Stateless Model Checking for POWER

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson
Dept. of Information Technology, Uppsala University, Sweden

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 [7]. 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, [28,44,27,8], 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) [31].

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) [24] 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 can easily reproduce detected bugs. SMC has been successfully implemented in tools, such as VeriSoft [25], CHESS [36], Concuerror [16], rinspect [48], 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 [46,35,23,17], adapted to SMC as dynamic
partial order reduction (DPOR) \cite{2,22,42,39}. 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 \cite{1,48}, 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 \cite{20,41,40,11}, 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 \cite{7,35,6}, avoid such redundancy by being defined in terms of an abstract representation of programmer-observable behavior, due to Shasha and Snir \cite{43}, 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 [7].

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 dif-
ferent 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 alter-
native 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 [32]. 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 compar-
ing with an existing state of the art analysis tool for POWER: goto-instrument [4].

Outline. The next section presents our derivation of execution models. Section 3
presents our RSMC algorithm, and Section 4 presents our implementation and exper-
iments. Proofs of all theorems, and formal definitions, are provided in the appendix.
Our implementation is available at [32].

2 Execution Model for Relaxed Memory Models

POWER — a Brief Glimpse. The programmer-observable behavior of POWER mul-
tiprocessors 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
\[ \langle \text{prog} \rangle ::= \langle \text{varinit} \rangle^* \langle \text{thrd} \rangle^+ \]
\[ \langle \text{varinit} \rangle ::= \langle \text{var} \rangle \#Z \]
\[ \langle \text{thrd} \rangle ::= \text{`thread'} \langle \text{tid} \rangle:;' \langle \text{linstr} \rangle^+ \]
\[ \langle \text{linstr} \rangle ::= \langle \text{label} \rangle:;' \langle \text{instr} \rangle :; \]
\[ \langle \text{instr} \rangle ::= \langle \text{reg} \rangle :=' \langle \text{expr} \rangle | \]
\[ \text{// register assignment} \]
\[ \text{`if'} \langle \text{expr} \rangle \text{`goto'} \langle \text{label} \rangle | \]
\[ \text{// conditional branch} \]
\[ \langle \text{reg} \rangle :=' \langle \text{expr} \rangle \text{[} \text{`]' } | \]
\[ \text{// memory load} \]
\[ \text{`[}' \langle \text{expr} \rangle \text{`]':='} \langle \text{expr} \rangle | \]
\[ \text{// memory store} \]
\[ \text{`sync'} | \text{`lwsync'} | \text{`isync'} \]
\[ \text{// fences} \]
\[ \langle \text{expr} \rangle ::= \text{(arithmetic expression over literals and registers)} \]

**Fig. 1.** The grammar of concurrent programs

\[
\begin{array}{c}
\text{x} = 0 \\
\text{thread P:} \\
\text{L0: } r_0 := x; \\
\text{L1: } y := r_0+1; \text{L3: } x := 1; \\
\text{L0: } r_0 := x & \text{data} \rightarrow \text{rf} \\
\text{L2: } r_1 := y & \text{rf} \\
\text{L1: } y := r_0+1 \\
\text{L3: } x := 1 \\
\end{array}
\]

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

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 \( \text{`r} :=' \text{[} a ] \text{'} \) loads the value from the memory address given by the arithmetic expression \( a \) into the register \( \text{r} \). A store \( \text{`[} a ] :=' a_1 \) 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 \( \text{[}x] \), where \( x \) is the address of \( x \). The instructions \( \text{sync} \), \( \text{lwsync} \), \( \text{isync} \) 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 [7], operate on an abstract representation of observable program behavior, introduced by Shasha and Snir [43], here called *traces*. A trace is a directed graph, in which vertices are executed instructions (called *events*), and edges capture dependencies between them. More precisely, a trace \( \pi \) is a quadruple \( (E, \text{po, co, rf}) \) where \( E \) is a set of *events*, and \( \text{po, co, and rf} \) are relations over the set of all possible events. Let \( E \) denote the set of all possible events. For an

---

1. [7] uses the term “execution” to denote what we call “trace.”
| Event | Parameter | Semantic Meaning |
|-------|-----------|------------------|
| L3: x := 1 | 0 | First in coherence order for x |
| L0: r₀ := x | L3 | Read value 1 from L3 |
| L1: y := r₀+1 | 0 | First in coherence order for y |
| L2: r₁ := 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.

...
events (i.e., events in \( E \)) only. The set of all possible states is denoted \( S \). The initial state \( \sigma_0 \in S \) is defined as \( \sigma_0 = (\lambda_0, E_0, E_0, \emptyset, \emptyset) \) where \( \lambda_0 \) is the function providing the initial label of each thread, and \( E_0 \) is the set of initializer events for all memory locations.

**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 \( \text{cb} \), which associates with each state \( \sigma = (\lambda, F, E, po, co, rf) \), a commit-before order \( \text{cb}_\sigma \subseteq F \times F \), which is a partial order on the set of fetched events. For each state \( \sigma \), the commit-before order \( \text{cb}_\sigma \) induces a predicate \( \text{enabled}_\sigma \) over the set of fetched events \( e \in F \) such that \( \text{enabled}_\sigma (e) \) holds if and only if \( e \notin E \) and the set \( \{ e' \in F \mid (e', e) \in \text{cb}_\sigma \} \) 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 \( \text{val}_\sigma (e, a) \) denotes the value taken by the arithmetic expression \( a \), when evaluated at the event \( e \) in the state \( \sigma \). The value is computed in the natural way, respecting data-flow. (Formal definition given in Appendix A.1) For example, in the state \( \sigma \) corresponding to the trace given in Fig. 2 where \( e \) is the event corresponding to label L1, we would have \( \text{val}_\sigma (e, t_0*1) = 2 \). The function \( \text{address}_\sigma (e) \) associates with each load or store event \( e \) the memory location accessed. For a label \( l \), let \( \lambda_{\text{next}} (l) \) denote the next label following \( l \) in the program code. Finally, for a state \( \sigma \) with coherence order \( co \) and a store \( e \) to some memory location \( x \), we let \( \text{extend}_\sigma (e) \) denote the set of coherence orders \( co' \) which result from inserting \( e \) anywhere in the total order of stores to \( x \) in \( co \). For each such order \( co' \), we let \( \text{position}_{\text{co}'} (e) \) denote the position of \( e \) in the total order: I.e. \( \text{position}_{\text{co}'} (e) \) 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 \( \mathcal{P} \) 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 $n = \text{position}_{w}^{\sigma}(e)$ in the coherence order. The guard $M(\text{exec}(\sigma'))$ 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 $M(\text{exec}(\sigma'))$ ensures that the resulting state is allowed.

Given two states $\sigma, \sigma' \in \mathbb{S}$, we use $\sigma \xrightarrow{\text{FLB}(\max)} \sigma'$ to denote that $\sigma \xrightarrow{\text{FLB}}^{*} \sigma'$ and there is no state $\sigma'' \in \mathbb{S}$ with $\sigma \xrightarrow{\text{FLB}} \sigma''$. A run $\tau$ from some state $\sigma$ is a sequence of parameterized events $e_{1}[p_{1}], e_{2}[p_{2}], \ldots, e_{k}[p_{k}]$ such that $\sigma \xrightarrow{\text{FLB}(\max)} \sigma_{1} \xrightarrow{e_{1}[p_{1}]} \sigma_{1}' \xrightarrow{\text{FLB}(\max)} \ldots \xrightarrow{e_{k}[p_{k}]} \sigma_{k}' \xrightarrow{\text{FLB}(\max)} \sigma_{k+1}$ 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 $\tau$. Observe that the
sequence $\tau$ leads to a uniquely determined state $\sigma_{k+1}$, which we denote $\tau(\sigma)$. A run $\tau$, from the initial state $\sigma_0$, is complete iff the reached trace $\text{exec}(\tau(\sigma_0))$ is complete. Fig. [3] shows an example complete run of the program in Fig. [2](left).

In summary, our execution model represents a program $P$ as a labeled transition system $TS_{M,cb}^P = (S, \sigma_0, \rightarrow)$, where $S$ is the set of states, $\sigma_0$ is the initial state, and $\rightarrow \subseteq S \times (P \cup \{FLB\}) \times S$ is the transition relation. We define the execution semantics under $M$ and $cb$ as a mapping, which maps each program $P$ to its denotation $[P]^E_M$, which is the set of complete runs $\tau$ induced by $TS_{M,cb}^P$.

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

We say that cb is monotonic w.r.t. $M$ if whenever $\sigma \leq_{cb} \sigma'$, then (i) $M(\text{exec}(\sigma')) \Rightarrow M(\text{exec}(\sigma))$, (ii) $cb_\sigma \subseteq cb_{\sigma'}$, and (iii) for all $e \in F$ such that either $e \in E$ or $(\text{enabled}_\sigma(e) \land e \notin E')$, we have $(e', e) \in cb_\sigma \Leftrightarrow (e', e) \in cb_{\sigma'}$ 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 $\sigma$ induces a number of relations over its fetched (possibly committed) events. Following [7], we let $\text{addr}_\sigma$, $\text{data}_\sigma$, $\text{ctrl}_\sigma$, denote respectively address dependency, data dependency and control dependency. Similarly, $\text{po-loc}_\sigma$ is the subset of $\text{po}$ that relates memory accesses to the same memory location. Lastly, $\text{sync}_\sigma$ and $\text{lwsync}_\sigma$ relate events that are separated in program order by respectively a sync or lwsync. The formal definitions can be found in [7], as well as in Appendix A.1. We can now define a weakest reasonable commit-before function $cb^0$, capturing natural dependencies:

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

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 $\sigma$ such that $M(\text{exec}(\sigma))$ we have that $cb_\sigma$ is acyclic and $cb^0_\sigma \subseteq cb_\sigma$.

**Theorem 1 (Equivalence with Axiomatic Model).** Let cb be a commit-before function valid w.r.t. a memory model $M$. Then $[P]_{Acb}^M = \{\text{exec}(\tau(\sigma_0)) \mid \tau \in [P]_{M,cb}^E\}$.

The commit-before function $cb^0$ is valid w.r.t. $M^\text{POWER}$, implying (by Theorem 1) that $[P]_{M,cb^0}^E$ 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. Fig. [5] shows an example of how the POWER semantics may deadlock when based on $cb^0$. 
Program Blocked run τ Blocked state σ

\begin{align*}
x &= 0 & y &= 0 & \text{L3} &: r_0: y \leftarrow \text{data} & \text{L3}: x := 3 \\
\text{thread P}: & \text{thread Q}: & \text{L5} &: & & \text{sync} \\
\text{L0}: r_0: y; & \text{L3}: x:=3; & \text{L0}[\text{L5}] & & \text{L1}: x := r_0; & \text{L4}: \text{sync} \\
\text{L1}: x := r_0; & \text{L4}: \text{sync}; & \text{L2}[\text{L0}] & & \text{L2}: x := 2; & \text{L5}: y := 1; \\
\text{L2}: x := 2; & \text{L5}: y := 1; & \text{(L1 blocked)} & & \text{L2}: x := 2 & \text{L5}: y := 1
\end{align*}

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 \( \tau \) (center) we reach a state \( \sigma \) (right) where L0, L2, L3-L5 are successfully committed. However, any attempt to commit L1 will close a cycle in the relation \( \text{co; sync}_\sigma; \text{rf}_\sigma; \text{data}_\sigma; \text{po-loc}_\sigma \), which is forbidden under POWER. This blocking behavior is prevented when the stronger commit-before function \( cb^{\text{power}} \) is used, since it requires L1 and L2 to be committed in program order.

We say that a memory model \( M \) and a commit before function \( cb \) are **deadlock free** if for all runs \( \tau \) from \( \sigma_0 \) and memory access events \( e \) such that \( \text{enabled}_{\tau(\sigma_0)}(e) \) there exists a parameter \( p \) such that \( \tau.e[p] \) is a run from \( \sigma_0 \). 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:

\[
\begin{align*}
\text{cb}^{\text{power}} = (\text{cb}^0 \cup (\text{addr}_\sigma \cup \text{po-loc}_\sigma \cup \text{sync}_\sigma \cup \text{lwsync}_\sigma))^+
\end{align*}
\]

**Theorem 2.** \( \text{cb}^{\text{power}} \) is valid w.r.t. \( M^{\text{POWER}} \).

**Theorem 3.** \( M^{\text{POWER}} \) and \( \text{cb}^{\text{power}} \) 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 any possible parameter value \( p \) of \( e \) in the current state, RSMC extends the state by \( e[p] \) and continues the exploration recursively from the new state.

(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 in the opposite direction (store followed by load) 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. An analogous observation applies to occurrences of several stores to the same memory location.

| 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[L0] 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 \texttt{Explore}, which takes parameters \(\tau\) and \(\sigma\) such that \(\sigma = \tau(\sigma_0)\). \texttt{Explore} will explore all states that can be reached by complete runs extending \(\tau\).

First, on line 1, we fetch instructions and commit all local instructions as far as possible from \(\sigma\). 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.

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 5 i.e., for each coherence position \(n\) that is allowed for \(e\) by the memory model, we explore the continuation of \(\tau\) obtained by committing \(e[n]\).

Finally, we call \texttt{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 \texttt{DetectRace}, and discussed later. On line 11 we compute all allowed parameters for the load \(e\). They are (some of the) stores in \(\tau\) which access the same address as \(e\). On line 13 we make one recursive call to \texttt{Explore} per allowed parameter. The structure of this exploration is illustrated in the two branches from \(\sigma_1\) to \(\sigma_2\) and \(\sigma_5\) in Fig. 8(a).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig7.png}
\caption{An algorithm to explore all traces of a given program. The initial call is \texttt{Explore}(.\(), \sigma_0\)).}
\end{figure}
Notice in the above that both for stores and loads, the available parameters are determined entirely by $\tau$, 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 $\tau$. In the case of loads, the parameters are the earlier stores occurring in $\tau$. 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 $\sigma_0$ to $\sigma_4$ 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 $\tau_2$, which is a template for a new run, including precisely the events in $\tau_1$ which are cb-before the store $\hat{e}_w$. The run template $\tau_2$ can be explored from the state $\sigma_1$ (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 $\tau_2$ is computed from the complete run in DetectRace on lines 23. This is done by first removing (at line 2) the prefix $\tau_0$ which precedes $e_r$ (stored in $P[\hat{e}_r]$) 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, $cut(\tau, e, \sigma)$ restricts $\tau$ to the events which are cb,-before $e$), and the resulting run is normalized. The function 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. (Formal definitions in Appendix A.2.) The run template $\tau_2.\hat{e}_w[*].e_r[\hat{e}_w]$ is then stored on line 5 in the set $Q[\hat{e}_r]$, 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 $Q[\hat{e}_r]$ are explored from the same call to Explore($\tau_0, \sigma_1$) 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 $\sigma_0$ to $\sigma_8$. Notice on line 18 that the new branch is explored by the function Traverse, rather than by Explore itself. This has the effect that $\tau_2$ is traversed, with each event using the parameter given in $\tau_2$, 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 5 in Traverse).

Let us now consider the situation in Fig. 8(b) in the run from $\sigma_0$ to $\sigma_{10}$. 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 $\hat{e}_w$. From earlier DPOR algorithms such as e.g. [22], one might expect that this case is handled by exploring a new branch of the form $\tau_2.\hat{e}_w[p].\tau_3.\hat{e}_w[p'].e_r[\hat{e}_w]$, where $e_r$ is simply delayed after $\sigma_7$ until $\hat{e}_w$ has been committed. Our algorithm handles the case differently, as shown in the run from $\sigma_0$ to $\sigma_{13}$. Notice that $P[\hat{e}_r]$ 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., $\sigma_1$ in Fig. 8(b). We start the new branch from that position ($\sigma_1$), rather than from the position where $e_r$ was committed when the race was detected (i.e., $\sigma_7$). The new branch $\tau_4$ is constructed when the race is detected on lines 24-23 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 $\hat{e}_w$.

The reason for returning all the way up to $\sigma_1$, rather than starting the new branch at $\sigma_7$, 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 $\sigma_0$ to $\sigma_3$ 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 $\sigma_0$ to $\sigma_6$ and from $\sigma_0$ to $\sigma_9$. 
respectively. In the run ending in $\sigma_9$, we discover the race from $L_0$ to $L_1$ again. This indicates that a run should be explored where $L_0$ reads from $L_1$. If we were to continue exploration from $\sigma_7$ by delaying $L_0$ until $L_1$ has been committed, we would follow the path from $\sigma_7$ to $\sigma_{11}$ in Fig. 10. In $\sigma_{11}$, we have successfully reversed the race between $L_0$ and $L_1$. However, the trace of $\sigma_{11}$ turns out to be identical to the one we already explored in $\sigma_6$. 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 $L_0$ to $L_1$ is discovered at $\sigma_9$, we consider the entire sub-run $L_2[0] \cdot L_0[L2] \cdot L_1[1]$ from $\sigma_0$, and construct the new sub-run $L_1[*] \cdot L_0[L1]$ by removing all events that are not cb-before $L_1$, generalizing the parameter to $L_1$, and by appending $L_0[L1]$ to the result. The new branch $L_1[*] \cdot L_2[L1]$ is added to $Q[L0]$. But $Q[L0]$ already contains the branch $L_1[*] \cdot L_2[L1]$ 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 $R$:

$L_0$: $r := x$  
$L_1$: $x := 1$  
$L_2$: $x := 2$

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

**Fig. 10.** Part of a faulty exploration tree for the program above, containing redundant branches. The branches ending in $\sigma_6$ and $\sigma_{11}$ correspond to the same trace. The **Explore** 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 $Q[e_r]$.

**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 \( \pi \in [P]_M^{Ax} \), the evaluation of a call to \( \text{Explore}(\langle \rangle, \sigma_0) \) will contain a recursive call to \( \text{Explore}(\tau, \sigma) \) for some \( \tau, \sigma \) such that \( \text{exec}(\sigma) = \pi \).

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

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 \( M = M^{POWER} \) and \( cb = cb^{power} \). Let \( \pi \in [P]_M^{Ax} \). Then during the evaluation of a call to \( \text{Explore}(\langle \rangle, \sigma_0) \), there will be exactly one call \( \text{Explore}(\tau, \sigma) \) such that \( \text{exec}(\sigma) = \pi \).

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 Section 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 [32], 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 the correctness of our implementation by successfully running all 8070 relevant litmus tests published with [7].

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

---

[2] The cbmc tool previously supported POWER [5], but has withdrawn support in later versions.
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 seconds. 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.

| Benchmark       | goto-instrument | Nidhugg    |      |     |     |
|-----------------|-----------------|------------|------|-----|-----|
|                 | $F$ | $LB$ | time   | $time$ | $SS$ | $B$ |
| dcl_singleton    | 7   | *0.40| 0.13   | 3     | 0    |     |
| dcl_singleton    | y   | 5.05| 0.19   | 7     | 0    |     |
| dekker           | 10  | *229.39| *0.11 | 5     | 0    |     |
| dekker           | y   | t/o | 0.76   | 246   | 0    |     |
| fib_false        |     |     | *1.86  |       |      |     |
| 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   | t/o | 0.20   | 50    | 2    |     |
| parker           | 5   | 4.20| *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   | t/o | 0.58   | 16    | 0    |     |
| pgsql_bnd        |     |     | *0.11  | 2     | 0    |     |
| pgsql_bnd        | y   | t/o | 0.58   | 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    |     |

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 [4]. 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 is optimal, in the sense that it 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 Section 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 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 0.01s with fences and 161.36s without fences.

We compare this with the results of the litmus test checking tool herd [7], 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 925.95s with fences and 78.09s 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 have presented the first framework for efficient application of SMC to programs running under POWER. Our framework 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. The RSMC algorithm 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 [1,48,19]. In this work we propose a novel and efficient SMC technique for programs running under POWER.

In [7], 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., [3,10,20]). Some of these works propose precise analysis techniques for finite-state programs under relaxed memory models (e.g., [3,10,20]). Others propose algorithms and tools for monitoring and testing programs running under relaxed memory models (e.g., [13,4,5]). 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., [26,37,29,33,30]). There are also a number of efforts to design bounded model checking techniques for programs under relaxed memory models (e.g., [5,12,45]) which encode the verification problem in SAT/SMT. Finally, there are code-to-code transformation techniques (e.g., [9]) 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 [20], the robustness problem for POWER has been shown to be PSPACE-complete.

The closest works to ours were presented in [5,17,4]. The work [4] extends cbmc to work with relaxed memory models (such as TSO, PSO and POWER) using a code-to-code transformation. The work in [5] 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 [5], but has withdrawn support in its later versions. The tool herd [7] 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 Section 4, we experimentally compare RSMC with the tools of [4] and [7].

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.F.: Stateless model checking for TSO and PSO. In: TACAS, LNCS, vol. 9035, pp. 353–367. Springer (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., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS, LNCS, vol. 7214, pp. 204–219. Springer (2012)
4. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: ESOP, LNCS, vol. 7792, pp. 512–532. Springer (2013)
5. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV, LNCS, vol. 8044, pp. 141–157. Springer (2013)
6. Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV, LNCS, vol. 6806, pp. 50–66. Springer (2011)
7. 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)
8. ARM: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2014)
9. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV, LNCS, vol. 6806, pp. 59–115. Springer (2011)
10. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP, LNCS, vol. 7792, pp. 533–553. Springer (2013)
11. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: EXPRESS/SOS 2012. EPTCS, vol. 89, pp. 19–33 (2012)
12. 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)
13. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV, LNCS, vol. 5123, pp. 107–120. Springer (2008)
14. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS. pp. 11–25. Springer (2011), LNCS 6605
15. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA. pp. 122–132. ACM (2011)
16. 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)
17. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. STTT 2(3), 279–287 (1999)
18. Dan, A.M., Moshman, Y., Vechev, M.T., Yahav, E.: Predicate abstraction for relaxed memory models. In: SAS. LNCS, vol. 7935, pp. 84–104. Springer (2013)
19. Demsky, B., Lam, P.: SATCheck: SAT-directed stateless model checking for SC and TSO. In: OOPSLA 2015, pp. 20–36. ACM (2015)
20. Derevenetc, E., Meyer, R.: Robustness against Power is PSpace-complete. In: ICALP (2). LNCS, vol. 8573, pp. 158–170. Springer (2014)
21. Flanagan, C., Freund, S.N.: Adversarial memory for detecting destructive races. In: PLDI. pp. 244–254. ACM (2010)
22. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL. pp. 110–121. ACM (2005)
23. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
24. Godefroid, P.: Model checking for programming languages using verisoft. In: POPL. pp. 174–186. ACM Press (1997)
25. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design 26(2), 77–101 (2005)
26. Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. Formal Methods in System Design 31(3), 281–305 (2007)
27. IBM: Power ISA, Version 2.07 (2013)
28. Intel Corporation: Intel 64 and IA-32 Architectures Software Developers Manual (2012)
29. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: FM-CAD. pp. 111–119. IEEE (2010)
30. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI. pp. 187–198. ACM (2011)
31. Lamport, L.: How to make a multiprocessor that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
32. Leonardsson, C.: Nidhugg. https://github.com/nidhugg/nidhugg
33. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. LNCS, vol. 7795, pp. 339–353. Springer (2013)
34. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440. ACM (2012)
35. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for POWER multiprocessors. In: CAV. LNCS, vol. 7358, pp. 495–512. Springer (2012)
36. 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)
37. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Trans. Computers 48(2), 227–235 (1999)
38. Peled, D.A.: All from one, one for all: on model checking using representatives. In: CAV. LNCS, vol. 697, pp. 409–423. Springer (1993)
39. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: ACSD. pp. 132–141. IEEE Computer Society (2012)
40. 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)
41. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI. pp. 175–186. ACM (2011)
42. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: HVC. LNCS, vol. 4383, pp. 166–182. Springer (2007)
43. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
44. SPARC International, Inc.: The SPARC Architecture Manual Version 9 (1994)
45. Törlak, E., Vaziri, M., Dolby, J.: Memsat: checking axiomatic specifications of memory models. In: PLDI. pp. 341–350. ACM (2010)
46. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri Nets. LNCS, vol. 483, pp. 491–515. Springer (1989)
47. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS. IEEE (2004)
48. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI. pp. 250–259. ACM (2015)
Appendix Overview. These appendices contain formal definitions and proofs elided from the main text.

Appendix A contains formal definitions of some concepts used in the semantics (Section 2) and RSMC algorithm (Section 3).

Appendix B contains proofs of theorems about the execution model (Section 2): In particular the proof of equivalence between an execution model and the axiomatic model it is derived from, the proof of validity of cbpower, and the deadlock freedom of Mpower and cbpower.

Appendix C contains proofs of theorems about the RSMC algorithm (Section 3): In particular the proof of soundness, and the proof of optimality.

A Additional Formal Definitions

Here we provide some formal definitions that were elided from the main text.

A.1 Additional Definitions for Section 2

\[ x = 0 \]
\[ y = 0 \]

thread \( P \): thread \( Q \):
\[
\begin{aligned}
L0: & \quad x := 1; \\
L3: & \quad r_1 := y; \\
L1: & \quad lwfence; \\
L4: & \quad ffence; \\
L2: & \quad y := 1; \\
L5: & \quad r_2 := x;
\end{aligned}
\]

Fig. 12. An example program: MP+lwfence+ffence.

In the following, we introduce some notations and definitions following [7] that are needed in order to define dependencies between events. We also give the formal definition of the partial function \( \text{val}_\sigma \) which gives the evaluation of an arithmetic expression.

Let \( \sigma = (F, E, po, co, rf) \in \mathcal{S} \) be a state. We define two partial functions \( \text{val}_\sigma \) and \( \text{adeps}_\sigma \) over the set of events and arithmetic expression so that \( \text{val}_\sigma(e, a) \) is the value of the arithmetic expression \( a \) when evaluated at the event \( e \in F \) in the state \( \sigma \), and \( \text{adeps}_\sigma(e, a) \) is the set of load events in \( F \) which are dependencies for the evaluation of the arithmetic expression \( a \) at the event \( e \). Here, \( \text{val}_\sigma(e, a) \) can be undefined (\( \text{val}_\sigma(e, a) = \bot \)) when the value of \( a \) at the event \( e \) depends on the value of a load which is not yet executed. Formally, we define \( \text{val}_\sigma(e, a) \) and \( \text{adeps}_\sigma(e, a) \) recursively, depending on the type of arithmetic expression:

- If \( a \) is a literal integer \( i \), then \( \text{val}_\sigma(e, a) = i \) and \( \text{adeps}_\sigma(e, a) = \emptyset \).
If \( a = f(a_0, \ldots, a_n) \) for some arithmetic operator \( f \) and subexpressions \( a_0, \ldots, a_n \), then \( \text{val}_\sigma(e, a) = f(\text{val}_\sigma(e, a_0), \ldots, \text{val}_\sigma(e, a_n)) \) and \( \text{adeps}_\sigma(e, a) = \bigcup_{i=0}^n \text{adeps}_\sigma(e, a_i) \).

If \( a = r \) for some register \( r \), then let \( e_r \in F \) be the \( p_0 \)-greatest event such that \( (e_r, e) \in p_0 \) and either \( \text{instr}(e_r) = r := a' \) or \( \text{instr}(e_r) = r : = [a'] \) for some expression \( a' \).

- If there is no such event \( e_r \), then \( \text{val}_\sigma(e, a) = 0 \) and \( \text{adeps}_\sigma(e, a) = \emptyset \).
- If \( \text{instr}(e_r) = r := a' \), then \( \text{val}_\sigma(e, a) = \text{val}_\sigma(e_r, a') \) and \( \text{adeps}_\sigma(e, a) = \text{adeps}_\sigma(e_r, a') \).
- If \( \text{instr}(e_r) = r : = [a'] \) and \( e_r \in E \) then let \( e_w \in E \) be the event such that \( (e_w, e_r) \in \text{rf} \). Let \( [a''] : = a''' \) be the arithmetic expressions s.t. \( \text{instr}(e_u) = [a'''] : = a''' \). Now we define \( \text{val}_\sigma(e, a) = \text{val}_\sigma(e_w, a'''') \) and \( \text{adeps}_\sigma(e, a) = \{ e_r \} \).
- If \( \text{instr}(e_r) = r : = [a'] \) and \( e_r \notin E \) then \( \text{val}_\sigma(e, a) = \bot \) and \( \text{adeps}_\sigma(e, a) = \{ e_r \} \).

We overload the function \( \text{adeps}_\sigma \) for event arguments:

- If \( \text{instr}(e) = r := a \), then \( \text{adeps}_\sigma(e) = \text{adeps}_\sigma(e, a) \).
- If \( \text{instr}(e) = \text{if } a \text{ goto } l \), then \( \text{adeps}_\sigma(e) = \text{adeps}_\sigma(e, a) \).
- If \( \text{instr}(e) = r : = [a] \), then \( \text{adeps}_\sigma(e) = \text{adeps}_\sigma(e, a) \).
- If \( \text{instr}(e) = [a] : = a' \), then \( \text{adeps}_\sigma(e) = \text{adeps}_\sigma(e, a) \cup \text{adeps}_\sigma(e, a') \).
- If \( \text{instr}(e) \in \{ \text{sync, lwsync, isync} \} \), then \( \text{adeps}_\sigma(e) = \emptyset \).

\[
x = 0 \\
y = 0
\]

\[
\text{thread } P: \quad \text{thread } Q \\
\text{L0: } r_0 := x; \quad \text{L3: } r_1 := x; \\
\text{L1: if } r_0 = 1 \text{ goto L0; } \text{L4: } [r_1] := 1 \\
\text{L2: } x := 1; \quad \text{L5: } r_2 := y;
\]

**Fig. 13.** A program with address and control dependencies.

We also define the address dependency relation \( \text{addr}_\sigma \subseteq (F \times F) \) to capture how events depend on earlier loads for the computation of their address. For a memory access event \( e \) with \( \text{instr}(e) \) is of the form \([a] : = a'\) or \( r : = [a] \), we have \( (e', e) \in \text{addr}_\sigma \) for any event \( e' \in \text{adeps}_\sigma(e, a) \). For instance, in the example described in Figure 13, there is an address dependency between the load L3 and the store L4.

We define the data dependency relation \( \text{data}_\sigma \subseteq (F \times F) \) to capture how events depend on earlier loads for the computation of their data. For an event \( e \) with \( \text{instr}(e) \) is of the form \( r : = a \), if \( a \) goto \( l \) or \([a'] : = a\), we have \( (e', e) \in \text{data}_\sigma \) for any event \( e' \in \text{adeps}_\sigma(e, a) \). For instance, in the example described in Figure 2, there is a data dependency between the load L0 and the store L1.
We define the relation \( \text{ctrl}_\sigma \subseteq F \times F \) to capture how the control flow to an event depends on earlier loads. For two events \( e \in F \) and \( e' \in F \) we have \( (e, e') \in \text{ctrl}_\sigma \) iff \( \text{instr}(e) = r:*[a'] \) (i.e., \( e \) is a load event) and there is a branch event \( e_b \) with \( \text{instr}(e_b) = \text{if} \ a \ \text{goto} \ l \) for some arithmetic expression \( a \) and label \( l \) such that \( (e, e_b), (e_b, e') \in \text{po} \) and \( e \in \text{adeps}_\sigma(e_b, a) \). In the example given in Figure 13 there is a control dependency between the load \( L0 \) and the store \( L2 \).

We define the relation \( \text{po-loc}_\sigma \subseteq F \times F \) to capture the program order between accesses to the same memory location: \( \text{po-loc}_\sigma = \{(e, e') \in \text{po} | \text{address}_\sigma(e) = \text{address}_\sigma(e') \land \text{address}_\sigma(e) \neq \bot\} \). In the example described in Figure 13 the pair \((L0, L2)\) is in \( \text{po-loc}_\sigma \).

Finally, we define the relations \( \text{sync}_\sigma, \text{lwsync}_\sigma \subseteq F \times F \) that contain the set of pairs of events that are separated by the fence instruction \( \text{sync} \) and \( \text{lwsync} \) respectively. For instance, in the example described in Figure 12 \( \text{sync}_\sigma \) and \( \text{lwsync}_\sigma \) will contain the pairs \((L3, L5)\) and \((L0, L2)\), respectively. We then define \( \text{lwsync}_\sigma = \{(e, e') \in \text{lwsync}_\sigma | \neg(e \text{ is a store, and } e' \text{ is a load})\} \), corresponding to the intuition that the order between a store and a later load is not enforced by an \( \text{lwsync} \) under \text{POWER}.

### A.2 Additional Definitions for Section 3

#### Definition of the cut Function

In order to define the cut function, we need to define an auxiliary function \( \text{cut}' \). We then define \( \text{cut}(\tau, e, \sigma) = \text{cut}'(\tau, e, \sigma, \lambda a. \varnothing) \).

The function \( \text{cut}'(\tau, e, \sigma, W) \) works by recursively traversing \( \tau \) and removing each event which is not \( \text{cb}_\sigma \)-before \( e \). While doing so, for each store \( e_w[n] \) that is removed, the parameter \( n \) is stored in \( W(\text{address}_\sigma(e_w)) \). When a store \( e_w[n] \) is retained in the run, its parameter \( n \) is updated to reflect that all the preceding stores with parameters \( W(\text{address}_\sigma(e_w)) \) have disappeared. Formally, the function \( \text{cut}' \) is defined as follows:

\[
\text{cut}'(\langle \rangle, e, \sigma, W) = \langle \rangle
\]

\[
\text{cut}'(e_0[p_0], \tau, e, \sigma, W) =
\begin{cases}
  e_0[p_0].\text{cut}'(\tau, e, \sigma, W) & \text{if } e_0 \in R \land (e_0, e) \in \text{cb}_\sigma \\
  \text{cut}'(\tau, e, \sigma, W) & \text{if } e_0 \in R \land (e_0, e) \notin \text{cb}_\sigma \\
  e_0[p_0].\text{cut}'(\tau, e, \sigma, W) & \text{if } e_0 \in W \land (e_0, e) \in \text{cb}_\sigma \\
\end{cases}
\]

where \( p_0 = p_0 - |\{i \in A | i < p_0\}| \)

where \( A = W(\text{address}_\sigma(e_0)) \)

where \( W' = W[a \leftarrow W(a) \cup \{p_0\}] \)

where \( a = \text{address}_\sigma(e_0) \)

### B Proofs for Section 2

Here we provide proofs for the various theorems appearing in Section 2.
B.1 Proof of Theorem 1 (Equivalence of Semantics)

Proof (Proof of theorem 1). We prove first that \( \{ \text{exec}(\tau(\sigma_0)) \mid \tau \in [P]^{Ex}_{M, cb} \} \subseteq [P]^{Ax}_{M, cb} \). This follows directly from the fact that every rule in the operational semantics checks the new state against \( M \) before allowing the transition, and from \( M(\sigma_0) \).

We turn instead to proving the other direction \( [P]^{Ax}_{M, cb} \subseteq \{ \text{exec}(\tau(\sigma_0)) \mid \tau \in [P]^{Ex}_{M, cb} \} \). Let \( \pi = (E, po, co, rf) \) be an execution in \( [P]^{Ax}_{M, cb} \). Let \( \sigma = (\lambda, E, E, po, co, rf) \) where \( \lambda \) maps every thread to its final state be the complete state corresponding to \( \pi \). From the assumption that \( cb \) is valid w.r.t. \( M \), we know that \( cb_\sigma \) is acyclic. Let \( \tau \) be some linearization w.r.t. \( cb_\sigma \) of the memory access events in \( E \), instantiated with parameters according to \( co \) and \( rf \). We will show that \( \tau \) is a run in \( [P]^{Ex}_{M, cb} \) such that \( \tau(\sigma_0) = \sigma \).

Let \( \tau = e_1[p_1], e_2[p_2], \ldots e_n[p_n] \), and for every \( 0 \leq i \leq n \), let \( \tau_i \) denote the prefix \( e_1[p_1], \ldots e_i[p_i] \) of \( \tau \). In the case that \( \tau_i \) is a run (it doesn’t block), let \( \sigma^i = (\lambda_i, F_i, E_i, po_i, co_i, rf_i) = \tau_i(\sigma_0) \) for all \( 0 \leq i \leq n \).

We will prove by induction that for all \( 0 \leq i \leq n \) it holds that \( \tau_i \) is a run, and \( \sigma^i \subseteq cb_\sigma \) and the restriction of \( E_i \) to memory access events is the set of events in \( \tau_i \).

Base case \( (i = 0) \): From the definition of runs, we see that \( \tau_0 = \langle \rangle \) is a run if there is a state \( \sigma^0 \) such that \( \sigma_0 \xrightarrow{\text{FLB(max)}} \sigma^0 \). This holds vacuously by the definition of \( \xrightarrow{\text{FLB(max)}} \).

Furthermore, we see from the definition of \( \xrightarrow{\text{FLB(max)}} \) that \( F_0 \) will consist of all events that can be fetched without committing any branch which depends on a load. The same events must necessarily be fetched in any complete state, and therefore we have \( F_0 \subseteq F \). We see that \( E_0 \) consists of all local instructions that do not depend on any memory access. For the same reason, the same events must also be committed in any complete state. And so we have \( E_0 \subseteq E \). It follows similarly that \( po_0 \subseteq po \). Since no memory access events have been committed, \( co_0 = \emptyset \subseteq co \) and \( rf_0 = \emptyset \subseteq rf \). Hence we have \( \sigma^0 \subseteq cb_\sigma \). No memory access events have been committed by \( \xrightarrow{\text{FLB(max)}} \), and no memory access events appear in \( \tau_0 = \langle \rangle \), and so the restriction of \( E_i \) to memory access events is the set of events in \( \tau_i \).

Inductive case \( (0 < i + 1 \leq n) \): We assume as inductive hypothesis that \( \tau_i \) is a run, and \( \sigma^i \subseteq cb_\sigma \) and the restriction of \( E_i \) to memory access events is the set of events in \( \tau_i \) for some \( 0 \leq i < n \).

We know that \( e_{i+1} \in F_i \), since all earlier branch events in \( F_i \) have been committed (in \( E_i \)) by \( \xrightarrow{\text{FLB(max)}} \) and all loads that depend on have been committed (notice \( (e_i, e_{i+1}) \in ctrl_\sigma \subseteq cb_\sigma \subseteq cb_\sigma \) for all such loads \( e_i \) with the same sources as in \( \sigma \). Since the restriction of \( E_i \) to memory access events is the set of events in \( \tau_i \), we know that \( e_{i+1} \notin E_{i+1} \). To show that \( \text{enabled}_\sigma(e_{i+1}) \) holds, it remains to show that for all events \( e \) such that \( (e, e_{i+1}) \in cb_\sigma \), it holds that \( e \in E_i \). This follows from the monotonicity of \( cb \) and \( M \) as follows: Since \( \sigma^i \subseteq cb_\sigma \), we have \( cb_\sigma \subseteq cb_\sigma \). Since \( \tau \) is a linearization of \( cb_\sigma \), for any \( e \) with \( (e, e_{i+1}) \in cb_\sigma \), it must hold that either \( e \) is a memory access, and then precedes \( e_{i+1} \) in \( \tau \) and is therefore already committed in \( \sigma^i \), or \( e \) is a local event which depends only on memory accesses that similarly precede \( e_{i+1} \), in which case \( e \) has been committed by \( \xrightarrow{\text{FLB(max)}} \). Hence we have \( \text{enabled}_\sigma(e_{i+1}) \).
We now split into cases, depending on whether \( e_{i+1} \) is a store or a load.

Assume first that \( e_{i+1} \) is a store. From the construction of \( \tau \), we know that the parameter \( p_{i+1} \) is a coherence position chosen such that \( p_{i+1} = \text{position}_{\text{co}, i+1} \) for some coherence order \( \text{co}_{i+1} \in \text{extend}_{\sigma^i}(e_{i+1}) \) such that \( e_{i+1} \) is ordered with the previous stores in \( E_i \) in the same order as in \( \text{co} \). In order to show that the rule \( \frac{e_{i+1}[p_{i+1}]}{\sigma^{i+1}} \) applies, we must first show that \( \text{enabled}_\sigma(e_{i+1}) \) holds. The rule \( \frac{e_{i+1}[p_{i+1}]}{\sigma^{i+1}} \) in the operational semantics will produce a state \( \sigma' = (\lambda_i, F_i, E_i \cup \{e_{i+1}\}, p_{i_0}, \text{co}_{i+1}, \tau_i) \), and then check whether \( M(\text{exec}(\sigma')) \) holds. In order to show that \( \tau_{i+1} \) is a run, we need to show that \( M(\text{exec}(\sigma')) \) does indeed hold. Since we have \( \sigma^i \leq \text{cb} \sigma \) and \( e_{i+1} \) was chosen for \( \tau \) from the committed events in \( \sigma \), and \( \text{co}_{i+1} \) orders \( e_{i+1} \) with \( E_i \) in the same way as co, we also have \( \sigma' \leq \text{cb} \sigma \). Then by the monotonicity of the memory model \( M \), we have \( M(\text{exec}(\sigma')) \). Hence \( \tau_{i+1} \) is a run, and \( \sigma^{i+1} = \tau_{i+1}(\sigma_0) \) for some \( \sigma^{i+1} \) such that \( \sigma' \xrightarrow{\text{FLB}(\text{max})} \sigma^{i+1} \). By the same argument as in the base case, it then follows that \( \sigma^{i+1} \leq \text{cb} \sigma \) and the restriction of \( E_{i+1} \) to memory access events is the set of events in \( \tau_{i+1} \).

Assume next that \( e_{i+1} \) is a load. From the construction of \( \tau \), we know that the parameter \( p_{i+1} \) is the store event \( e_w \) such that \( (e_w, e_{i+1}) \in \tau \). Since \( \tau \) is a linearization of \( \text{cb}_{\sigma} \), and \( (e_w, e_{i+1}) \in \tau \), we know that \( e_w \) appears before \( e_{i+1} \) in \( \tau \). Therefore we know that \( e_w \) is already committed in \( \sigma^i \), and that it is therefore available as a parameter for the load \( e_{i+1} \). Since all loads that precede \( e_w \) and \( e_{i+1} \) in \( \text{cb}_{\sigma} \), have been committed in the same way as in \( \sigma \) we know that the addresses accessed by \( e_w \) and \( e_{i+1} \) are computed in the same way in \( \sigma^i \) as in \( \sigma \) and therefore we have address\( _\sigma(e_w) = \text{address}_\sigma(e_{i+1}) \). The rule \( \frac{e_{i+1}[p_{i+1}]}{\sigma^{i+1}} \) in the operational semantics will produce a state \( \sigma' = (\lambda_i, F_i, E_i \cup \{e_{i+1}\}, p_{i_0}, \text{co}_{i+1}, \tau_i \cup \{(e_w, e_{i+1})\}) \), and then check whether \( M(\text{exec}(\sigma')) \) holds. In order to show that \( \tau_{i+1} \) is a run, we need to show that \( M(\text{exec}(\sigma')) \) does indeed hold. Since we have \( \sigma' \leq \text{cb} \sigma \) and \( e_{i+1} \) was chosen for \( \tau \) from the committed events in \( \sigma \), and \( e_w \) was chosen such that \( (e_w, e_{i+1}) \in \tau \), we also have \( \sigma' \leq \text{cb} \sigma \). Then by the monotonicity of the memory model \( M \), we have \( M(\text{exec}(\sigma')) \). Hence \( \tau_{i+1} \) is a run, and \( \sigma^{i+1} = \tau_{i+1}(\sigma_0) \) for some \( \sigma^{i+1} \) such that \( \sigma' \xrightarrow{\text{FLB}(\text{max})} \sigma^{i+1} \). By the same argument as in the base case, it then follows that \( \sigma^{i+1} \leq \text{cb} \sigma \) and the restriction of \( E_{i+1} \) to memory access events is the set of events in \( \tau_{i+1} \).

This concludes the inductive sub-proof.

Since \( \tau_n = \tau \) is a run, and \( \sigma^n = \tau(\sigma_0) \leq \text{cb} \sigma \), and the committed memory access events in \( \tau(\sigma_0) \) are the same as in \( \sigma \), and \( \sigma \) is a complete state, we have that \( \tau(\sigma_0) = \sigma \). Then \( \sigma \) must also be complete, and hence we have \( \sigma \in \mathcal{P}_{\text{M,cb}}^{\text{Ex}} \). This concludes the proof.

\section*{B.2 Proof of Theorem 2 (Validity of cb^{power})}

\textit{Proof (Proof of Theorem 2).} Monotonicity is proven in Lemma 1. Acyclicity is proven in Lemma 2. That \( \text{cb}^0_{\sigma} \leq \text{cb}^{\text{power}}_{\sigma} \) for any state \( \sigma \) follows directly from the definition of \( \text{cb}^{\text{power}} \). \qed
Monotonicity of POWER

Lemma 1. \( \text{cb}^{\text{power}} \) is monotonic w.r.t. \( \text{M}^{\text{POWER}} \).

Proof (Proof of Lemma[7]). Assume that \( \text{cb} = \text{cb}^{\text{power}} \) and \( \text{M} = \text{M}^{\text{POWER}} \). Let \( \sigma = (\lambda, F, E, \text{po}, \text{co}, \text{rf}) \) and \( \sigma' = (\lambda', F', E', \text{po}', \text{co}', \text{rf}') \) be two states such that \( \sigma \leq_{\text{cb}} \sigma' \).

We prove first condition (i): that if \( \text{M}(\sigma') \) then \( \text{M}(\sigma) \). To see this, we need to study the definition of the POWER axiomatic memory models as given in [7]. We see that an execution is allowed by the axiomatic memory model, unless it contains certain cycles in the relations between events. All such forbidden cycles are constructed from an execution is allowed by the axiomatic memory model, unless it contains certain cycles in the relations between events. All such forbidden cycles are constructed from some combination of the following relations: \text{po-loc}, \text{co}, \text{rf}, \text{addr}, \text{data}, \text{fre}, \text{rfe}, \text{rfl}, \text{ctrl} + \text{isync}, \text{co}, \text{ctrl}, \text{addr}; \text{po}, \text{sync}, \text{lwsync}. The construction of the forbidden cycles is such that adding more relations between events can never cause a forbidden cycle to disappear. Studying these relations one by one, we see that for each of them, the relation in \( \sigma \) is a subset of the relation in \( \sigma' \). We discuss here only one of the more interesting cases: \text{po-loc}. Consider two events \( e \) and \( e' \) which are committed in \( \sigma \), and where \( (e, e') \in \text{po-loc}_\sigma \). The same events must also be committed in \( \sigma' \), and be ordered in the same way in program order in \( \sigma' \) as in \( \sigma \). Therefore we must argue that \( e \) and \( e' \) both access the same memory location in \( \sigma' \) as in \( \sigma \). This follows from the fact that the set of committed events \( E \) in \( \sigma \) is \( \text{cb}_{\sigma'} \)-closed. Since \( \text{cb}^{\text{power}} \) contains all three of \text{addr}, \text{data} and \text{rf}, the computation of the address in \( e \) and \( e' \) must produce the same value in \( \sigma' \) as in \( \sigma \). Hence we have \( (e, e') \in \text{po-loc}_{\sigma'} \). Since all of the relations participating in forbidden cycles in \( \sigma \) are subsets of the corresponding relations in \( \sigma' \), we know that any forbidden cycle in \( \sigma \) must also be in \( \sigma' \). Therefore \( \neg \text{M}(\sigma) \Rightarrow \neg \text{M}(\sigma') \). The contrapositive gives us condition (i).

We turn now to condition (ii): that \( \text{cb}_{\sigma} \subseteq \text{cb}_{\sigma'} \). We will show that any edge in \( \text{cb}_{\sigma} \) is also in \( \text{cb}_{\sigma'} \). From the definition of \( \text{cb}^{\text{power}} \), we know that \( \text{cb}_{\sigma} \) is the transitive irreflexive closure of the union of the following relations: \text{addr}, \text{data}, \text{ctrl}, \text{rf}, \text{addr}; \text{po}, \text{po-loc}, \text{sync}, \text{lwsync}. We will consider an arbitrary edge \( (e, e') \in \text{cb}_{\sigma} \) which is in one of those relations, and show that \( (e, e') \) is also in the corresponding relation in \( \sigma' \). If \( (e, e') \) is in \text{addr} or \text{data}, then \( e' \) uses the value in a register provided by the program order-earlier event \( e \). We know that in the extended state \( \sigma' \), the same relation persists. This is because any new event \( e'' \) which might appear in \( \sigma' \) and which breaks the data-flow from \( e \) to \( e' \) must be between \( e \) and \( e' \) in program order. This would contradict the assumption that \( F \) is a \text{po}'-closed subset of \( F' \). The case when \( (e, e') \in \text{addr}; \text{po} \) follows similarly. The case \( (e, e') \in \text{po-loc}_\sigma \) was covered in the proof for condition (i) above. In all of the remaining cases, \text{ctrl}, \text{lwsync}, \text{sync}, there is some event \( e'' \) (a branch or some fence) which comes between \( e \) and \( e' \) in program order in \( \sigma \). Since we have \( F \subseteq F' \) and \( \text{po} = \text{po}' \), the same event must also appear in \( \sigma \), and cause the same relation between \( e \) and \( e' \).

Finally we turn to proving condition (iii): that for all \( e \in F \) such that either \( \text{enabled}_{\sigma}(e) \) and \( e \not\in E' \) or \( e \in E \), we have \( (e', e) \in \text{cb}_{\sigma} \Rightarrow (e', e) \in \text{cb}_{\sigma'} \) for all \( e' \in F' \). We have already shown that \( \text{cb}_{\sigma} \subseteq \text{cb}_{\sigma'} \). So we have \( (e', e) \in \text{cb}_{\sigma} \Rightarrow (e', e) \in \text{cb}_{\sigma'} \), for all \( e, e' \in F' \). It remains to show that for any event \( e \) which is either enabled or committed in \( \sigma \), and which does not become committed when extending \( e \) to \( e' \), we have \( (e', e) \in \text{cb}_{\sigma} \Rightarrow (e', e) \in \text{cb}_{\sigma} \) for all \( e' \in F' \), i.e., that no additional incoming cb edges to \( e \) appear in \( \sigma' \) which are not in \( \sigma \). Let \( e \) be such an event. If any new incoming...
Lemma 2. For any state \( \sigma \) assume that a state \( \sigma \) is a model is defined in the way described in the Herding Cats paper \([7]\). We will assume \( \sigma \) is a \( \text{cb} \) model of \( \sigma \). From the definition of the commit-before functions, we see that \( \text{cb} \) is acyclic in states that are allowed under \( \text{POWER} \). This concludes the proof.

Acyclicity Proof for \( \text{cb}^0 \) and \( \text{cb}^\text{power} \) under \( \text{POWER} \). In the following we prove that the commit-before functions \( \text{cb}^0 \) and \( \text{cb}^\text{power} \) are acyclic in states that are allowed under \( \text{POWER} \).

**Lemma 2.** For any state \( \sigma \) such that \( M^{\text{POWER}}(\text{exec}(\sigma)) \), the relations \( \text{cb}^0_\sigma \) and \( \text{cb}^\text{power}_\sigma \) are acyclic.

From the definition of the commit-before functions, we see that \( \text{cb}^\text{power} \) is the strongest one. It is sufficient to only prove the acyclicity of \( \text{cb}^\text{power} \) w.r.t. to \( \text{POWER} \). Here we assume that the \( \text{POWER} \) memory model is defined in the way described in the Herding Cats paper \([7]\). We will assume that the reader is familiar with the notations and definitions used in \([7]\).

Define the sets \( R, W, M \subseteq E \) as the sets of load events, store events and memory accesses events respectively:

\[
R = \{ e \in E | \exists r : a = [a] \} \text{ and } W = \{ e \in E | \exists a, a'.\text{instr}(e) = [a] := a' \} \text{ and } M = R \cup W. \text{ Define } RR = R \times R. \text{ Define } RW, RM, WR, WM, WM, MR, MW, MM similarly.
\]

The proof of the acyclicity of \( \text{cb}^\text{power} \) w.r.t. to \( \text{POWER} \) is done by contradiction. Let us assume that a state \( \sigma = (F, E, po, co, rf) \) such that \( M(\text{exec}(\sigma)) \), with \( M = M^{\text{POWER}} \), holds and acyclic(\( \text{cb}^\text{power}_\sigma \)) does not. This implies that there is a sequence of events \( e_0, e_1, \ldots, e_n \in F \) such that \( (e_0, e_1), (e_1, e_2), \ldots, (e_{n-1}, e_n), (e_n, e_0) \in \text{cb}^\text{power} \) is a cycle. Let \( rfe = \{ (e, e') \in rf | \text{tid}(e) \neq \text{tid}(e') \} \), \( rfi = rf \setminus rfe \), \( dp = \text{addr}_\sigma \cup \text{data}_\sigma \), \( cc_0 = dp \cup \text{ctrl}_\sigma \cup (\text{addr}_\sigma \cup \text{po-loc}_\sigma) \cup \text{fences} \), and \( rfic = \text{sync}_\sigma \cup \text{lwsync}_\sigma \). First, we will show the acyclicity of the relation \( cc_0 \cup fences \cup rfi \).

**Lemma 3.** The relation \( cc_0 \cup fences \cup rfi \) is acyclic.
Proof. This is an immediate consequence of the fact that \( po \) is acyclic by definition and \( cc_0 \cup \text{fences} \cup \text{rfi} \subseteq po \). \qed

Since \( \text{rfe} \) is the only relation in the definition of \( \text{cb}^\text{power} \) which relates events of different threads, the cycle should contains at least two events belonging to two different threads and related by \( \text{rfe} \) (otherwise, we will have a cycle in \( \text{cb}^\text{power} \setminus \text{rfe} = cc_0 \cup \text{fences} \cup \text{rfi} \) contains a cycle and this contradicts Lemma 3). We assume w.l.o.g. that \( \text{tid}(e_0) \neq \text{tid}(e_n) \). Since \((e_n,e_0) \in \text{rfe} \), we have that \( e_0 \in R \) and \( e_n \in W \). This implies \((e_0,e_1) \not\in \text{rfe} \).

Let \( i_1, i_2, \ldots, i_k \in \{0, \ldots, n\} \) be the maximal sequence of indices such that for every \( j \in \{1, \ldots, k\} \), we have \((e_{i_j},e_{(i_j+1)\mod (n+1)}) \in \text{rfe} \). Let \( i_0 = -1 \). For every \( j \in \{1, \ldots, k\} \), we have \( e_{i_j} \in W \cap E \) and \( e_{i_j+1} \in R \cap E \). In the following, we will show that \((e_{i_j+1},e_{i_j}) \in \text{ppo} \cup \text{fences} \) for all \( j \in \{1, \ldots, k\} \). (Observe that \( \text{ppo} \) is defined as in (11).) This can be seen as an immediate consequence of Lemma 4 since \((e_{i_1+1},e_{i_1}) \in (cc_0 \cup \text{fences} \cup \text{rfi})^* \cap RW \) by definition. This implies that the sequence of events \( e_0, e_1, \ldots, e_k \) forms a cycle in \( hb = ppo \cup \text{fences} \cup \text{rfe} \). Furthermore \((e_0,e_1, \ldots, e_k) \) are events in \( \text{exec}(\sigma) \). This contradicts the POWER axiom "NO THIN AIR" which requires the acyclicity of \( hb \) in order that \( M(\text{exec}(\sigma)) \) holds. The rest of the proof is dedicated to the proof of the following lemma under the POWER memory model:

**Lemma 4.** \((cc_0 \cup \text{fences} \cup \text{rfi})^* \cap RW \subseteq \text{ppo} \cup \text{fences} \).

**Proof.** Assume two events \( e, e' \in E \) such that \((e,e') \in (cc_0 \cup \text{fences} \cup \text{rfi})^* \cap RW \). We will show that \((e,e') \in \text{ppo} \cup \text{fences} \).

Since \((e,e') \in (cc_0 \cup \text{fences} \cup \text{rfi})^* \cap RW \) then there is a sequence of events \( e_0, e_1, \ldots, e_n \in E \) such that \( e_0 = e, e_n = e' \), and \((e_{i-1},e_{i}) \in (cc_0 \cup \text{fences} \cup \text{rfi}) \) for all \( i \in \{1, \ldots, n\} \).

Let us assume first that there is some \( i \in \{1, \ldots, n\} \) such that \((e_{i-1},e_{i}) \in \text{fences} \). Then, there is some fence (sync or lwsync) which is program order between \( e_{i-1} \) and \( e_i \). Since \( cc_0 \cup \text{fences} \cup \text{rfi} \subseteq po \) it also holds that the fence is program order between \( e_0 \) and \( e_n \). Since we know that \((e_0,e_n) \in RW \), we have \((e_0,e_n) \in \text{fences} \). Thus we conclude that \((e,e') \) is in \( \text{ppo} \cup \text{fence} \).

Let us assume now that there is no \( i \in \{1, \ldots, n\} \) such that \((e_{i-1},e_{i}) \in \text{fences} \). We will show that for each \( i \in \{1, \ldots, n\} \), we have \((e_{i-1},e_{i}) \in \text{cc} \). First observe that \( \text{rfi} \subseteq \text{po-loc}_c \subseteq cc_0 \). Then, \((e_{i-1},e_{i}) \in \text{cc} \) trivially holds since \((e_{i-1},e_{i}) \in cc_0 \cup \text{rfi} \subseteq cc_0 \) and by definition of \( cc \), we have \( cc_0 \subseteq cc \). Since \( cc \) is transitive by definition we have \((e,e') \in cc \). Finally, from the definition of \( ppo \), we have \( cc \subseteq ic \) and \((ic \cap RW) \subseteq ppo \). This implies that \((e,e') \in \text{ppo} \) since \((e,e') \in RW \). \qed

This concludes the proof of Lemma 2.

**B.3 Proof of Theorem 3 (Deadlock Freedom of POWER)**

We prove here that our operational semantics instantiated with \( M = M^\text{POWER} \) and \( \text{cb} = \text{cb}^\text{power} \) never deadlocks.
We recall what needs to be proven: Assume that $M = M^{\text{POWER}}$ and $\text{cb} = \text{cb}^{\text{power}}$.

Let $\tau$ be a run from $\sigma_0$, with $\sigma_\tau = \tau(\sigma_0)$. Assume that $e$ is a memory access event (load or store) such that $\text{enabled}_\tau(e)$. There exists a parameter $p$ such that $\tau.e[p]$ is a run from $\sigma_0$ with $\sigma'_\tau = \tau.e[p](\sigma_0)$.

**Proof (Proof of Theorem 3).** If $e$ is a store, then let $p$ be the number of committed stores in $\sigma_\tau$ to the same address as $e$, so that $e$ becomes co-last in $\sigma'_\tau$. If $e$ is a load, then let $p$ be the co-last store to address $\sigma_\tau(e)$.

We will now investigate the new edges in various inter-event relations in $\sigma'_\tau$. Let $\pi_\tau = (E_{\pi_\tau}, \text{po}_{\pi_\tau}, \text{co}_{\pi_\tau}, \text{rf}_{\pi_\tau}) = \text{exec}(\sigma_\tau)$ and $\pi'_\tau = (E'_{\pi_\tau}, \text{po}'_{\pi_\tau}, \text{co}'_{\pi_\tau}, \text{rf}'_{\pi_\tau}) = \text{exec}(\sigma'_\tau)$.

**E:** We have $E'_{\pi_\tau} = E_{\pi_\tau} \cup \{e\}$.

| Event | Update |
|-------|--------|
| po-loc | $\text{po-loc}_{\pi'_\tau} \subseteq \text{po-loc}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| co | $\text{co}_{\pi'_\tau} \subseteq \text{co}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| fr | $\text{fr}_{\pi'_\tau} \subseteq \text{fr}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| fre | $\text{fre}_{\pi'_\tau} \subseteq \text{fre}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| rf | $\text{rf}_{\pi'_\tau} \subseteq \text{rf}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| rfi | $\text{rfi}_{\pi'_\tau} \subseteq \text{rfi}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| rfe | $\text{rfe}_{\pi'_\tau} \subseteq \text{rfe}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| com | $\text{com}_{\pi'_\tau} \subseteq \text{com}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |
| fences | $\text{fences}_{\pi'_\tau} \subseteq \text{fences}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$ |

Since, for each new edge $e_0 \xrightarrow{\text{fences}_{\pi_\tau}} e_1$ with $(e_0, e_1) \notin \text{fences}_{\pi_\tau}$, we must have $e_0 = e$ or $e_1 = e$. Furthermore, we cannot have $e \xrightarrow{\text{fences}_{\pi_\tau}} e_1$, since $\text{fences}_{\pi'_\tau} \subseteq \text{cb}^{\text{power}}_{\pi'_\tau}$, and $e$ is $\text{cb}^{\text{power}}_{\pi'_\tau}$-last.

**fence:** We have $\text{fence}_{\pi'_\tau} \subseteq \text{fence}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences.

**dp:** We have $\text{dp}_{\pi'_\tau} \subseteq \text{dp}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences.

**rdw:** We have $\text{rdw}_{\pi'_\tau} \subseteq \text{rdw}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences, where we notice that $\text{rdw}_{\pi'_\tau} \subseteq \text{po-loc}_{\pi'_\tau} \subseteq \text{cb}^{\text{power}}_{\pi'_\tau}$.

**ctrl+cfence:** We have $\text{ctrl+cfence}_{\pi'_\tau} \subseteq \text{ctrl+cfence}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences, where we notice that $\text{ctrl+cfence}_{\pi'_\tau} \subseteq \text{ctrl}_{\pi'_\tau} \subseteq \text{cb}^{\text{power}}_{\pi'_\tau}$.

**detour:** We have $\text{detour}_{\pi'_\tau} \subseteq \text{detour}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences, where we notice that $\text{detour}_{\pi'_\tau} \subseteq \text{po-loc}_{\pi'_\tau} \subseteq \text{cb}^{\text{power}}_{\pi'_\tau}$.

**po-loc:** We have $\text{po-loc}_{\pi'_\tau} \subseteq \text{po-loc}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences.

**ctrl:** We have $\text{ctrl}_{\pi'_\tau} \subseteq \text{ctrl}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences.

**addr:po:** We have $\text{addr}_{\pi'_\tau} \subseteq \text{addr}_{\pi_\tau} \cup \{(e', e) | e' \in E\}$

Same motivation as for fences.
ppo: We have $\text{ppo}_{\pi'} \subseteq \text{ppo}_{\pi_0} \cup \{(e', e) | e' \in E\}$

To see this, let $(e_0, e_1)$ be any edge in $\text{ppo}_{\pi_0} \setminus \text{ppo}_{\pi_0}$. From the definition of ppo, we have that $e_0 \overrightarrow{\pi_0 \pi_0} e_1$. Furthermore there must be at least one edge along that chain which is not in $\text{rfe}_{\pi_0} \cup \text{cc}_{\pi_0} \pi_0$. We have seen above that any edge which is in $\text{rfe}_{\pi_0} \cup \text{cc}_{\pi_0} \pi_0$, but not in $\text{rfe}_{\pi_0} \cup \text{cc}_{\pi_0} \pi_0$, must be of the form $(e_2, e)$ for some $e_2$. Hence we have

$$
\begin{array}{c}
(e_0, e_1) \\
\rightarrow \\
(e_2, e) \\
\rightarrow \\
(e_2, e) \\
\rightarrow \\
e_0 \overrightarrow{\pi_0 \pi_0} e_1.
\end{array}
$$

But there is no edge going from $e$ in $\text{rfe}_{\pi_0} \cup \text{cc}_{\pi_0} \pi_0$, so it must be the case that $e = e_1$.

hb: We have $\text{hb}_{\pi'} \subseteq \text{hb}_{\pi_0} \cup \{(e', e) | e' \in E\}$.

By the above, we have

$$
\text{hb}_{\pi'} = \text{fences}_{\pi_0} \cup \text{rfe}_{\pi_0} \cup \text{ppo}_{\pi_0} \subseteq \text{fences}_{\pi_0} \cup \text{rfe}_{\pi_0} \cup \text{ppo}_{\pi_0} \cup \{(e', e) | e' \in E\} = \text{hb}_{\pi_0} \cup \{(e', e) | e' \in E\}.
$$

Notice that none of the $\pi_0$-relations contain links to or from $e$, since $e$ is not in $E_{\pi_0}$.

We will now show that $\text{POWER}(\pi')$. We need to show that each of the four $\text{POWER}$ axioms [7] holds for $\pi'$.

**Subproof 1:** Show $\text{acyclic}(\text{po-loc}_{\pi'} \cup \text{com}_{\pi'})$

By the assumption $\text{POWER}(\pi_0)$, we have $\text{acyclic}(\text{po-loc}_{\pi_0} \cup \text{po-loc}_{\pi_0})$. Since $(\text{po-loc}_{\pi_0} \cup \text{po-loc}_{\pi_0}) \setminus (\text{po-loc}_{\pi_0} \cup \text{po-loc}_{\pi_0})$ only contains edges leading to $e$, and $\text{po-loc}_{\pi_0} \cup \text{po-loc}_{\pi_0}$ contains no edges leading from $e$, we also have $\text{acyclic}(\text{po-loc}_{\pi_0} \cup \text{po-loc}_{\pi_0})$.

**Subproof 2:** Show $\text{acyclic}(\text{hb}_{\pi'})$

The proof is analogue to that in Subproof 1.

**Subproof 3:** Show $\text{irreflexive}(\text{fre}_{\pi'} \cup \text{prop}_{\pi'} \cup \text{hb}_{\pi'} \cup \text{com}_{\pi'})$

By the assumption $\text{POWER}(\pi_0)$, we have $\text{irreflexive}(\text{fre}_{\pi_0} \cup \text{prop}_{\pi_0} \cup \text{hb}_{\pi_0} \cup \text{com}_{\pi_0})$. Assume for a contradiction that we have $(e_0, e_0) \in \text{fre}_{\pi_0} \cup \text{prop}_{\pi_0} \cup \text{hb}_{\pi_0} \cup \text{com}_{\pi_0}$. By examining the definition of prop, we see that every edge building up the chain from $e_0$ to $e_0$ through $\text{fre}_{\pi_0} \cup \text{prop}_{\pi_0} \cup \text{hb}_{\pi_0} \cup \text{com}_{\pi_0}$, must be in one of $\text{fre}_{\pi_0} \cup \text{prop}_{\pi_0} \cup \text{hb}_{\pi_0} \cup \text{com}_{\pi_0}$ or $\text{fence}_{\pi_0}$. Since at least one edge must not be in the corresponding $\pi_0$-relation, the chain of relations must go through $e$. But we have seen above that there is no edge going out from $e$ in any of the above mentioned relations. Hence there can be no such cycle.
Subproof 4: Show acyclic($\text{co}_{\tau^r} \cup \text{prop}_{\tau^r}$)

The proof is analogue to that in Subproof 3.
This concludes the proof. □

C Proofs for Section 3

Here we provide proofs for the theorems appearing in Section 3.

C.1 Proof of Theorem 4 (Soundness of RSMC)

Lemma 5. Assume that cb is valid w.r.t. M, and that M and cb are deadlock free. Assume that $\tau_A = \tau.\epsilon_r[\epsilon_w[\epsilon'_w]]$ is a run from $\sigma_0$. Further assume that $\text{enabled}_{\tau(\sigma_0)}(\epsilon_r)$. Then there is a run $\tau_B = \tau.\epsilon_r[\epsilon'_w]$ from $\sigma_0$ such that $\text{enabled}_{\tau_B(\sigma_0)}(\epsilon_r)$.

Proof (Proof of Lemma 5). The lemma follows from deadlock freedom and monotonicity. □

Lemma 6. Assume that cb is valid w.r.t. M, and that M and cb are deadlock free. Assume that $\tau_A = \tau.\epsilon_r[\epsilon_w[\epsilon'_w]]$ is a run from $\sigma_0$. Assume that $\tau_B = \tau.\epsilon_r[\epsilon_w[\epsilon'_w]]$ is a run from $\sigma_0$. Let $\sigma_B = \tau_B(\sigma_0)$. Then either

- $\tau_B = \tau.\epsilon_r[\epsilon_w[\epsilon'_w]], \tau'$ is a run from $\sigma_0$ and $\text{enabled}_{\tau(\sigma_0)}(\epsilon_w)$, or
- there is an event $\epsilon'_w \in \tau'$ such that $\tau_D = \tau.\epsilon_r[\epsilon'_w[\epsilon'_w]], \tau''$ is a run from $\sigma_0$ with $\tau''.\epsilon'_w[\epsilon'_w], \tau''' = \tau'$ and $\text{enabled}_{\tau_D(\sigma_0)}(\epsilon_w)$.

Proof (Proof of Lemma 6). Monotonicity of cb and Lemma 5 give that there exists a run $\tau_B' = \tau.\epsilon_r[p]$ from $\sigma_0$ for some store $p$ with $\text{enabled}_{\tau_B(\sigma_0)}(\epsilon_w)$. We case split on whether or not $p \in \tau'$:

Assume first that $p \notin \tau'$. Then since $\text{enabled}_{\tau_B(\sigma_0)}(\epsilon_r)$, we know that $\epsilon_r$ is not cb-related to any event in $\tau'$. Therefore, we can rearrange $\tau_B'$ into $\tau_C = \tau.\epsilon_r[p], \tau'$, which is a run from $\sigma_0$. Furthermore, since the committed events are the same in $\tau_C$ as in $\tau_B$, we have $\text{enabled}_{\tau_C(\sigma_0)}(\epsilon_w)$.

Assume instead that $p \in \tau'$. Then $\tau_B' = \tau.\epsilon_r[p[\epsilon'_w]], \tau''$, $\epsilon_r[\epsilon'_w]$ for some $\tau''$, $\epsilon'_w$, $\tau'''$ and $\epsilon'_w = p$. Then by the same argument as above, we can reorder this run to form $\tau_D = \tau.\epsilon_r[\epsilon'_w[\epsilon'_w]], \tau''$, $\epsilon_r[\epsilon'_w], \tau'''$ which is a run from $\sigma_0$ and $\tau''.\epsilon'_w[\epsilon'_w], \tau''' = \tau'$ and $\text{enabled}_{\tau_D(\sigma_0)}(\epsilon_w)$. □

We are now ready to state and prove the main lemma, from which the soundness theorem directly follows:

Lemma 7. Assume that cb is valid w.r.t. M, and that M and cb are deadlock free. Let $\tau, \sigma, \pi$ be such that $\tau(\sigma_0) = \sigma$ and $\text{exec}(\sigma) = \pi$ and $M(\pi)$. Then for all $\pi'$ s.t. $M(\pi')$, and $\pi'$ is a complete cb-extension of $\pi$, the evaluation of $\text{Explore}(\tau, \sigma)$ will contain a recursive call to $\text{Explore}(\tau', \sigma')$ for some $\tau', \sigma'$ such that $\text{exec}(\sigma') = \pi'$. 
Proof (Proof of Lemma). By assumption there is an upper bound $B$ on the length of any run of the fixed program. Therefore, we can perform the proof by total induction on $B$ minus the length of $\tau$.

Fix arbitrary $\tau, \sigma = (\lambda, F, E, po, co, rf)$ and $\pi$. We will show that for all $\pi'$ s.t. $\Pi(\pi')$, and $\pi'$ is an complete cb-extension of $\pi$, the evaluation of $\text{Explore}(\tau, \sigma)$ will contain a recursive call to $\text{Explore}(\tau', \sigma')$ for some $\tau', \sigma'$ such that $\text{exec}(\sigma') = \pi'$. Our inductive hypothesis states that for all $\tau'', \sigma''$, $\pi''$ such that $|\tau| < |\tau''|$ and $\tau''(\sigma_0) = \sigma''$ and $\text{exec}(\sigma'') = \pi''$ and $\Pi(\pi'')$, it holds that for all $\pi'''$ s.t. $\Pi(\pi'''')$, and $\pi'''$ is a complete cb-extension of $\pi''$, the evaluation of $\text{Explore}(\tau'', \sigma'')$ will contain a recursive call to $\text{Explore}(\tau''', \sigma''')$ for some $\tau''', \sigma'''$ such that $\text{exec}(\sigma''') = \pi'''$.

Consider the evaluation of $\text{Explore}(\tau, \sigma)$. If there is no enabled event on line 2, then $\sigma$ is complete, and there are no (non-trivial) cb-extensions of $\pi$. The Lemma is then trivially satisfied. Assume therefore instead that a enabled event $e$ is selected on line 2.

We notice first that for all executions $\pi' = (E', po', co', rf')$ s.t. $\Pi(\pi')$, and $\pi'$ is a complete cb-extension of $\pi$, it must be the case that $e \in E'$. This is because $\pi'$ is an extension of $\pi$, and $e$ is enabled after $\pi$ and $\pi'$ is complete. Since $e$ is enabled in $\pi$, it must be executed at some point before the execution can become complete.

The event $e$ is either a store or a load. Assume first that $e$ is a store. Let $\pi'' = (E', po', co', rf')$ be an arbitrary execution s.t. $\Pi(\pi')$, and $\pi''$ is a complete cb-extension of $\pi$. There exists some parameter (natural number) $n$ for $e$ which inserts $e$ in the same position in the coherence order relative to the other stores in $E$ as $e$ has in $\pi''$. Let $\sigma_n = \tau.e[n]\sigma_n$ and $\pi_n = \text{exec}(\sigma_n)$. Notice that $\pi'$ is an extension of $\pi_n$. We have $\Pi(\pi')$, and by the monotonicity of the memory model we then also have that the parameter $n$ is allowed for the event $e$ by the memory model, i.e., $\Pi(\pi_n)$. Since the parameter $n$ is allowed for $e$ in $\sigma$ by the memory model, $(n, \sigma_n)$ will be in $S$ on line 4 and so there will be a recursive call $\text{Explore}(\tau.e[n], \sigma_n)$ on line 6. The run $\tau.e[n]$ is a longer run than $\tau$. Hence the inductive hypothesis can be applied, and yields that the sought recursive call $\text{Explore}(\tau', \sigma')$ will be made. This concludes the case when $e$ is a store.

Next assume instead that $e$ is a load. Let $\mathcal{E}$ be the set of all executions $\pi'$ s.t. $\Pi(\pi')$, and $\pi'$ is a complete cb-extension of $\pi$. Now define the set $\mathcal{E}_0 \subseteq \mathcal{E}$ s.t. $\mathcal{E}_0$ contains precisely the executions $\pi' = (E', po', co', rf') \in \mathcal{E}$ where $(e_w, e) \in rf'$ for some $e_w \in E$. I.e., we define $\mathcal{E}_0$ to be the executions in $\mathcal{E}$ where the read-from source for $e$ is already committed in $\pi$. Let $\mathcal{E}_1 = \mathcal{E} \setminus \mathcal{E}_0$. We will show that the lemma holds, first for all executions in $\mathcal{E}_0$, and then for all executions in $\mathcal{E}_1$.

Let $\pi'$ be an arbitrary execution in $\mathcal{E}_0$. We can now apply a reasoning analogue to the reasoning for the case when $e$ is a store to show that there will be a recursive call $\text{Explore}(\tau', \sigma')$ with $\text{exec}(\sigma') = \pi'$.

We consider instead the executions in $\mathcal{E}_1$. Assume for a proof by contradiction that there are some executions in $\mathcal{E}_1$ that will not be explored. Let $\mathcal{E}_2 \subseteq \mathcal{E}_1$ be the set of executions in $\mathcal{E}_1$ that we fail to explore. I.e., let $\mathcal{E}_2 \subseteq \mathcal{E}_1$ be the set of executions $\pi'$ such that there are no $\tau'$, $\sigma'$ where $\tau'(\sigma_0) = \sigma'$ and $\text{exec}(\sigma') = \pi'$ and there is a recursive call $\text{Explore}(\tau', \sigma')$ made during the evaluation of $\text{Explore}(\tau, \sigma)$. We will now define a ranking function $R$ over executions in $\mathcal{E}_2$, and then investigate one of the executions that minimize $R$. For a run $\tau'$ of the form $\tau.\tau_0.e_w[n_w].e[e_w].\tau_1$ (notice the fixed run $\tau$
and the fixed load event $e$, define $R(\tau') = |\tau_0|$. For an execution $\pi' \in \mathfrak{E}_2$, let $T(\pi')$ be the set of runs $\tau'$ of the form $\tau.\tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}. \tau_1$ such that $\text{exec}(\tau'(\sigma_0)) = \pi'$. Now define $R(\tau') = R(\pi')$ for a run $\tau' \in T(\pi')$ that minimizes $R$ within $T(\pi')$.

Now let $\tau' \in \mathfrak{E}_2$ be an execution minimizing $R$ in $\mathfrak{E}_2$ and $\tau' = \tau.\tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}. \tau_1$ be a run in $T(\pi')$ minimizing $R$ in $T(\pi')$. Let $\sigma' = \tau'(\sigma_0)$. Lemma 6 tells us that either

- $\tau_e.\tau_0 \tau_0$ is a run from $\sigma_0$ and $\text{enabled}_{\tau_C(\sigma_0)}(e_{\lceil w \rceil})$, or
- there is an event $e_{\lceil w \rceil}' \in \tau_0$ such that $\tau_D = \tau.\tau'.e_{\lceil n \rceil . e_{\lceil w \rceil}. \tau''$ is a run from $\sigma_0$ and $\tau'.\tau'' | \tau_0 = \tau_0$ and $\text{enabled}_{\tau_D(\sigma_0)}(e_{\lceil w \rceil})$.

Consider first the case when $\tau_C = \tau.e_{\lceil w \rceil eradicate \tau_0}$ is a run from $\sigma_0$ and $\text{enabled}_{\tau_C(\sigma_0)}(e_{\lceil w \rceil})$. Since $\tau_C$ is a run, we know that $e_{\lceil w \rceil}$ is an allowed parameter for $e$ after $\tau$. So $(e_{\lceil w \rceil}, \sigma_{\lceil w \rceil})$ will be added to $S$ for some $\sigma_{\lceil w \rceil}$ on line 11 in the call $\text{Explore}(\tau, \sigma)$, and a recursive call $\text{Explore}(\tau.e_{\lceil w \rceil}^1, \sigma_{\lceil w \rceil})$ will be made on line 13. Since $\tau.e_{\lceil w \rceil}$ is a longer run than $\tau$, the inductive hypothesis tells us that all complete cb-extensions of $\text{exec}(\sigma_{\lceil w \rceil})$ will be explored. Let $\pi_C'$ be any complete cb-extension of $\text{exec}(\tau_C(\sigma_0))$. Notice that $\pi_C'$ is also a complete cb-extension of $\text{exec}(\sigma_{\lceil w \rceil})$. Since we have $\text{enabled}_{\tau_C(\sigma_0)}(e_{\lceil w \rceil})$ it must be the case that $e_{\lceil w \rceil}$ is committed in $\pi_C'$. Then the race detection code in $\text{DetectRace}$ is executed for $e_{\lceil w \rceil}$ on line 8 in the call to $\text{Explore}$ where $e_{\lceil w \rceil}$ is committed. When the race detection code is run for $e_{\lceil w \rceil}$, the $R \rightarrow W$ race from $e$ to $e_{\lceil w \rceil}$ will be detected. To see this, notice that $\pi_C'$ is an extension of $\text{exec}(\tau_C(\sigma_0))$.

Hence we know that the events that are cb-before $e_{\lceil w \rceil}$ in $\pi_C'$ are the same as the events that are cb-before $e_{\lceil w \rceil}$ in $\tau_C$ and also in $\tau'$. Therefore $e_{\lceil w \rceil}$ targets the same memory location in $\pi_C'$ as it does in $\tau'$. Furthermore, $e$ cannot be cb-before $e_{\lceil w \rceil}$, since $e$ appears after $e_{\lceil w \rceil}$ in $\tau'$. Therefore, the race from $e$ to $e_{\lceil w \rceil}$ is detected, and the branch $\tau_0.e_{\lceil w \rceil}^1.e_{\lceil w \rceil}$ is added to $Q[e]$. When the lines 13-19 are executed in the call $\text{Explore}(\tau, \sigma)$, that branch will be traversed. During the traversal, all parameters for $e_{\lceil w \rceil}$ will be explored, and in particular the following call will be made: $\text{Explore}(\tau.\tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}, \sigma_{\lceil w \rceil})$ for some $\sigma_{\lceil w \rceil}$. Since the run $\tau.\tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}$ is longer than $\tau$, the inductive hypothesis tells us that all its complete cb-extensions will be explored. These include $\pi''$, contradicting our assumption that $\pi''$ is never explored.

Consider then the case when there is an event $e_{\lceil w \rceil}' \in \tau_0$ such that $\tau_D = \tau.\tau'.e_{\lceil n \rceil . e_{\lceil w \rceil}. \tau''$ is a run from $\sigma_0$ and $\tau'.\tau'' | \tau_0 = \tau_0$ and $\text{enabled}_{\tau_D(\sigma_0)}(e_{\lceil w \rceil})$. Let $\pi_D'$ be any execution which is a complete cb-extension of $\text{exec}(\tau_D(\sigma_0))$. Since $\pi_D'$ can be reached by a run which extends $\tau_D$, it must be the case that $\pi_D'$ has a lower rank than $\pi''$, i.e., $R(\pi_D') < R(\pi'')$. Since, $\pi''$ has a minimal rank in $\mathfrak{E}_2$, it must be the case that $\pi_D' \in \mathfrak{E}_2$, and so we know that $\pi_D'$ is explored by some call $\text{Explore}(\tau_D, \sigma_D')$, made recursively from $\text{Explore}(\tau, \sigma)$, with $\tau_D'(\sigma_0) = \sigma_D'$ and $\text{exec}(\sigma_D') = \pi_D'$. By a reasoning analogue to that in the previous case, the store $e_{\lceil w \rceil}$ must be committed in $\pi_D'$, and the $R \rightarrow W$ race from $e$ to $e_{\lceil w \rceil}$ is detected by $\text{DetectRace}$. Again, the branch $\tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}$ is added to $Q[e]$. So there will be a recursive call $\text{Explore}(\tau_w, \sigma_w)$ for $\tau_w = \tau_0.e_{\lceil n \rceil . e_{\lceil w \rceil}$ and $\tau_w(\sigma_0) = \sigma_w$ where the parameter $*$ for $e_{\lceil w \rceil}$ has been instantiated with $n_{\lceil w \rceil}$. Since $\tau_w$ is a longer run than $\tau$, the inductive hypothesis tells us that all extensions of $\tau_w$ will be explored. The sought execution $\pi''$ is an extension of $\text{exec}(\tau_w(\sigma_0))$, and so it will be explored. This again contradicts our assumption that $\pi''$ is never explored. This concludes the proof. \qed
Proof (Proof of Theorem 4). The theorem follows directly from Lemma 7, since each complete execution a cb-extension of exec(σ₀).

C.2 Proof of Theorem 5 (Optimality of RSMC for POWER)

Lemma 8. Assume that M = M^POWER and cb = cb^power. Let τ, and τ.A.e[p_A] and τ.B.e[p_B] be runs. Let σ_A = τ.A.σ₀ and σ_B = τ.B.σ₀. Let τ_a = normalize(cut(τ_A, e, σ_A)) and τ_b = normalize(cut(τ_B, e, σ_B)). Assume τ_a ≠ τ_b.

Then there is some event e' and parameters p_a ≠ p_b such that e'[p_a] ∈ τ_a and e'[p_b] ∈ τ_b.

Proof (Proof of Lemma 8). We start by considering the control flow leading to e in the thread tid(e). Either the control flow to e is the same in both τ_A and τ_B, or it differs. Assume first that the control flow differs. Then there is some branch instruction which program order-precedes e which evaluates differently in τ_A and τ_B. That means that the arithmetic expression which is the condition for a conditional branch evaluates differently in τ_A and τ_B. Our program semantics does not allow data nondeterminism. The only way for an expression to evaluate differently is for it to depend on the value of a program order-earlier load e_l. This loaded value must differ in τ_A and τ_B. This can only happen if e_l gets its value from some chain (possibly empty) of read-from and data dependencies, which starts in a load e'_l (possibly equal to e_l) which takes different parameters in τ_A and τ_B. However, since both read-from and data dependencies are a part of cb^power, this gives us (e'_l, e_l) ∈ cb_{σ_A}^* and (e'_l, e_l) ∈ cb_{σ_B}^*. Furthermore, since e_l provides a value used in a branch that program order-precedes e, we have a control dependency between e_l and e. Control dependencies are also part of cb^power, so we have (e'_l, e) ∈ cb_{σ_A}^* and (e'_l, e) ∈ cb_{σ_B}^*. Then e'_l is the sought witness event.

Now assume instead that the control flow leading to e is the same in τ_A and τ_B.

Consider the sets A = \{e ∈ E|∃p.e[p] ∈ τ_a\} and B = \{e ∈ E|∃p.e[p] ∈ τ_b\}.

Assume first that A = B. Then if all events in A (or equivalently in B) have the same parameters in τ_a as in τ_b, then τ_a and τ_b consist of exactly the same parameterized events. Since all the events have the same parameters in τ_a and τ_b, they must also be related in the same way by the commit-before relation. But then since τ_a and τ_b are normalized, (by the function normalize) it must hold that τ_a = τ_b, which contradicts our assumption τ_a ≠ τ_b. Hence there must be some event which appears in both τ_a and τ_b, which takes different parameters in τ_a and τ_b.

Assume next that A ≠ B. Assume without loss of generality that A contains some event which is not in B. Let C be the largest subset of A such that C ∩ B = ∅.

Then by the construction of τ_a, at least one of the events e_c ∈ C must precede some event e_a ∈ (A ∩ B) ∪ \{e\} in cb_{σ_A}. Since e_c ∉ B we also have (e_c, e_a) ∉ cb_{σ_B}.

We now look to the definition of cb^power, and consider the relation between e_c and e_a in σ_A. From (e_c, e_a) ∈ cb_{σ_A} we know that (e_c, e_a) ∈ addr_{σ_A} ∪ data_{σ_A} ∪ ctrl_{σ_A} ∪ (addr_{σ_A} ∪ po_{σ_A}) ∪ sync_{σ_A} ∪ lwsync_{σ_A} ∪ po-loc_{σ_A} ∪ rf_{σ_A}. However, all the relations addr_{σ_A}, data_{σ_A}, ctrl_{σ_A}, (addr_{σ_A} ∪ po_{σ_A}), sync_{σ_A}, lwsync_{σ_A} are given by the program (and the control flow, which is fixed by assumption). So if (e_c, e_a) ∈ addr_{σ_A} ∪ data_{σ_A} ∪ ctrl_{σ_A} ∪ (addr_{σ_A} ∪ po_{σ_A}) ∪ sync_{σ_A} ∪ lwsync_{σ_A}, then the same relation holds in τ_B, and then we would have (e_c, e_a) ∈ cb_{σ_B} contradicting our previous assumption. Hence it must be
Theorem 5 (Optimality for POWER). Assume that $M = M^{\text{POWER}}$ and $cb = cb^{\text{power}}$. Let $\pi \in [\mathcal{P}]_M^{\text{Ax}}$. Then during the evaluation of a call to $\text{Explore}(\langle \rangle, \sigma_0)$, there will be exactly one call $\text{Explore}(\tau, \sigma)$ such that $\text{exec}(\sigma) = \pi$.

**Proof (Proof of Theorem 5).** It follows from Corollary 1 that at least one call $\text{Explore}(\tau, \sigma)$ such that $\text{exec}(\sigma) = \pi$ is made. It remains to show that at most one such call is made.

Assume for a proof of contradiction that two separate calls $\text{Explore}(\tau_a, \sigma_a)$ and $\text{Explore}(\tau_b, \sigma_b)$ are made such that $\text{exec}(\sigma_a) = \text{exec}(\sigma_b) = \pi$.

Let $\text{Explore}(\tau_c, \sigma_c)$ be the latest call to $\text{Explore}$, which is an ancestor to both of the calls $\text{Explore}(\tau_a, \sigma_a)$ and $\text{Explore}(\tau_b, \sigma_b)$. Since at least two calls were made from $\text{Explore}(\tau_c, \sigma_c)$ to $\text{Explore}$ or $\text{Traverse}$, we know that an event $\tau = \tau_c$ must have been chosen on line 2 in the call $\text{Explore}(\tau_c, \sigma_c)$.

The event $\tau_c$ is a store or a load. Assume first that $\tau_c$ is a store. We see that $\text{Explore}(\tau_c, \sigma_c)$ may make calls to $\text{Explore}$, but not to $\text{Traverse}$. Hence the calls $\text{Explore}(\tau_a, \sigma_a)$ and $\text{Explore}(\tau_b, \sigma_b)$ must be reached from two different calls to $\text{Explore}$ on line 6 in the call $\text{Explore}(\tau_c, \sigma_c)$. However, we see that the different calls made to $\text{Explore}$ from $\text{Explore}(\tau_c, \sigma_c)$ will all fix different parameters for $\tau_c$. Therefore there will be two stores which are ordered differently in the coherence order of the different calls to $\text{Explore}$. As we proceed deeper in the recursive evaluation of $\text{Explore}$, new coherence edges may appear. But coherence edges can never disappear. Therefore it cannot be the case that $\text{exec}(\sigma_a) = \text{exec}(\sigma_b)$, which is the sought contradiction.

Next we assume instead that $\tau_c$ is a load. In this case the calls $\text{Explore}(\tau_a, \sigma_a)$ and $\text{Explore}(\tau_b, \sigma_b)$ will be reached via calls to either $\text{Explore}$ on line 13 or $\text{Traverse}$ on line 18. If both are reached through calls to $\text{Explore}$, then the contradiction is reached in an way analogue to the case when $\tau_c$ is a store above.

Assume instead that the call $\text{Explore}(\tau_a, \sigma_a)$ is reached via a recursive call to $\text{Explore}(\tau_c, \tau_c', \sigma_a, \sigma_d)$ from the call $\text{Explore}(\tau_c, \sigma_c)$, and that $\text{Explore}(\tau_b, \sigma_b)$ is reached via a call to $\text{Traverse}(\tau_c, \sigma_c, \tau_c')$ from the call $\text{Explore}(\tau_c, \sigma_c)$. Notice from the way that subruns in $Q[\epsilon_c]$ are constructed that $\tau_c'$ must have the form
\[ \tau''_c, e'_w[\star], e_c[e'_w] \] for some subrun \( \tau''_c \) and some store \( e'_w \) which is not committed in \( \sigma_c \). In the evaluation of \( \text{Traverse}(\tau_c, \sigma_c, \tau''_c) \), the subrun \( \tau''_c \) will be traversed, and for any call to \( \text{Explore}(\tau_c, \sigma_c) \) made during that evaluation it will hold that \( e_c \) loads from the store \( e'_w \). However, in the call \( \text{Explore}(\tau_c, \sigma_c, e'_w) \) it holds that \( e_c \) loads from \( e_w \). Since \( e_w \) is committed in \( \sigma_c \), it must be that \( e_w \neq e'_w \). And so by the same reasoning as above, we derive the contradiction \( \text{exec}(\sigma_a) \neq \text{exec}(\sigma_b) \).

Next we assume that \( \text{Explore}(\tau_a, \sigma_a) \) and \( \text{Explore}(\tau_b, \sigma_b) \) are reached via calls from \( \text{Explore}(\tau_c, \sigma_c) \) to \( \text{Traverse}(\tau_c, \sigma_c, \tau''_c, \sigma_d) \) and \( \text{Traverse}(\tau_c, \sigma_c, \tau''_c, \sigma_d, \sigma_c, e'_w[\star], e_c[e'_w]) \) respectively. If \( e'_w \neq e'_w \), then the contradiction is derived as above. Assume we have that \( e'_w = e'_w \).

If \( \tau''_c = \tau''_b \), then the entire new branches are equal: \( \tau''_c, e'_w[\star], e_c[e'_w] = \tau''_b, e'_w[\star], e_c[e'_w] \). By the mechanism on lines \( 15 \) to \( 19 \) using the set \( \text{explored} \), we know that the calls \( \text{Traverse}(\tau_c, \sigma_c, \tau''_c, \sigma_d, e'_w[\star], e_c[e'_w]) \) and \( \text{Traverse}(\tau_c, \sigma_c, \tau''_c, \sigma_d, e'_w[\star], e_c[e'_w]) \) must then be the same call, since \( \text{Traverse} \) is not called twice with the same new branch. The call to \( \text{Traverse} \) will eventually call \( \text{Explore} \) after traversing the new branch. From the assumption that the call \( \text{Explore}(\tau_c, \sigma_c) \) is the last one that is an ancestor to both calls \( \text{Explore}(\tau_a, \sigma_a) \) and \( \text{Explore}(\tau_b, \sigma_b) \), we know that the call to \( \text{Traverse} \) must perform two different calls to \( \text{Explore} \), which will eventually lead to the calls \( \text{Explore}(\tau_a, \sigma_a) \) and \( \text{Explore}(\tau_b, \sigma_b) \) respectively. However, when the function \( \text{Traverse} \) traverses a branch, all events have fixed parameters, except the last store (i.e. \( e'_w \) or \( e'_w \)). If different calls to \( \text{Explore} \) from \( \text{Traverse} \) lead to the calls \( \text{Explore}(\tau_a, \sigma_a) \) and \( \text{Explore}(\tau_b, \sigma_b) \), then \( e'_w \) (which is the same as \( e'_w \)) must have different coherence positions in \( \sigma_a \) and \( \sigma_b \). This leads to the usual contradiction.

Hence we know that \( \tau''_c \neq \tau''_b \). From the way new branches are constructed in \( \text{DetectRace} \), we know that the branches \( \tau''_c, e'_w[\star], e_c[e'_w] \) and \( \tau''_c, e'_w[\star], e_c[e'_w] \) must have been constructed and added to \( Q[e_c] \) during the exploration of some continuation of \( \tau_c \). So there must exist runs \( \tau_c, \tau''_c, e'_w[\star], e_c[e'_w] \) and \( \tau_c, \tau''_c, e'_w[\star], e_c[e'_w] \) ending in the states \( \sigma_A \) and \( \sigma_B \) respectively. Furthermore \( \tau''_c \) is the restriction of \( \tau''_c \) to events that precede \( e'_w \) in \( c_b \sigma_A \), and \( \tau''_c \) is the restriction of \( \tau''_c \) to events that precede \( e'_w \) in \( c_b \sigma_B \). From \( \tau''_c \neq \tau''_b \) and Lemma \( 8 \), it now follows that there is an event \( e_d \) which appears both in \( \tau''_c \) and \( \tau''_c \) but which has different parameters in the two subruns. This difference will also be reflected in \( \sigma_a \) and \( \sigma_b \). Hence \( \text{exec}(\sigma_a) \neq \text{exec}(\sigma_b) \), which gives a contradiction and concludes the proof.

\( \square \)