Context-Bounded Analysis for POWER

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Ahmed Bouajjani[2],
and Tuan Phong Ngo[1](✉)

[1] Uppsala University, Uppsala, Sweden
{parosh,mohamed_faouzi.atig,tuan-phong.ngo}@it.uu.se
[2] IRIF, Université Paris Diderot, Paris, France
abou@irif.fr

Abstract. We propose an under-approximate reachability analysis algorithm for programs running under the POWER memory model, in the spirit of the work on context-bounded analysis intitiated by Qadeer et al. in 2005 for detecting bugs in concurrent programs (supposed to be running under the classical SC model). To that end, we first introduce a new notion of context-bounding that is suitable for reasoning about computations under POWER, which generalizes the one defined by Atig et al. in 2011 for the TSO memory model. Then, we provide a polynomial size reduction of the context-bounded state reachability problem under POWER to the same problem under SC: Given an input concurrent program $\mathcal{P}$, our method produces a concurrent program $\mathcal{P}'$ such that, for a fixed number of context switches, running $\mathcal{P}'$ under SC yields the same set of reachable states as running $\mathcal{P}$ under POWER. The generated program $\mathcal{P}'$ contains the same number of processes as $\mathcal{P}$, and operates on the same data domain. By leveraging the standard model checker CBMC, we have implemented a prototype tool and applied it on a set of benchmarks, showing the feasibility of our approach.

1 Introduction

For performance reasons, modern multi-processors may reorder memory access operations. This is due to complex buffering and caching mechanisms that make the response memory queries (load operations) faster, and allow to speed up computations by parallelizing independent operations and computation flows. Therefore, operations may not be visible to all processors at the same time, and they are not necessarily seen in the same order by different processors (when they concern different addresses/variables). The only model where all operations are visible immediately to all processors is the Sequential Consistency (SC) model [28] which corresponds to the standard interleaving semantics where the program order between operations of a same processor is preserved. Modern architectures adopt weaker models (in the sense that they allow more behaviours) due to the relaxation in various ways of the program order. Examples of such weak models are TSO adopted in Intel x86 machines for instance, POWER adopted in PowerPC machines, or the model adopted in ARM machines.

A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 56–74, 2017.
DOI: 10.1007/978-3-662-54580-5_4

Apprehending the effects of all the relaxations allowed in such models is extremely hard. For instance, while TSO allows reordering stores past loads (of different addresses/variables) reflecting the use of store buffers, a model such as POWER allows reordering of all kinds of store and load operations under quite subtle conditions. A lot of work has been devoted to the definition of formal models that accurately capture the program semantics corresponding to models such as TSO and POWER [11,30,32,34,35]. Still, programming against weak memory models is a hard and error prone task. Therefore, developing formal verification approaches under weak memory models is of paramount importance. In particular, it is crucial in this context to have efficient algorithms for automatic bug detection. This paper addresses precisely this issue and presents an algorithmic approach for checking state reachability in concurrent programs running on the POWER semantics as defined in [21] (which is essentially the POWER model presented in [34] with small changes that have been introduced in order to increase the accuracy and the precision of the model).

The verification of concurrent programs under weak memory models is known to be complex. Indeed, encoding the buffering and storage mechanisms used in these models leads in general to complex, infinite-state formal operational models involving unbounded data structures like FIFO queues (or more generally unbounded partial order constraints). For the case of TSO, efficient, yet precise encodings of the effects of its storage mechanism have been designed recently [3,5]. It is not clear how to define such precise and practical encodings for POWER.

In this paper, we consider an alternative approach. We investigate the issue of defining approximate analysis. Our approach consists in introducing a parametric under-approximation schema in the spirit of context-bounding [12,25,27,31,33]. Context-bounding has been proposed in [33] as a suitable approach for efficient bug detection in multithreaded programs. Indeed, for concurrent programs, a bounding concept that provides both good coverage and scalability must be based on aspects related to the interactions between concurrent components. It has been shown experimentally that concurrency bugs usually show up after a small number of context switches [31].

In the context of weak memory models, context-bounded analysis has been extended in [12] to the case of programs running on TSO. The work we present here aims at extending this approach to the case of POWER. This extension is actually very challenging due to the complexity of POWER and requires developing new techniques that are different from, and much more involved than, the ones used for the case of TSO. First, we introduce a new concept of bounding that is suitable for POWER. Intuitively, the architecture of POWER is similar to a distributed system with a replicated memory, where each processor has its own replica, and where operations are propagated between replicas according to some specific protocol. Our bounding concept is based on this architecture. We consider that a computation is divided in a sequence of “contexts”, where a context is a computation segment for which there is precisely one *active* processor. All actions within a context are either operations issued by the active processor, or propagation actions performed by its storage subsystem. Then, in our analysis, we consider only computations that have a number of contexts that is less or equal than some given bound. Notice that while we bound the number of contexts in a computation, we do not put any bound on the lengths of the contexts, nor on the size of the storage system.

We prove that for every bound $\mathbb{K}$, and for every concurrent program *Prog*, it is possible to construct, using code-to-code translation, another concurrent program $Prog^{\bullet}$ such that for every $\mathbb{K}$-bounded computation π in *Prog* under the POWER semantics there is a corresponding $\mathbb{K}$-bounded computation $\pi^{\bullet}$ of $Prog^{\bullet}$ under the SC semantics that reaches the same set of states and vice-versa. Thus, the context-bounded state reachability problem for *Prog* can be reduced to the context-bounded state reachability problem for $Prog^{\bullet}$ under SC. We show that the program $Prog^{\bullet}$ has the same number of processes as *Prog*, and only $O(|\mathcal{P}||\mathcal{X}|\mathbb{K} + |\mathcal{R}|)$ additional shared variables and local registers compared to *Prog*, where $|\mathcal{P}|$ is the number of processes, $|\mathcal{X}|$ is the number of shared variables and $|\mathcal{R}|$ is the number of local registers in *Prog*. Furthermore, the obtained program has the same type of data structures and variables as the original one. As a consequence, we obtain for instance that for finite-data programs, the context-bounded analysis of programs under POWER is decidable. Moreover, our code-to-code translation allows to leverage existing verification tools for concurrent programs to carry out verification of safety properties under POWER.

To show the applicability of our approach, we have implemented our reduction, and we have used cbmc version 5.1 [17] as the backend tool for solving SC reachability queries. We have carried out several experiments showing the efficiency of our approach. Our experimental results confirm the assumption that concurrency bugs manifest themselves within small bounds of context switches. They also confirm that our approach based on context-bounding is more efficient and scalable than approaches based on bounding sizes of computations and/or of storage systems.

Related work. There has been a lot of work on automatic program verification under weak memory models, based on precise, under-approximate, and abstract analyses, e.g., [2,5,8,10,12–16,18–20,23,24,26,29,36–40]. While most of these works concern TSO, only a few of them address the safety verification problem under POWER (e.g., [6,9–11,36]). The paper [21] addresses the different issue of checking robustness against POWER, i.e., whether a program has the same (trace) semantics for both POWER and SC.

The work in [9] extends the cbmc framework by taking into account weak memory models including TSO and POWER. While this approach uses reductions to SC analysis, it is conceptually and technically different from ours. The work in [10] develops a verification technique combining partial orders with bounded model checking, that is applicable to various weak memory models including TSO and POWER. However, these techniques are not anymore supported by the latest version of cbmc. The work in [6] develops stateless model-checking techniques under POWER. In Sect. 4, we compare the performances of our approach with those of [6,9]. The tool herd [11] operates on small litmus tests under various memory models. Our tool can handle in an efficient and precise way such litmus tests.

Recently, Tomasco et al. [36] presented a new verification approach, based on code-to-code translations, for programs running under TSO and PSO. They also discuss the extension of their approach to programs running under POWER (however the detailed formalization and the implementation of this extension are kept for future work). Our approach and the one proposed in [36] are orthogonal since we are using different bounding parameters: In this paper, we are bounding the number of contexts while Tomasco et al. [36] are bounding the number of write operations.

2 Concurrent Programs

In this section, we first introduce some notations and definitions. Then, we present the syntax we use for *concurrent programs* and its semantics under POWER as in [21,34].

Preliminaries. Consider sets A and B. We use $[A \mapsto B]$ to denote the set of functions from A to B, and write $f : A \mapsto B$ to indicate that $f \in [A \mapsto B]$. We write $f(a) = \bot$ to denote that f is undefined for a. We use $f[a \leftarrow b]$ to denote the function g such that $g(a) = b$ and $g(x) = f(x)$ if $x \neq a$. We will use a function $\texttt{gen}$ which, for a given set A, returns an arbitrary element $\texttt{gen}\,(A) \in A$. For integers i, j, we use $[i..j]$ to denote the set $\{i, i+1, \ldots, j\}$. We use A^* to denote the set of finite words over A. For words $w_1, w_2 \in A^*$, we use $w_1 \cdot w_2$ to denote the concatenation of w_1 and w_2.

Syntax. Figure 1 gives the grammar for a small but general assembly-like language that we use for defining concurrent programs. A program *Prog* first declares a set $\mathcal{X}$ of (shared) variables followed by the code of a set $\mathcal{P}$ of processes. Each process p has a finite $\mathcal{R}\,(p)$ of (local) *registers*. We assume w.l.o.g. that the sets of registers of the different processes are disjoint, and define $\mathcal{R} := \cup_p \mathcal{R}\,(p)$. The code of each process $p \in \mathcal{P}$ starts by declaring a set of registers followed by a sequence of instructions.

For the sake of simplicity, we assume that the data domain of both the shared variables and registers is a single set $\mathcal{D}$. We assume a special element $0 \in \mathcal{D}$ which is the initial value of each shared variable or register. Each instruction $\mathfrak{i}$ is of the form $\lambda : \mathfrak{s}$ where λ is a unique label (across all processes) and $\mathfrak{s}$ is a statement. We define $\texttt{lbl}\,(\mathfrak{i}) := \lambda$ and $\texttt{stmt}\,(\mathfrak{i}) := \mathfrak{s}$. We define $\mathfrak{I}_p$ to be the set of instructions occurring in p, and define $\mathfrak{I} := \cup_{p \in \mathcal{P}} \mathfrak{I}_p$. We assume that $\mathfrak{I}_p$ contains a designated *initial* instruction $\mathfrak{i}_p^{init}$ from which p starts its execution. A *read* instruction in a process $p \in \mathcal{P}$ has a statement of the form $\$r \leftarrow x$, where $\$r$ is a register in p and $x \in \mathcal{X}$ is a variable. A *write* instruction has a statement of the form $x \leftarrow exp$ where $x \in \mathcal{X}$ is a variable and exp is an *expression*. We will assume a set of expressions containing a set of operators applied to constants and registers, but not referring to the content of memory (i.e., the set of variables). Assume, conditional, and iterative instructions (collectively called *aci* instructions) can be explained in a similar manner. The statement $\texttt{term}$ will cause the process to

$$\begin{aligned}
\mathit{Prog} &::= \texttt{var}\ x^*\ (\texttt{proc}\ p\ \texttt{reg}\ \$r^*\ \ \mathfrak{i}^*)^* \\
\mathfrak{i} &::= \lambda : \mathfrak{s} \\
\mathfrak{s} &::= \$r \leftarrow x \mid x \leftarrow \mathit{exp} \mid \texttt{assume}\ \mathit{exp} \\
&\quad\ \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathfrak{i}^*\ \texttt{else}\ \mathfrak{i}^* \\
&\quad\ \mid \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathfrak{i}^* \mid \texttt{term}
\end{aligned}$$

Fig. 1. Syntax of concurrent programs.

terminate its execution. We assume that $\texttt{term}$ occurs only once in the code of a process p and that it has the label $\lambda_p^{\texttt{term}}$. For an expression *exp*, we use $\mathcal{R}(exp)$ to denote the set of registers that occur in *exp*. For a write or an aci instruction $\mathtt{i}$, we define $\mathcal{R}(\mathtt{i}) := \mathcal{R}(exp)$ where *exp* is the expression that occurs in $\texttt{stmt}(\mathtt{i})$.

For an instruction $\mathtt{i} \in \mathfrak{I}_p$, we define $\texttt{next}(\mathtt{i})$ to be the set of instructions that may follow $\mathtt{i}$ in a run of a process. Notice that this set contains two elements if $\mathtt{i}$ is an aci instruction (in the case of an assume instruction, we assume that if the condition evaluates to *false*, then the process moves to $\lambda_p^{\texttt{term}} : \texttt{term}$), no element if $\mathtt{i}$ is a terminating instruction, and a single element otherwise. We define $\texttt{Tnext}(\mathtt{i})$ (resp. $\texttt{Fnext}(\mathtt{i})$) to be the (unique) instruction to which the process execution moves in case the condition in the statement of $\mathtt{i}$ evaluates to *true* (resp. *false*).

Configurations. We will assume an infinite set $\mathcal{E}$ of *events*, and will use an event to represent a single execution of an instruction in a process. A given instruction may be executed several times during a run of the program (for instance, when it is in the body of a loop). In such a case, the different executions are represented by different events. An event $\mathbb{e}$ is executed in several steps, namely it is *fetched*, *initialized*, and then *committed*. Furthermore, a write event may be propagated to the other processes. A *configuration* $\mathbb{c}$ is a tuple $\langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop}, \prec_{\texttt{co}} \rangle$, defined as follows.

Events. $\mathbb{E} \subseteq \mathcal{E}$ is a finite set of *events*, namely the events that have been created up to the current point in the execution of the program. $\texttt{ins} : \mathbb{E} \mapsto \mathfrak{I}$ is a function that maps an event $\mathbb{e}$ to the instruction $\texttt{ins}(\mathbb{e})$ that $\mathbb{e}$ is executing. We partition the set $\mathbb{E}$ into disjoint sets $\mathbb{E}_p$, for $p \in \mathcal{P}$, where $\mathbb{E}_p := \{\mathbb{e} \in \mathbb{E} \mid \texttt{ins}(\mathbb{e}) \in \mathfrak{I}_p\}$, i.e., for a process $p \in \mathcal{P}$, the set $\mathbb{E}_p$ contains the events whose instructions belong to p. For an event $\mathbb{e} \in \mathbb{E}_p$, we define $\texttt{proc}(\mathbb{e}) := p$. We say that $\mathbb{e}$ is a *write* event if $\texttt{ins}(\mathbb{e})$ is a write instruction. We use $\mathbb{E}^{\texttt{W}}$ to denote the set of write events. Similarly, we define the set $\mathbb{E}^{\texttt{R}}$ of *read* events, and the set $\mathbb{E}^{\texttt{ACI}}$ of *aci* events whose instructions are either assume, conditional, or iterative. We define $\mathbb{E}_p^{\texttt{W}}$, $\mathbb{E}_p^{\texttt{R}}$, and $\mathbb{E}_p^{\texttt{ACI}}$, to be the restrictions of the above sets to $\mathbb{E}_p$. For an event $\mathbb{e}$ where $\texttt{stmt}(\texttt{ins}(\mathbb{e}))$ is of the form $x \leftarrow exp$ or $\$r \leftarrow x$, we define $\texttt{var}(\mathbb{e}) := x$. If $\mathbb{e}$ is neither a read nor a write event, then $\texttt{var}(\mathbb{e}) := \perp$.

Program Order. The *program-order* relation $\prec \subseteq \mathbb{E} \times \mathbb{E}$ is an irreflexive partial order that describes, for a process $p \in \mathcal{P}$, the order in which events are fetched from the code of p. We require that (i) $\mathbb{e}_1 \not\prec \mathbb{e}_2$ if $\texttt{proc}(\mathbb{e}_1) \neq \texttt{proc}(\mathbb{e}_2)$, i.e., $\prec$ only relates events belonging to the same process, and that (ii) $\prec$ is a total order on $\mathbb{E}_p$.

Status. The function $\texttt{status} : \mathbb{E} \mapsto \{\texttt{fetch}, \texttt{init}, \texttt{com}\}$ defines, for an event $\mathbb{e}$, the current *status* of $\mathbb{e}$, i.e., whether it has been fetched, initialized, or committed.

Propagation. The function $\texttt{Prop} : \mathcal{P} \times \mathcal{X} \mapsto \mathbb{E}^{\texttt{W}} \cup \mathcal{E}^{\texttt{init}}$ defines, for a process $p \in \mathcal{P}$ and variable $x \in \mathcal{X}$, the latest write event on x that has been propagated to p. Here $\mathcal{E}^{\texttt{init}} := \{\mathbb{e}_x^{\texttt{init}} \mid x \in \mathcal{X}\}$ is a set disjoint from the set of events $\mathcal{E}$, and will be used to define the initial values of the variables.

Read-From. The function $\mathtt{rf} : \mathbb{E}^{\mathtt{R}} \mapsto \mathbb{E}^{\mathtt{W}} \cup \mathcal{E}^{\mathtt{init}}$ defines, for a read event $\mathbb{e} \in \mathbb{E}^{\mathtt{R}}$, the write event $\mathtt{rf}\,(\mathbb{e})$ from which $\mathbb{e}$ gets its value.

Coherence Order. All processes share a global view about the order in which write events are propagated. This is done through the *coherence order* $\prec_{\mathtt{co}}$ that is a partial order on $\mathbb{E}^{\mathtt{W}}$ s.t. $\mathbb{e}_1 \prec_{\mathtt{co}} \mathbb{e}_2$ only if $\mathtt{var}\,(\mathbb{e}_1) = \mathtt{var}\,(\mathbb{e}_2)$, i.e., it relates only events that write on identical variables. If a write event $\mathbb{e}_1$ is propagated to a process before another write event $\mathbb{e}_2$ and both events write on the same variable, then $\mathbb{e}_1 \prec_{\mathtt{co}} \mathbb{e}_2$ holds. Furthermore, the events cannot be propagated to any other process in the reverse order. However, it might be the case that a write event is never propagated to a given process.

Dependencies. We introduce a number of dependency orders on events that we will use in the definition of the semantics. We define the *per-location program-order* $\prec_{\mathtt{poloc}} \subseteq \mathbb{E} \times \mathbb{E}$ such that $\mathbb{e}_1 \prec_{\mathtt{poloc}} \mathbb{e}_2$ if $\mathbb{e}_1 \prec \mathbb{e}_2$ and $\mathtt{var}\,(\mathbb{e}_1) = \mathtt{var}\,(\mathbb{e}_2)$, i.e., it is the restriction of $\prec$ to events with identical variables. We define the *data dependency* order $\prec_{\mathtt{data}}$ s.t. $\mathbb{e}_1 \prec_{\mathtt{data}} \mathbb{e}_2$ if (i) $\mathbb{e}_1 \in \mathbb{E}^{\mathtt{R}}$, i.e., $\mathbb{e}_1$ is a read event; (ii) $\mathbb{e}_2 \in \mathbb{E}^{\mathtt{W}} \cup \mathbb{E}^{\mathtt{ACI}}$, i.e., $\mathbb{e}_2$ is either a write or an aci event; (iii) $\mathbb{e}_1 \prec \mathbb{e}_2$; (iv) $\mathtt{stmt}\,(\mathtt{ins}\,(\mathbb{e}_1))$ is of the form $\$r \leftarrow x$; (v) $\$r \in \mathcal{R}\,(\mathtt{ins}\,(\mathbb{e}_2))$; and (vi) there is no event $\mathbb{e}_3 \in \mathbb{E}^{\mathtt{R}}$ such that $\mathbb{e}_1 \prec \mathbb{e}_3 \prec \mathbb{e}_2$ and $\mathtt{stmt}\,(\mathtt{ins}\,(\mathbb{e}_3))$ is of the form $\$r \leftarrow y$. Intuitively, the loaded value by $\mathbb{e}_1$ is used to compute the value of the expression in the statement on the instruction of $\mathbb{e}_2$. We define the *control dependency* order $\prec_{\mathtt{ctrl}}$ such that $\mathbb{e}_1 \prec_{\mathtt{ctrl}} \mathbb{e}_2$ if $\mathbb{e}_1 \in \mathbb{E}^{\mathtt{ACI}}$ and $\mathbb{e}_1 \prec \mathbb{e}_2$.

We say that $\mathbb{c}$ is *committed* if $\mathtt{status}\,(\mathbb{e}) = \mathtt{com}$ for all events $\mathbb{e} \in \mathbb{E}$. The *initial configuration* $\mathbb{c}_{init}$ is defined by $\langle \emptyset, \emptyset, \lambda\mathbb{e}.\bot, \lambda\mathbb{e}.\bot, \lambda\mathbb{e}.\bot, \lambda p.\lambda x.\mathbb{e}_x^{\mathtt{init}}, \emptyset \rangle$. We use $\mathbb{C}$ to denote the set of all configurations.

Transition Relation. We define the transition relation as a relation $\rightarrow\ \subseteq \mathbb{C} \times \mathcal{P} \times \mathbb{C}$. For configurations $\mathbb{c}_1, \mathbb{c}_2 \in \mathbb{C}$ and a process $p \in \mathcal{P}$, we write $\mathbb{c}_1 \xrightarrow{p} \mathbb{c}_2$ to denote that $\langle \mathbb{c}_1, p, \mathbb{c}_2 \rangle \in\rightarrow$. Intuitively, this means that p moves from the current configuration $\mathbb{c}_1$ to $\mathbb{c}_2$. The relation $\rightarrow$ is defined through the set of inference rules shown in Fig. 2.

The rule $\mathtt{Fetch}$ chooses the next instruction to be executed in the code of a process $p \in \mathcal{P}$. This instruction should be a possible successor of the instruction that was last executed by p. To satisfy this condition, we define $\mathtt{MaxI}\,(\mathbb{c}, p)$ to be the set of instructions as follows: (i) If $\mathbb{E}_p = \emptyset$ then define $\mathtt{MaxI}\,(\mathbb{c}, p) := \left\{\mathbb{i}_p^{init}\right\}$, i.e., the first instruction fetched by p is $\mathbb{i}_p^{init}$. (ii) If $\mathbb{E}_p \neq \emptyset$, let $\mathbb{e}'$ be the maximal event of p (w.r.t. $\prec$) in the configuration $\mathbb{c}$ and then define $\mathtt{MaxI}\,(\mathbb{c}, p) := \mathtt{next}\,(\mathtt{ins}\,(\mathbb{e}'))$. In other words, we consider the instruction $\mathbb{i}' = \mathtt{ins}\,(\mathbb{e}') \in \mathfrak{I}_p$, and take its possible successors. The possibility of choosing any of the (syntactically) possible successors corresponds to *speculatively* fetching statements. As seen below, whenever we commit an aci event, we check whether the made speculations are correct or not. We create a new event $\mathbb{e}$, label it by $\mathbb{i} \in \mathtt{MaxI}\,(\mathbb{c}, p)$, and make it larger than all the other events of p w.r.t. $\prec$. In such a way, we maintain the property that the order on the events of p reflects the order in which they are fetched in the current run of the program.

There are two ways in which read events get their values, namely either from *local* write events that are performed by the process itself, or from write events that

$$\frac{\mathbb{e} \notin \mathbb{E},\ \prec' = \prec \cup \left\{\langle \mathbb{e}', \mathbb{e}\rangle \mid \mathbb{e}' \in \mathbb{E}_p\right\},\ \mathbb{i} \in \mathtt{MaxI}\,(\mathbb{c}, p)}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E} \cup \{\mathbb{e}\}, \prec', \mathtt{ins}[\mathbb{e} \leftarrow \mathbb{i}], \mathtt{status}[\mathbb{e} \leftarrow \mathtt{fetch}], \mathtt{rf}, \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Fetch}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{R}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{fetch},\ \mathtt{CW}\,(\mathbb{c}, \mathbb{e}) = \mathbb{e}',\ \mathtt{status}\,(\mathbb{e}') = \mathtt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Local\text{-}Read}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{R}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{fetch},\ (\mathtt{CW}\,(\mathbb{c}, \mathbb{e}) = \bot) \vee (\mathtt{CW}\,(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \mathtt{status}\,(\mathbb{e}') = \mathtt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{rf}[\mathbb{e} \leftarrow \mathtt{Prop}\,(\mathtt{p}, \mathtt{var}\,(\mathbb{e}))], \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Prop\text{-}Read}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{R}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{init},\ \mathtt{ComCnd}\,(\mathbb{c}, \mathbb{e}),\ \mathtt{RdCnd}\,(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Com\text{-}Read}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{W}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{fetch},\ \mathtt{WrInitCnd}\,(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Init\text{-}Write}$$

$$\frac{\begin{gathered}\mathbb{e} \in \mathbb{E}_p^{\mathtt{W}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{init},\ \mathtt{ComCnd}\,(\mathbb{c}, \mathbb{e}),\\ \prec'_{\mathtt{co}} = \prec_{\mathtt{co}} \cup \left\{\langle \mathbb{e}', \mathbb{e}\rangle \mid \mathbb{e}' \preceq_{\mathtt{co}} \mathtt{Prop}\,(\mathtt{p}, \mathtt{var}\,(\mathbb{e}))\right\}\end{gathered}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}[\langle \mathtt{p}, \mathtt{var}\,(\mathbb{e})\rangle \leftarrow \mathbb{e}], \prec'_{\mathtt{co}}\rangle}\ \mathtt{Com\text{-}Write}$$

$$\frac{\begin{gathered}q \in \mathcal{P},\ \mathbb{e} \in \mathbb{E}_p^{\mathtt{W}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{com},\ \mathtt{Prop}\,(\mathtt{q}, \mathtt{var}\,(\mathbb{e})) \prec_{\mathtt{co}} \mathbb{e},\\ \prec'_{\mathtt{co}} = \prec_{\mathtt{co}} \cup \left\{\langle \mathbb{e}', \mathbb{e}\rangle \mid \mathbb{e}' \preceq_{\mathtt{co}} \mathtt{Prop}\,(\mathtt{q}, \mathtt{var}\,(\mathbb{e}))\right\}\end{gathered}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}, \mathtt{rf}, \mathtt{Prop}[\langle \mathtt{q}, \mathtt{var}\,(\mathbb{e})\rangle \leftarrow \mathbb{e}], \prec'_{\mathtt{co}}\rangle}\ \mathtt{Prop}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{ACI}},\ \mathtt{status}\,(\mathbb{e}) = \mathtt{fetch},\ \mathtt{ComCnd}\,(\mathbb{c}, \mathbb{e}),\ \mathtt{ValidCnd}\,(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}, \prec_{\mathtt{co}}\rangle}\ \mathtt{Com\text{-}ACI}$$

Fig. 2. Inference rules defining the relation $\xrightarrow{p}$ where $p \in \mathcal{P}$.

are *propagated* to the process. The first case is covered by the rule `Local-Read` in which the process p initializes a read event $\mathbb{e} \in \mathbb{E}^{\mathtt{R}}$ on a variable (say x), where $\mathbb{e}$ has already been fetched. Here, the event $\mathbb{e}$ is made to read its value from a local write event $\mathbb{e}' \in \mathbb{E}_p^{\mathtt{W}}$ on x such that (i) $\mathbb{e}'$ has been initialized but not yet committed, and such that (ii) $\mathbb{e}'$ is the closest write event that precedes $\mathbb{e}$ in the order $\prec_{\mathtt{poloc}}$. Notice that, by condition (ii), $\mathbb{e}'$ is unique if it exists. To formalize this, we define the *Closest Write* function $\mathtt{CW}\,(\mathbb{c}, \mathbb{e}) := \mathbb{e}'$ where $\mathbb{e}'$ is the unique event such that (i) $\mathbb{e}' \in \mathbb{E}^{\mathtt{W}}$, (ii) $\mathbb{e}' \prec_{\mathtt{poloc}} \mathbb{e}$, and (iii) there is no event $\mathbb{e}''$ such that $\mathbb{e}'' \in \mathbb{E}^{\mathtt{W}}$ and $\mathbb{e}' \prec_{\mathtt{poloc}} \mathbb{e}'' \prec_{\mathtt{poloc}} \mathbb{e}$. Notice that $\mathbb{e}'$ may not exist, i.e., it may be the case that $\mathtt{CW}\,(\mathbb{c}, \mathbb{e}) = \bot$. If $\mathbb{e}'$ exists and it has been inititialized but not commited, we initialize $\mathbb{e}$ and update the read-from relation appropriately. On the other hand, if such an event does not exist, i.e., if there is no write event on x before $\mathbb{e}$ by p, or if the closest write event on x before $\mathbb{e}$ by p has already been committed, then we use the rule `Prop-Read` to let $\mathbb{e}$ fetch its value from the latest write event on x that has been propagated to p. Notice this event is the value of $\mathtt{Prop}\,(p, x)$.

To commit an initialized read event $\mathbb{e} \in \mathbb{E}_p^{\mathtt{R}}$, we use the rule Com-Read. The rule can be performed if $\mathbb{e}$ satisfies two conditions in $\mathbb{c}$. The first condition is defined as $\mathtt{RdCnd}(\mathbb{c}, \mathbb{e}) := \forall \mathbb{e}' \in \mathbb{E}^{\mathtt{R}} : (\mathbb{e}' \prec_{\mathtt{poloc}} \mathbb{e}) \implies (\mathtt{rf}(\mathbb{e}') \preceq_{\mathtt{co}} \mathtt{rf}(\mathbb{e}))$. It states that for any read event $\mathbb{e}'$ such that $\mathbb{e}'$ precedes $\mathbb{e}$ in the order $\prec_{\mathtt{poloc}}$, the write event from which $\mathbb{e}'$ reads its value is equal to or precedes the write event for $\mathbb{e}$ in the coherence order $\prec_{\mathtt{co}}$. The second condition is defined by $\mathtt{ComCnd}(\mathbb{c}, \mathbb{e}) := \forall \mathbb{e}' \in \mathbb{E} : (\mathbb{e}' \prec_{\mathtt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\mathtt{ctrl}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\mathtt{poloc}} \mathbb{e}) \implies (\mathtt{status}(\mathbb{e}') = \mathtt{com})$. It states that all events $\mathbb{e}' \in \mathbb{E}$ that precede $\mathbb{e}$ in one of the orders $\prec_{\mathtt{data}}$, $\prec_{\mathtt{ctrl}}$, or $\prec_{\mathtt{poloc}}$ should have already been committed.

To initialize a fetched write event $\mathbb{e} \in \mathbb{E}_p^{\mathtt{R}}$, we use the rule Init-Write that requires all events that precede $\mathbb{e}$ in the order $\prec_{\mathtt{data}}$ should have already been initialized. This condition is formulated as $\mathtt{WrInitCnd}(\mathbb{c}, \mathbb{e}) := \forall \mathbb{e}' \in \mathbb{E}^{\mathtt{R}} : (\mathbb{e}' \prec_{\mathtt{data}} \mathbb{e}) \implies (\mathtt{status}(\mathbb{e}') = \mathtt{init} \vee \mathtt{status}(\mathbb{e}') = \mathtt{com})$. When a write event in a process $p \in \mathcal{P}$ is committed, it is also immediately propagated to p itself. To maintain the coherence order, the semantics keeps the invariant that the latest write event on a variable $x \in \mathcal{X}$ that has been propagated to a process $p \in \mathcal{P}$ is the largest one in the coherence order among all write events on x that have been propagated to p up to now in the run. This invariant is maintained in Com-Write by requiring that the event $\mathbb{e}$ (that is being propagated) is strictly larger in the coherence order than the latest write event on the same variable as $\mathbb{e}$ that has been propagated to p.

Write events are propagated to other processes through the rule Prop. A write event $\mathbb{e}$ on a variable x is allowed to be propagated to a process q only if it has a coherence order that is strictly larger than the coherence of any event that has been to propagated to q up to now. Notice that this is given by coherence order of $\mathtt{Prop}(q, x)$ which is the latest write event on x that has been propagated to q.

When committing an aci event through the rule Com-ACI, we also require that we verify any potential speculation that have been made when fetching the subsequent events. We assume that we are given a function $\mathtt{Val}(\mathbb{c}, \mathbb{e})$ that takes as input an aci event $\mathbb{e}$ and returns the value of the expression of the conditional statement in the instruction of $\mathbb{e}$ when evaluated in the configuration $\mathbb{c}$. The $\mathtt{Val}(\mathbb{c}, \mathbb{e})$ is only defined when all events that precede $\mathbb{e}$ in the order $\prec_{\mathtt{data}}$ should have already been initialized.

To that end, we define predicate $\mathtt{ValidCnd}(\mathbb{c}, \mathbb{e}) := (\exists \mathbb{e}' \in \mathbb{E} :\ \mathbb{e} \prec \mathbb{e}' \wedge \nexists \mathbb{e}'' \in \mathbb{E} :\ \mathbb{e} \prec \mathbb{e}'' \prec \mathbb{e}') \implies ((\mathtt{Val}(\mathbb{c}, \mathbb{e}) = \mathit{true} \wedge \mathtt{ins}(\mathbb{e}') = \mathtt{Tnext}(\mathtt{ins}(\mathbb{e}))) \vee (\mathtt{Val}(\mathbb{c}, \mathbb{e}) = \mathit{false} \wedge \mathtt{ins}(\mathbb{e}') = \mathtt{Fnext}(\mathtt{ins}(\mathbb{e}))))$. The rule intuitively finds the event $\mathbb{e}'$ that was fetched immediately after $\mathbb{e}$. Notice that such an event may not exist and it is unique if it exists. The predicate requires the choice of $\mathbb{e}'$ is consistent with the value $\mathtt{Val}(\mathbb{c}, \mathbb{e})$ of the expression in the statement of the instruction of $\mathbb{e}$.

Bounded Reachability. A *run* π is a sequence of transitions $\mathbb{c}_0 \xrightarrow{p_1} \mathbb{c}_1 \xrightarrow{p_2} \mathbb{c}_2 \cdots \mathbb{c}_{n-1} \xrightarrow{p_n} \mathbb{c}_n$. In such a case, we write $\mathbb{c}_0 \xrightarrow{\pi} \mathbb{c}_n$. We define $\mathtt{last}(\pi) := \mathbb{c}_n$. We define $\pi \uparrow := p_1 p_2 \cdots p_n$, i.e., it is the sequence of processes performing the transitions in π. For a sequence $\sigma = p_1 p_2 \cdots p_n \in \mathcal{P}^*$, we say that σ is a *context* if there is a process $p \in \mathcal{P}$ such that $p_i = p$ for all $i : 1 \leq i \leq n$. We say that π is *committed* (resp. *k-bounded*) if $\mathtt{last}(\pi)$ is committed (resp. if $\pi \uparrow = \sigma_1 \cdot \sigma_2 \cdot \cdots \cdot \sigma_k$ where σ_i is a context for all $i : 1 \leq i \leq k$).

For $\mathbb{c} \in \mathbb{C}$ and $p \in \mathcal{P}$, we define the set of *reachable labels* of the configuration $\mathbb{c}$ as follows. (i) If $\mathbb{c} = \mathbb{c}_{init}$ then $\mathtt{lbl}\,(\mathbb{c}) := \{\bot\}$, i.e. process p does not reach to any label in the initial configuration. (ii) If $\mathbb{c} \neq \mathbb{c}_{init}$, let $\mathbb{e}$ be the maximal event of p (w.r.t. $\prec$) in $\mathbb{c}$. We define $\mathtt{lbl}\,(\mathbb{c}) := \{\mathtt{lbl}\,(\mathtt{ins}\,(\mathbb{e}))\}$, i.e. process p reaches to the label of the maximal event $\mathbb{e}$ of p (w.r.t. $\prec$) in the configuration $\mathbb{c}$. In the *reachability problem*, we are given a label λ and asked whether there is a committed run π and a configuration $\mathbb{c}$ such that $\mathbb{c}_{init} \xrightarrow{\pi} \mathbb{c}$ where $\lambda \in \mathtt{lbl}\,(\mathbb{c})$. For a natural number $\mathbb{K}$, the $\mathbb{K}$-*bounded reachability problem* is defined by requiring that the run π in the above definition is $\mathbb{K}$-bounded.

3 Translation

In this section, we introduce an algorithm that reduces, for a given number $\mathbb{K}$, the $\mathbb{K}$-bounded reachability problem for POWER to the corresponding problem for SC. Given an input concurrent program *Prog*, the algorithm constructs an output concurrent program $Prog^\bullet$ whose size is polynomial in *Prog* and $\mathbb{K}$, such that for each $\mathbb{K}$-bounded run π in *Prog* under the POWER semantics there is a corresponding $\mathbb{K}$-bounded run $\pi^\bullet$ of $Prog^\bullet$ under the SC semantics that reaches the same set of process labels. Below, we first present a scheme for the translation of *Prog*, and mention some of the challenges that arise due to the POWER semantics. Then, we give a detailed description of the data structures we use in $Prog^\bullet$. Finally, we describe the codes of the processes in $Prog^\bullet$.

$$
\begin{aligned}
\llbracket Prog \rrbracket_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{var}\ \xcancel{x^*}\ \langle \mathtt{addvars} \rangle_{\mathbb{K}}\,;\ \langle \mathtt{iniProc} \rangle_{\mathbb{K}} \\
&\phantom{\stackrel{\text{def}}{=}}\ \langle \mathtt{verProc} \rangle_{\mathbb{K}}\ (\llbracket \mathtt{proc}\ p\ \mathtt{reg}\ \$r^*\ \mathtt{i}^* \rrbracket_{\mathbb{K}})^* \\
\langle \mathtt{addvars} \rangle_{\mathbb{K}} &\stackrel{\text{def}}{=} \mu\,(|\mathcal{P}|,|\mathcal{X}|,\mathbb{K})\ \mu^{init}\,(|\mathcal{P}|,|\mathcal{X}|,\mathbb{K}) \\
&\phantom{\stackrel{\text{def}}{=}}\ \alpha\,(|\mathcal{P}|,|\mathcal{X}|,\mathbb{K})\ \alpha^{init}\,(|\mathcal{P}|,|\mathcal{X}|,\mathbb{K}) \\
&\phantom{\stackrel{\text{def}}{=}}\ \nu\,(|\mathcal{P}|,|\mathcal{X}|)\ \mathtt{iR}\,(|\mathcal{P}|,|\mathcal{X}|)\ \mathtt{cR}\,(|\mathcal{P}|,|\mathcal{X}|) \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{iW}\,(|\mathcal{P}|,|\mathcal{X}|)\ \mathtt{cW}\,(|\mathcal{P}|,|\mathcal{X}|)\ \mathtt{iReg}\,(|\mathcal{R}|) \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{cReg}\,(|\mathcal{R}|)\,\mathtt{ctrl}\,(|\mathcal{P}|)\,\mathtt{active}\,(\mathbb{K})\,\mathtt{cntxt} \\
\langle \mathtt{iniProc} \rangle_{\mathbb{K}} &\stackrel{\text{def}}{=} \llbracket \mathtt{iniProc} \rrbracket_{\mathbb{K}} \\
\langle \mathtt{verProc} \rangle_{\mathbb{K}} &\stackrel{\text{def}}{=} \llbracket \mathtt{verProc} \rrbracket_{\mathbb{K}} \\
\llbracket \mathtt{proc}\ p\ \mathtt{reg}\ \$r^*\,\mathtt{i}^* \rrbracket_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{proc}\ p\ \mathtt{reg}\ \$r^*\ (\llbracket \mathtt{i} \rrbracket^{p}_{\mathbb{K}})^* \\
\llbracket \mathtt{i} \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \lambda : \langle \mathtt{activeCnt} \rangle^{p}_{\mathbb{K}}\ \llbracket \mathfrak{s} \rrbracket^{p}_{\mathbb{K}}\ \langle \mathtt{closeCnt} \rangle^{p}_{\mathbb{K}} \\
\langle \mathtt{activeCnt} \rangle^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{assume}\,(\mathtt{active}\,(\mathtt{cntxt}) = p) \\
\langle \mathtt{closeCnt} \rangle^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{cntxt} \leftarrow \mathtt{cntxt} + \mathtt{gen}\,([0..\mathbb{K}-1])\,; \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{assume}(\mathtt{cntxt} \leq \mathbb{K}) \\
\llbracket \$r \leftarrow x \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \llbracket \$r \leftarrow x \rrbracket^{p,\mathtt{Read}}_{\mathbb{K}} \\
\llbracket x \leftarrow exp \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \llbracket x \leftarrow exp \rrbracket^{p,\mathtt{Write}}_{\mathbb{K}} \\
\llbracket \mathtt{assume}\ exp \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{assume}\ exp;\ \langle \mathtt{control} \rangle^{p}_{\mathbb{K}} \\
\llbracket \mathtt{if}\ exp\ \mathtt{then}\ \mathtt{i}^* &\stackrel{\text{def}}{=} \mathtt{if}\ exp\ \mathtt{then}\ (\llbracket \mathtt{i} \rrbracket^{p}_{\mathbb{K}})^* \\
\mathtt{else}\ \mathtt{i}^* \rrbracket^{p}_{\mathbb{K}} &\phantom{\stackrel{\text{def}}{=}}\ \mathtt{else}\ (\llbracket \mathtt{i} \rrbracket^{p}_{\mathbb{K}})^*; \langle \mathtt{control} \rangle^{p}_{\mathbb{K}} \\
\llbracket \mathtt{while}\ exp\ \mathtt{do}\ \mathtt{i}^* \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{while}\ exp\ \mathtt{do}\ (\llbracket \mathtt{i} \rrbracket^{p}_{\mathbb{K}})^*; \langle \mathtt{control} \rangle^{p}_{\mathbb{K}} \\
\langle \mathtt{control} \rangle^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{ctrl}\,(p) \leftarrow \mathtt{ctrl}\,(p) + \mathtt{gen}\,([0..\mathbb{K}-1])\,; \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{assume}(\mathtt{ctrl}\,(p) \leq \mathbb{K}) \\
\llbracket \mathtt{term} \rrbracket^{p}_{\mathbb{K}} &\stackrel{\text{def}}{=} \mathtt{term}
\end{aligned}
$$

Fig. 3. Translation map $\llbracket . \rrbracket_{\mathbb{K}}$. We omit the label of an intermediary instruction when it is not relevant.

Scheme. Our construction is based on code-to-code translation scheme that transforms the program *Prog* into the program $Prog^\bullet$ following the map function $\llbracket . \rrbracket_{\mathbb{K}}$ given in Fig. 3. Let $\mathcal{P}$ and $\mathcal{X}$ be the sets of processes and (shared) variables in *Prog*. The map $\llbracket . \rrbracket_{\mathbb{K}}$ *replaces* the variables of *Prog* by $(|\mathcal{P}| \cdot (2\mathbb{K}+1))$ copies of the set $\mathcal{X}$, in addition to a finite set of *finite-data* structures (which will be formally defined in the **Data Structures** paragraph). The map function then declares two additional processes `iniProc` and `verProc` that will be used to initialize the data structures and to check the reachability problem at the end of the run of $Prog^\bullet$. The formal definition of `iniProc` (resp. `verProc`) will be given in the **Initializing process** (resp. **Verifier**

process) paragraph. Furthermore, the map function $[\![.]\!]_{\mathbb{K}}$ transforms the code of each process $p \in \mathcal{P}$ to a corresponding process $p^{\bullet}$ that will simulate the moves of p. The processes p and $p^{\bullet}$ will have the same set of registers. For each instruction $\mathfrak{i}$ appearing in the code of the process p, the map $[\![\mathfrak{i}]\!]_{\mathbb{K}}^{p}$ transforms it to a sequence of instructions as follows: First, it adds the code defined by `activeCnt` to check if the process p is active during the current context, then it transforms the statement $\mathfrak{s}$ of the instruction $\mathfrak{i}$ into a sequence of instructions following the map $[\![\mathfrak{s}]\!]_{\mathbb{K}}^{p}$, and finally it adds the sequence of instructions defined by `closeCnt` to guess the occurrence of a context switch. The translation of an aci statement keeps the same statements and adds `control` to guess the contexts when the corresponding event will be committed. The terminating statement remains identical by the map function $[\![\texttt{term}]\!]_{\mathbb{K}}^{p}$. The translations of write and read statements will be described in the **Write Instructions** and **Read Instructions** paragraphs respectively.

Challenges. There are two *aspects* of the POWER semantics (cf. Sect. 2) that make it difficult to simulate the run π under the SC semantics, namely *non-atomicity* and *asynchrony*. First, events are not executed atomically. In fact, an event is first fetched and initialized before it is committed. In particular, an event may be fetched in one context and be initialized and committed only in later contexts. Since there is no bound on the number of events that may be fetched in a given context, our simulation should be able to handle unbounded numbers of pending events. Second, write events of one process are propagated in an *asynchronous* manner to the other processes. This implies that we may have unbounded numbers of "traveling" events that are committed in one context and propagated to other processes only in subsequent contexts. This creates two *challenges* in the simulation. On the one hand, we need to keep track of the coherence order among the different write events. On the other hand, since write events are not distributed to different processes at the same time, the processes may have different views of the values of a given variable at a given point of time.

Since it is not feasible to record the initializing, committing, and propagating contexts of an unbounded number of events in an SC run, our algorithm will instead predict the *summary* of effects of arbitrarily long sequences of events that may occur in a given context. This is implemented using an intricate scheme that first *guesses* and then *checks* these summaries. Concretely, each event $\mathbb{e}$ in the run π is simulated by a sequence of instructions in $\pi^{\bullet}$. This sequence of instructions will be executed atomically (without interruption from other processes and events). More precisely, if $\mathbb{e}$ is fetched in a context $k : 1 \leq k \leq \mathbb{K}$, then the corresponding sequence of instructions will be executed in the same context k in $\pi^{\bullet}$. Furthermore, we let $\pi^{\bullet}$ *guess* (*speculate*) (i) the contexts in which $\mathbb{e}$ will be initialized, committed, and propagated to other processes, and (ii) the values of variables that are seen by read operations. Then, we *check* whether the guesses made by $\pi^{\bullet}$ are valid w.r.t. the POWER semantics. As we will see below, these checks are done both on-the-fly during $\pi^{\bullet}$, as well as at the end of $\pi^{\bullet}$. To implement the guess-and-check scheme, we use a number of data structures, described below.

Data Structures. We will introduce the data structures used in our simulation in order to deal with the above asynchrony and non-atomicity challenging aspects.

Asynchrony. In order to keep track of the coherence order, we associate a *time stamp* with each write event. A time stamp τ is a mapping $\mathcal{P} \mapsto \mathbb{K}^{\otimes}$ where $\mathbb{K}^{\otimes} := \mathbb{K} \cup \{\otimes\}$. For a process $p \in \mathcal{P}$, the value of $\tau(p)$ represents the context in which the given event is propagated to p. In particular, if $\tau(p) = \otimes$ then the event is never propagated to p. We use $\mathbb{T}$ to denote the set of time stamps. We define an order $\sqsubseteq$ on $\mathbb{T}$ such that $\tau_1 \sqsubseteq \tau_2$ if, for all processes $p \in \mathcal{P}$, either $\tau_1(p) = \otimes$, or $\tau_2(p) = \otimes$, or $\tau_1(p) \leq \tau_2(p)$. Notice that if $\tau_1 \sqsubseteq \tau_2$ and there is a process $p \in \mathcal{P}$ such that $\tau_1(p) \neq \otimes$, $\tau_2(p) \neq \otimes$, and $\tau_1(p) < \tau_2(p)$ then $\tau_1(q) \leq \tau_2(q)$ whenever $\tau_1(q) \neq \otimes$ and $\tau_2(q) \neq \otimes$. In such a case, $\tau_1 \sqsubset \tau_2$. On the other hand, if either $\tau_1(p) = \otimes$ or $\tau_2(p) = \otimes$ for all $p \in \mathcal{P}$, then both $\tau_1 \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_1$. The coherence order $\prec_{\mathsf{co}}$ on write events will be reflected in the order $\sqsubseteq$ on their time stamps. In particular, for events $\mathbb{e}_1$ and $\mathbb{e}_2$ with time stamps τ_1 and τ_2 respectively, if $\tau_1 \sqsubset \tau_2$ then $\mathbb{e}_1$ precedes $\mathbb{e}_2$ in coherence order. The reason is that there is at least one process p to which both $\mathbb{e}_1$ and $\mathbb{e}_2$ are propagated, and $\mathbb{e}_1$ is propagated to p before $\mathbb{e}_2$. However, if both $\tau_1 \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_1$ then the events are never propagated to the same process, and hence they need not to be related by the coherence order.

If $\tau_1 \sqsubseteq \tau_2$ then we define the *summary* of τ_1 and τ_2, denoted by $\tau_1 \oplus \tau_2$, to be the time stamp τ such that $\tau(p) = \tau_1(p)$ if $\tau_2(p) = \otimes$, and $\tau(p) = \tau_2(p)$ otherwise. For a sequence $\sigma = \tau_0 \sqsubseteq \tau_1 \sqsubseteq \cdots \sqsubseteq \tau_n$ of time stamps, we define the summary $\oplus\,\sigma := \tau'_n$ where τ'_i is defined inductively by $\tau'_0 := \tau_0$, and $\tau'_i := \tau'_{i-1} \oplus \tau_i$ for $i : 1 \leq i \leq n$. Notice that, for $p \in \mathcal{P}$, we have $\oplus\,\sigma(p) = \tau_i(p)$ where i is the largest $j : 1 \leq j \leq n$ s.t. $\tau_j(p) \neq \otimes$.

Our simulation observes the sequence of write events received by a process in each context. In fact, the simulation will initially *guess* and later *verify* the summaries of the time stamps of such a sequence. This is done using data structures α^{init} and α. The mapping $\alpha^{init} : \mathcal{P} \times \mathcal{X} \times \mathbb{K} \mapsto [\mathcal{P} \mapsto \mathbb{K}^{\otimes}]$ stores, for a process $p \in \mathcal{P}$, a variable $x \in \mathcal{X}$, and a context $k : 1 \leq k \leq \mathbb{K}$, an *initial guess* $\alpha^{init}(p, x, k)$ of the summary of the time stamps of the sequence of write events on x propagated to p up to the *start* of context k. Starting from a given initial guess for a given context k, the time stamp is updated successively using the sequence of write events on x propagated to p in k. The result is stored using the mapping $\alpha : \mathcal{P} \times \mathcal{X} \times \mathbb{K} \mapsto [\mathcal{P} \mapsto \mathbb{K}^{\otimes}]$. More precisely, we initially set the value of α to α^{init}. Each time a new write event $\mathbb{e}$ on x is created by p in the context k, we guess the time stamp β of $\mathbb{e}$, and then update $\alpha(p, x, k)$ by computing its summary with β. Thus, given a point in a context k, $\alpha(p, x, k)$ contains the summary of the time stamps of the whole sequence of write events on x that have been propagated to p up to that point. At the end of the simulation, we *verify*, for each context $k : 1 \leq k < \mathbb{K}$, that the value of α for a context k is equal to the value of α^{init} for the next context $k + 1$.

Furthermore, we use three data structures for storing the values of variables. The mapping $\mu^{init} : \mathcal{P} \times \mathcal{X} \times \mathbb{K} \mapsto \mathcal{D}$ stores, for a process $p \in \mathcal{P}$, a variable $x \in \mathcal{X}$, and a context $k : 1 \leq k \leq \mathbb{K}$, an *initial guess* $\mu^{init}(p, x, k)$ of the value of the latest write event on x propagated to p up to the *start* of the context k. The mapping $\mu : \mathcal{P} \times \mathcal{X} \times \mathbb{K} \mapsto \mathcal{D}$ stores, for a process $p \in \mathcal{P}$, a variable $x \in \mathcal{X}$, and a point in a context $k : 1 \leq k \leq \mathbb{K}$, the value $\mu(p, x, k)$ of the latest write event on x that has been propagated to p up to that point. Moreover, the mapping $\nu : \mathcal{P} \times \mathcal{X} \mapsto \mathcal{D}$ stores, for a process $p \in \mathcal{P}$ and a variable $x \in \mathcal{X}$, the latest value $\nu(p, x)$ that has been written on x by p.

Non-atomicity. In order to satisfy the different dependencies between events, we need to keep track of the contexts in which they are initialized and committed. One aspect of our translation is that it only needs to keep track of the *context* in which the *latest* read or write event on a given variable in a given process is initialized or committed. The mapping $\texttt{iW} : \mathcal{P} \times \mathcal{X} \mapsto \mathbb{K}$ defines, for $p \in \mathcal{P}$ and $x \in \mathcal{X}$, the context $\texttt{iW}(p, x)$ in which the latest write event on x by p is initialized. The mapping $\texttt{cW} : \mathcal{P} \times \mathcal{X} \mapsto \mathbb{K}$ is defined in a similar manner for committing (rather than initializing) write events. Furthermore, we define similar mappings $\texttt{iR}$ and $\texttt{cR}$ for read events. The mapping $\texttt{iReg} : \mathcal{R} \mapsto \mathbb{K}$ gives, for a register $\$r \in \mathcal{R}$, the initializing context $\texttt{iReg}(\$r)$ of the latest read event loading a value to $\$r$. For an expression *exp*, we define $\texttt{iReg}(exp) := \max\{\texttt{iReg}(\$r) \mid \$r \in \mathcal{R}(exp)\}$. The mapping $\texttt{cReg} : \mathcal{R} \mapsto \mathbb{K}$ gives the contexts for committing (rather than initializing) of the read events. We extend $\texttt{cReg}$ from registers to expressions in a similar manner to $\texttt{iReg}$. Finally, the mapping $\texttt{ctrl} : \mathcal{P} \mapsto \mathbb{K}$ gives, for a process $p \in \mathcal{P}$, the committing context $\texttt{ctrl}(p)$ of the latest aci event in p.

Initializing Process. Algorithm 1 shows the initialization process. The for-loop of lines 1, 3 and 5 define the values of the initializing and committing data structures for the variables and registers together with $\nu(p, x)$, $\mu(p, x, 1)$, $\alpha(p, x, 1)$ and $\texttt{ctrl}(p)$ for all $p \in \mathcal{P}$ and $x \in \mathcal{X}$. The for-loop of line 7 defines the initial values of α and μ at the start of each context $k \geq 2$ (as described above). The for-loop of line 10 chooses an *active* process to execute in each context. The *current context* variable $\texttt{cntxt}$ is initialized to 1.

Write Instructions. Consider a write instruction $\mathfrak{i}$ in a process $p \in \mathcal{P}$ whose statement is of the form $x \leftarrow exp$. The translation of $\mathfrak{i}$ is shown in Algorithm 3. The code simulates an event $\mathbb{e}$ executing $\mathfrak{i}$, by encoding the effects of the inference rules `Init-Write`, `Com-Write` and `Prop` that initialize, commit, and propagate a write event respectively. The translation consists of three parts, namely *guessing*, *checking* and *update*.

Guessing. We guess the initializing and committing contexts for the event $\mathbb{e}$, together with its time stamp. In line 1, we guess the context in which the event $\mathbb{e}$ will be initialized, and store the guess in $\texttt{iW}(p, x)$. Similarly, in line 3, we guess the context in which the event $\mathbb{e}$ will be committed, and store the guess in $\texttt{cW}(p, x)$ (having stored its old value in the previous line). In the for-loop of line 4, we guess a time stamp for $\mathbb{e}$ and store it in β. This means that, for each process $q \in \mathcal{P}$, we guess the context in which the event $\mathbb{e}$ will be propagated to q and we store this guess in $\beta(q)$.

Checking. We perform sanity checks on the guessed values in order to verify that they are consistent with the POWER semantics. Lines 6–8 perform the sanity checks for $\texttt{iW}(p, x)$. In lines 6–7, we verify that the initializing context of the event

Alg. 1: Translating $\llbracket \texttt{iniProc} \rrbracket_{\mathbb{K}}$.

```
1  for $p \in \mathcal{P} \wedge x \in \mathcal{X}$ do
2      $\texttt{iR}(p,x) \leftarrow 1$; $\texttt{cR}(p,x) \leftarrow 1$;
       $\texttt{iW}(p,x) \leftarrow 1$; $\texttt{cW}(p,x) \leftarrow 1$;
       $\nu(p,x) \leftarrow 0$; $\mu(p,x,1) \leftarrow 0$;
       $\alpha(p,x,1) \leftarrow \otimes^{|\mathcal{P}|}$;
3  for $p \in \mathcal{P}$ do
4      $\texttt{ctrl}(p) \leftarrow 1$;
5  for $\$r \in \mathcal{R}$ do
6      $\texttt{iReg}(\$r) \leftarrow 1$; $\texttt{cReg}(\$r) \leftarrow 1$;
7  for $p \in \mathcal{P} \wedge x \in \mathcal{X} \wedge k \in [2..\mathbb{K}]$ do
8      $\alpha(p,x,k) \leftarrow \alpha^{init}(p,x,k)$;
9      $\mu(p,x,k) \leftarrow \mu^{init}(p,x,k)$;
10 for $k \in [1..\mathbb{K}]$ do
11     $\texttt{active}(k) \leftarrow \texttt{gen}(\mathcal{P})$;
12 $\texttt{cntxt} \leftarrow 1$;
```

Alg. 2: Translating $\llbracket \$r \leftarrow x \rrbracket_{\mathbb{K}}^{p,\texttt{Read}}$.

```
   // Guess
1  $\texttt{old-iR} \leftarrow \texttt{iR}(p,x)$;
2  $\texttt{iReg}(\$r) \leftarrow \texttt{iR}(p,x) \leftarrow \texttt{gen}([1..\mathbb{K}])$;
3  $\texttt{old-cR} \leftarrow \texttt{cR}(p,x)$;
4  $\texttt{cReg}(\$r) \leftarrow \texttt{cR}(p,x) \leftarrow \texttt{gen}([1..\mathbb{K}])$;
   // Check
5  $\texttt{assume}(\texttt{iR}(p,x) \geq \texttt{cntxt})$;
6  $\texttt{assume}(\texttt{active}(\texttt{iR}(p,x)) = p)$;
7  $\texttt{assume}(\texttt{iR}(p,x) \geq \texttt{iW}(p,x))$;
8  $\texttt{assume}(\texttt{iR}(p,x) \geq \texttt{cW}(p,x) \implies$
                 $\texttt{iR}(p,x) \geq$
     $\alpha(p,x,\texttt{old-iR})(p))$;
9  $\texttt{assume}(\texttt{cR}(p,x) \geq \texttt{iR}(p,x))$;
10 $\texttt{assume}(\texttt{active}(\texttt{cR}(p,x)) = p)$;
11 $\texttt{assume}(\texttt{cR}(p,x) \geq$
             $\max\{\texttt{ctrl}(p), \texttt{old-cR}, \texttt{cW}(p,x)\})$;

   // Update
12 if $\texttt{iR}(p,x) < \texttt{cW}(p,x)$ then
     $\$r \leftarrow \nu(p,x)$ ;
13 else $\$r \leftarrow \mu(p,x,\texttt{iR}(p,x))$ ;
```

Alg. 3: Translating $\llbracket x \leftarrow exp \rrbracket_{\mathbb{K}}^{p,\texttt{Write}}$.

```
   // Guess
1  $\texttt{iW}(p,x) \leftarrow \texttt{gen}([1..\mathbb{K}])$;
2  $\texttt{old-cW} \leftarrow \texttt{cW}(p,x)$;
3  $\texttt{cW}(p,x) \leftarrow \texttt{gen}([1..\mathbb{K}])$;
4  for $q \in \mathcal{P}$ do
5      $\beta(q) \leftarrow \texttt{gen}(\mathbb{K}^{\otimes})$;
   // Check
6  $\texttt{assume}(\texttt{iW}(p,x) \geq \texttt{cntxt})$;
7  $\texttt{assume}(\texttt{active}(\texttt{iW}(p,x)) = p)$;
8  $\texttt{assume}(\texttt{iW}(p,x) \geq \texttt{iReg}(exp))$;
9  $\texttt{assume}(\texttt{cW}(p,x) \geq \texttt{iW}(p,x))$;
10 $\texttt{assume}(\texttt{cW}(p,x) \geq$
     $\max\{\texttt{cReg}(exp), \texttt{ctrl}(p), \texttt{cR}(p,x), \texttt{old-cW}\})$;

11 for $q \in \mathcal{P}$ do
12     if $q = p$ then
13         $\texttt{assume}(\beta(q) = \texttt{cW}(p,x))$;
14     if $q \neq p$ then
15         $\texttt{assume}(\beta(q) \neq \otimes \implies \beta(q) \geq \texttt{cW}(p,x))$;
16     if $\beta(q) \neq \otimes$ then
17         $\texttt{assume}(\alpha(q,x,\beta(q)) \sqsubseteq \beta)$;
18         $\texttt{assume}(\texttt{active}(\beta(q)) = p)$;
   // Update
19 for $q \in \mathcal{P}$ do
20     if $\beta(q) \neq \otimes$ then
21         $\alpha(q,x,\beta(q)) \leftarrow \alpha(q,x,\beta(q)) \oplus \beta$;
22         $\mu(q,x,\beta(q)) \leftarrow exp$;
23 $\nu(p,x) \leftarrow exp$;
```

Alg. 4: Translating $\llbracket \texttt{verProc} \rrbracket_{\mathbb{K}}$.

```
1  for $p \in \mathcal{P} \wedge x \in \mathcal{X} \wedge k \in [1..\mathbb{K}-1]$ do
2      $\texttt{assume}(\alpha(p,x,k) = \alpha^{init}(p,x,k+1))$;
3      $\texttt{assume}(\mu(p,x,k) = \mu^{init}(p,x,k+1))$;
4  if $\lambda$ is reachable then error ;
```

$\mathbb{e}$ is not smaller than the current context. This captures the fact that initialization happens after fetching of $\mathbb{e}$. It also verifies that initialization happens in a context in which p is active. In line 8, we check whether `WrInitCnd` in the rule `Init-Write` is satisfied. To do that, we verify that the data dependency order $\prec_{\texttt{data}}$ holds. More precisely, we find, for each register $\$r$ that occurs in *exp*, the initializing context of the latest read event loading to $\$r$. We make sure that the initializing context of $\mathbb{e}$ is later than the initializing contexts of all these read events. By definition, the largest of all these contexts is stored in $\texttt{iReg}(exp)$.

Lines 9–10 perform the sanity checks for $\texttt{cW}(p,x)$. In line 9, we check the committing context of the event $\mathbb{e}$ is at least as large as its initializing context. In line 10, we check that `ComCnd` in the rule `Com-Write` is satisfied. To do that, we check that the committing context is larger than (i) the committing context of all the read events from which the registers in the expression *exp* fetch their values (to satisfy the data dependency order $\prec_{\texttt{data}}$, in a similar manner to that described for initialization above), (ii) the committing contexts of the latest read and write events on x in p, i.e., $\mathtt{cR}\,(p, x)$ and $\mathtt{cW}\,(p, x)$ (to satisfy the per-location program order $\prec_{\mathtt{poloc}}$), and (iii) the committing context of the latest aci event in p, i.e., $\mathtt{ctrl}\,(p)$ (to satisfy the control order $\prec_{\mathtt{ctrl}}$).

The for-loop of line 11 performs three sanity checks on the time stamp β. In line 12, we verify that the event $\mathbb{e}$ is propagated to p in the same context as the one in which it is committed. This is consistent with the rule `Com-Write` which requires that when a write event is committed then it is immediately propagated to the committing process. In line 14, we verify that if the event $\mathbb{e}$ is propagated to a process q (different from p), then the propagation takes place in a context later than or equal to the one in which $\mathbb{e}$ is committed. This is to be consistent with the fact that a write event is propagated to other processes only after it has been committed. In line 17, we check that guessed time stamp of the event $\mathbb{e}$ does not cause a violation of the coherence order $\prec_{\mathtt{co}}$. To do that, we consider each process $q \in \mathcal{P}$ to which $\mathbb{e}$ will be propagated (i.e., $\beta\,(q) \neq \otimes$). The time stamp of $\mathbb{e}$ should be larger than the time stamp of any other write event $\mathbb{e}'$ on x that has been propagated to q up to the current point (since $\mathbb{e}$ should be larger in the coherence order than $\mathbb{e}'$). Notice that by construction the time stamp of the largest such event $\mathbb{e}'$ is currently stored in $\alpha\,(q, x, \beta\,(q))$. Moreover, in line 18, we check that the event is propagated to q in a context in which p is active.

Updating. The for-loop of line 19 uses the values guessed above for updating the global data structure α. More precisely, if the event $\mathbb{e}$ is propagated to a process q, i.e., $\beta\,(q) \neq \otimes$, then we add β to the summary of the time stamps of the sequence of write operations on x propagated to q up to the current point in the context $\beta\,(q)$. Lines 22–23 assign the value *exp* to $\mu\,(p, x, \beta\,(q))$ and $\nu\,(p, x)$ respectively. Recall that the former stores the value defined by the latest write event on x propagated to q up to the current point in the context $\beta\,(q)$, and the latter stores the value defined by the latest write on x by p.

Read Instructions. Consider a read instruction $\mathfrak{i}$ in a process $p \in \mathcal{P}$ whose statement is of the form $\$r \leftarrow x$. The translation of $\mathfrak{i}$ is shown in Algorithm 2. The code simulates an event $\mathbb{e}$ running $\mathfrak{i}$ by encoding the three inference rules `Local-Read`, `Prop-Read`, and `Com-Read`. In a similar manner to a write instruction, the translation scheme for a read instruction consists of guessing, checking and update parts. Notice however that the initialization of the read event is carried out through two different inference rules.

Guessing. In line 1, we store the old value of $\mathtt{iR}\,(p, x)$. In line 2, we guess the context in which the event $\mathbb{e}$ will be initialized, and store the guessed context both in $\mathtt{iR}\,(p, x)$ and $\mathtt{iReg}\,(\$r)$. Recall that the latter records the initializing context of the latest read event loading a value to $\$r$. In lines 3–4, we execute similar instructions for committing (rather than initializing).

Checking. Lines 5–8 perform the sanity checks for $\mathtt{iR}\,(p, x)$. Lines 5–6 check that the initializing context for the event $\mathbb{e}$ is not smaller than the current context and the initialization happens in a context in which p is active. Line 7 makes sure that at least one of the two inference rules `Local-Read` and `Prop-Read` is satisfied, by checking that the closest write event $\mathtt{CW}(\mathbb{c}, \mathbb{e})$ (if it exists) has already been initialized. In line 8, we satisfy `RdCnd` in the rule `Com-Read`. Lines 9–11 perform the sanity checks for $\mathtt{cR}(p, x)$ in a similar manner to the corresponding instructions for write events (see above).

Updating. The purpose of the update part (the if-statement of line 12) is to ensure that the correct read-from relation is defined as described by the inference rules `Local-Read` and `Prop-Read`. If $\mathtt{iR}(p, x) < \mathtt{cW}(p, x)$, then this means that the latest write event $\mathbb{e}'$ on x by p is not committed and hence, according to `Local-Read`, the event $\mathbb{e}$ reads its value from that event. Recall that this value is stored in $\nu(p, x)$. On the other hand, if $\mathtt{iR}(p, x) \geq \mathtt{cW}(p, x)$ then the event $\mathbb{e}'$ has been committed and hence, according to `Prop-Read`, the event $\mathbb{e}$ reads its value from the latest write event on x propagated to p in the context where $\mathbb{e}$ is initialized. We notice that this value is stored in $\mu(p, x, \mathtt{iR}(p, x))$.

Verifier Process. The verifier process makes sure that the updated value α of the time stamp at the end of a given context $k : 1 \leq k \leq \mathbb{K} - 1$ is equal to the corresponding guessed value α^{init} at the start of the next context. It also performs the corresponding checking for the values written on the variables (by comparing μ and μ^{init}). Finally, it checks whether we reach an error label λ or not.

4 Experimental Results

In order to evaluate the efficiency of our approach, we have implemented a context-bounded model checker for programs under POWER, called power2sc[1]. We use cbmc version 5.1 [17] as the backend tool. However, observe that our code-to-code translation can be implemented on the top of any backend tool that provides safety verification of concurrent programs running under the SC semantics. In the following, we present the evaluation of power2sc on 28 C/pthreads benchmarks collected from goto-instrument [9], nidhugg [6], memorax [5], and the SV-COMP17 bechmark suit [1]. These are widespread medium-sized benchmarks that are used by many tools for analyzing concurrent programs running under weak memory models (e.g. [2–4,7,8,10,12–15,22,24,37,40]). We divide our results in two sets. The first set concerns unsafe programs while the second set concerns safe ones. In both parts, we compare results obtained from power2sc to the ones obtained from goto-instrument and nidhugg, which are, to the best of our knowledge, the only two tools supporting C/pthreads programs under POWER[2]. All experiments were run on a machine equipped with a 2.4 GHz Intel x86-32 Core2 processor and 4 GB RAM.

Table 1a shows that power2sc performs well in detecting bugs compared to the other tools for most of the unsafe examples. We observe that power2sc manages to

[1] https://www.it.uu.se/katalog/tuang296/mguess.

[2] cbmc previously supported POWER [10], but has withdrawn support in later versions.

Table 1. Comparing ③ power2sc with ① goto-instrument and ② nidhugg on two sets of benchmarks: (a) unsafe and (b) safe (with manually inserted synchronizations). The *LB* column indicates whether the tools were instructed to unroll loops up to a certain bound. The *CB* column gives the context bound for power2sc. The program size is the number of code lines. A *t/o* entry means that the tool failed to complete within 1800 s. The best running time (in seconds) for each benchmark is given in bold font.

(a)						(b)					
Program/size	LB	①	②	③		Program/size	LB	①	②	③	
		Time	Time	Time	CB			Time	Time	Time	CB
Bakery/76 [5]	8	226	t/o	**1**	3	Bakery/85 [5]	8	t/o	t/o	**70**	3
Burns/74 [5]	8	t/o	t/o	**1**	3	Burns/79 [5]	8	t/o	t/o	**1018**	3
Dekker/82 [1]	8	t/o	t/o	**1**	2	Dekker/88 [1]	8	t/o	t/o	**1158**	2
Sim Dekker/69 [5]	8	12	t/o	**1**	2	Sim Dekker/73 [5]	8	209	t/o	**14**	2
Dijkstra/82 [5]	8	t/o	t/o	**5**	3	Dijkstra/88 [5]	8	t/o	t/o	t/o	3
Szymanski/83 [1]	8	t/o	t/o	**1**	4	Szymanski/93 [1]	8	t/o	t/o	**89**	4
Fib_bench_0/36 [1]	-	**2**	1101	6	6	Fib_bench_1/36 [1]	-	9	t/o	**5**	6
Lamport/109 [1]	8	t/o	**1**	**1**	3	Lamport/119 [1]	8	t/o	t/o	t/o	3
Peterson/76 [1]	8	25	1056	**1**	3	Peterson/84 [1]	8	928	t/o	**7**	3
Peterson_3/96 [5]	8	t/o	**1**	3	4	Peterson_3/111 [5]	8	t/o	t/o	**348**	4
Pgsql/69 [9]	8	1079	**1**	**1**	2	Pgsql/73 [9]	8	1522	**2**	38	2
Pgsql_bnd/71 [6]	-	t/o	**1**	**1**	2	Pgsql_bnd/75 [6]	-	t/o	t/o	**10**	2
Tbar_2/75 [5]	8	16	**1**	**1**	3	Tbar_2/80 [5]	8	t/o	332	**29**	3
Tbar_3/94 [5]	8	104	**1**	**1**	3	Tbar_3/103 [5]	8	t/o	t/o	**138**	3

find all the errors using at most 6 contexts while nidhugg and goto-instrument time out to return the errors for several examples. This also confirms that few context switches are sufficient to find bugs. Table 1b demonstrates that our approach is also effective when we run safe programs. power2sc manages to run most of the examples (except Dijkstra and Lamport) using the same context bounds as in the case of their respective unsafe examples. While nidhugg and goto-instrument time out for several examples, they do not impose any bound on the number of context switches while power2sc does.

We have also tested the performance of power2sc with respect to the verification of small litmus tests. power2sc manages to successfully run all 913 litmus tests published in [34]. Furthermore, the output result returned by power2sc matches the ones returned by the tool herd [11] in all the litmus tests.

References

1. SV-COM17 benchmark suit (2017). https://sv-comp.sosy-lab.org/2017/benchmarks.php
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_28

3. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: CONCUR. LIPIcs, vol. 59, pp. 5:1–5:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
4. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33125-1_13
5. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_15
6. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). doi:10.1007/978-3-319-41540-6_8
7. Abdulla, P.A., Atig, M.F., Lång, M., Ngo, T.P.: Precise and sound automatic fence insertion procedure under PSO. In: Bouajjani, A., Fauconnier, H. (eds.) NETYS 2015. LNCS, vol. 9466, pp. 32–47. Springer, Cham (2015). doi:10.1007/978-3-319-26850-7_3
8. Abdulla, P.A., Atig, M.F., Ngo, T.-P.: The best of both worlds: trading efficiency and optimality in fence insertion for TSO. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 308–332. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46669-8_13
9. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37036-6_28
10. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_9
11. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM TOPLAS **36**(2), 7:1–7:74 (2014)
12. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_9
13. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37036-6_29
14. 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)
15. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_12
16. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA, pp. 122–132. ACM (2011)
17. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
18. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 84–104. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_7

19. Dan, A., Meshman, Y., Vechev, M., Yahav, E.: Effective abstractions for verification under relaxed memory models. Comput. Lang. Syst. Struct. **47**(Part 1), 62–76 (2017)
20. Demsky, B., Lam, P.: Satcheck: sat-directed stateless model checking for SC and TSO. In: OOPSLA 2015, pp. 20–36. ACM (2015)
21. Derevenetc, E., Meyer, R.: Robustness against power is PSpace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8573, pp. 158–170. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43951-7_14
22. Huang, S., Huang, J.: Maximal causality reduction for TSO and PSO. In: OOPSLA 2016, pp. 447–461 (2016)
23. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: FMCAD, pp. 111–119. IEEE (2010)
24. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI, pp. 187–198. ACM (2011)
25. Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_36
26. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 479–495. Springer, Cham (2016). doi:10.1007/978-3-319-48989-6_29
27. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. FMSD **35**(1), 73–97 (2009)
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **C–28**(9), 690–691 (1979)
29. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI 2012, pp. 429–440. ACM (2012)
30. 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: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_36
31. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM (2007)
32. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03359-9_27
33. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31980-1_7
34. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI, pp. 175–186. ACM (2011)
35. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. CACM **53**, 89–97 (2010)
36. Tomasco, E., Lam, T.N., Fischer, B., La Torre, S., Parlato, G.: Embedding weak memory models within eager sequentialization (2016). http://eprints.soton.ac.uk/402285/

37. Tomasco, E., Lam, T.N., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: FMCAD 2016, pp. 193–200 (2016)
38. Travkin, O., Wehrheim, H.: Verification of concurrent programs on weak memory models. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 3–24. Springer, Cham (2016). doi:10.1007/978-3-319-46750-4_1
39. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: IPDPS. IEEE (2004)
40. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI, pp. 250–259. ACM (2015)