z. The running time of Nidhugg is 0.01 s with fences and 161.36 s without fences.

```
x = 0     y = 0     z = 0

thread P:          thread Q:
L0:  x := 1;       M0:  y := 1;
L1:  sync;         M1:  sync;
L2:  r0 := y;      M2:  r1 := x;
L3:  if r0 = 1     M3:  if r1 = 1
       goto L14;          goto M14;
L4:  z := 1;       M4:  z := 1;
L5:  z := 1;       M5:  z := 1;
L6:  z := 1;       M6:  z := 1;
L7:  z := 1;       M7:  z := 1;
L8:  z := 1;       M8:  z := 1;
L9:  z := 1;       M9:  z := 1;
L10: z := 1;       M10: z := 1;
L11: z := 1;       M11: z := 1;
L12: z := 1;       M12: z := 1;
L13: z := 1;       M13: z := 1;
L14: r0 := 0;      M14: r1 := 0;
```

Fig. 11. SB+10W+syncs: A litmus test based on the idiom known as "Dekker" or "SB". It has 3 allowed Shasha-Snir traces under POWER. If the sync fences at lines `L1` and `M1` are removed, then it has 184759 allowed Shasha-Snir traces. This test is designed to have a large difference between the *total* number of coherent Shasha-Snir traces and the number of *allowed* Shasha-Snir traces.

We compare this with the results of the litmus test checking tool herd [8], which operates by generating all possible Shasha-Snir traces, and then checking which are allowed by the memory model. The running time of herd on SB+10W+syncs is 925.95 s with fences and 78.09 s without fences. Thus herd performs better than Nidhugg on the litmus test without fences. This is because a large proportion of the possible Shasha-Snir traces are allowed by the memory model. For each of them herd needs to check the trace only once. On the other hand, when the fences are added, the performance of herd deteriorates. This is because herd still checks every Shasha-Snir trace against the memory model, and each check becomes more expensive, since the fences introduce many new dependency edges into the traces.

We conclude that our approach is particularly superior for application style programs with control structures, mutual exclusion primitives etc., where relaxed memory effects are significant, but where most potential Shasha-Snir traces are forbidden.

5 Conclusions

We present the first framework for efficient SMC for programs running under POWER. It combines solutions to several challenges. We developed a scheme for systematically deriving execution models that are suitable for SMC, from axiomatic ones. We present RSMC, a novel algorithm for exploring all relaxed-memory traces of a program, based on our derived execution model. We show that RSMC is sound for POWER, meaning that it explores all Shasha-Snir traces of a program, and optimal in the sense that it explores the same complete trace exactly once. RSMC can in some situations waste effort by exploring blocked runs, but our experimental results shows that this is rare in practice. Our implementation shows that the RSMC approach is competitive relative to an existing state-of-the-art implementation. We expect that RSMC will be sound also for other similar memory models with suitably defined commit-before functions.

Related Work. Several SMC techniques have been recently developed for programs running under the memory models TSO and PSO [1,20,49]. In this work we propose a novel and efficient SMC technique for programs running under POWER.

In [8], a similar execution model was suggested, also based on the axiomatic semantics. However, compared to our semantics, it will lead many spurious executions that will be blocked by the semantics as they are found to be disallowed. This would cause superfluous runs to be explored, if used as a basis for stateless model checking.

Beyond SMC techniques for relaxed memory models, there have been many works related to the verification of programs running under relaxed memory models (e.g., [3,11,13–15,19,30,31,35,48]). Some of these works propose precise analysis techniques for finite-state programs under relaxed memory models (e.g., [3,11,21]). Others propose algorithms and tools for monitoring and testing programs running under relaxed memory models (e.g., [14–16,22,35]). Different techniques based on explicit state-space exploration for the verification of programs running under relaxed memory models have also been developed during the last years (e.g., [27,30,31,34,38]). There are also a number of efforts to design bounded model checking techniques for programs under relaxed memory models (e.g., [6,13,46,48]) which encode the verification problem in SAT/SMT. Finally, there are code-to-code transformation techniques (e.g., [5,10,11]) which reduce verification of a program under relaxed memory models to verification of a transformed program under SC. Most of these works do not handle POWER. In [21], the robustness problem for POWER has been shown to be PSPACE-complete.

The closest works to ours were presented in [5,6,8]. The work [5] extends cbmc to work with relaxed memory models (such as TSO, PSO and POWER) using a code-to-code transformation. The work in [6] develops a bounded model checking technique that can be applied to different memory models (e.g., TSO, PSO, and POWER). The cbmc tool previously supported POWER [6], but has withdrawn support in its later versions. The tool herd [8] operates by generating all possible Shasha-Snir traces, and then for each one of them checking whether it is allowed by the memory model. In Sect. 4, we experimentally compare RSMC with the tools of [5,8].

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015)
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384. ACM (2014)
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER (to appear)
5. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
7. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
9. ARM: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2014)
10. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
11. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
12. Boudol, G., Petri, G., Serpette, B.P.: Relaxed operational semantics of concurrent programming languages. In: EXPRESS/SOS 2012. EPTCS, vol. 89, pp. 19–33 (2012)
13. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21. ACM (2007)
14. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
15. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
16. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA, pp. 122–132. ACM (2011)
17. Christakis, M., Gotovos, A., Sagonas, K.F.: Systematic testing for detecting concurrency errors in erlang programs. In: ICST, pp. 154–163. IEEE Computer Society (2013)

18. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. STTT **2**(3), 279–287 (1999)
19. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 84–104. Springer, Heidelberg (2013)
20. Demsky, B., Lam, P.: SATCheck: SAT-directed stateless model checking for SC and TSO. In: OOPSLA 2015, pp. 20–36. ACM (2015)
21. Derevenetc, E., Meyer, R.: Robustness against power is PSpace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014, Part II. LNCS, vol. 8573, pp. 158–170. Springer, Heidelberg (2014)
22. Flanagan, C., Freund, S.N.: Adversarial memory for detecting destructive races. In: PLDI, pp. 244–254. ACM (2010)
23. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM (2005)
24. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
25. Godefroid, P.: Model checking for programming languages using verisoft. In: POPL, pp. 174–186. ACM Press (1997)
26. Godefroid, P.: Software model checking: the VeriSoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005)
27. Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. Form. Methods Syst. Des. **31**(3), 281–305 (2007)
28. IBM: Power ISA, Version 2.07 (2013)
29. Intel Corporation: Intel 64 and IA-32 Architectures Software Developers Manual (2012)
30. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: FMCAD, pp. 111–119. IEEE (2010)
31. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI, pp. 187–198. ACM (2011)
32. Lamport, L.: How to make a multiprocessor that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
33. Leonardsson, C.: Nidhugg. https://github.com/nidhugg/nidhugg
34. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 339–353. Springer, Heidelberg (2013)
35. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI, pp. 429–440. ACM (2012)
36. Mador-Haim, S., et al.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012)
37. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI, pp. 267–280. USENIX (2008)
38. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Trans. Comput. **48**(2), 227–235 (1999)
39. Peled, D.A.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 403–423. Springer, Heidelberg (1993)
40. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: ACSD, pp. 132–141. IEEE Computer Society (2012)

41. Sarkar, S., Memarian, K., Owens, S., Batty, M., Sewell, P., Maranget, L., Alglave, J., Williams, D.: Synchronising C/C++ and POWER. In: PLDI, pp. 311–322. ACM (2012)
42. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI, pp. 175–186. ACM (2011)
43. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2007)
44. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**(2), 282–312 (1988)
45. SPARC International Inc.: The SPARC Architecture Manual Version 9 (1994)
46. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: PLDI, pp. 341–350. ACM (2010)
47. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Advances in Petri Nets 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
48. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: IPDPS. IEEE (2004)
49. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI, pp. 250–259. ACM (2015)