Portability Analysis for Weak Memory Models
PORTHOS: One *Tool* for all *Models*

Hernán Ponce-de-León[1⋆], Florian Furbach[2], Keijo Heljanko[3], and Roland Meyer[4⋆]

[1]fortiss GmbH, Germany [2]TU Kaiserslautern, Germany [3]Aalto University and HIIT, Finland [4]TU Braunschweig, Germany
ponce@fortiss.org, furbach@cs.uni-kl.de, keijo.heljanko@aalto.fi, roland.meyer@tu-braunschweig.de

Abstract. We present PORTHOS, the first tool that discovers porting bugs in performance-critical code. PORTHOS takes as input a program and the memory models of the source architecture for which the program has been developed and the target model to which it is ported. If the code is not portable, PORTHOS finds a bug in the form of an unexpected execution — an execution that is consistent with the target but inconsistent with the source memory model. Technically, PORTHOS implements a bounded model checking method that reduces the portability analysis problem to satisfiability modulo theories (SMT). There are two main problems in the reduction that we present novel and efficient solutions for. First, the formulation of the portability problem contains a quantifier alternation (consistent + inconsistent). We introduce a formula that encodes both in a single existential query. Second, the supported memory models (e.g., Power) contain recursive definitions. We compute the required least fixed point semantics for recursion (a problem that was left open in [48]) efficiently in SMT. Finally we present the first experimental analysis of portability from TSO to Power.

1 Introduction

Porting code from one architecture to another is a routine task in system development. Given that no functionality has to be added, porting is rarely considered interesting from a programming point of view. At the same time, porting is non-trivial as the hardware influences both the semantics and the compilation of the code in subtle ways. The unfortunate combination of being routine and yet subtle makes porting prone to mistakes. This is particularly true for performance-critical code that interacts closely with the execution environment. Such code often has data races and thus exposes the programmer to the details of the underlying hardware. When the architecture is changed, the code may have to be adapted to the primitives of the target hardware.

We tackle the problem of porting performance-critical code among hardware architectures. Our contribution is the new (and to the best of our knowledge first) tool PORTHOS to fight porting bugs. The tool takes as input a piece of

⋆ This work was carried out when the author was at Aalto University.

code, a model of the source architecture for which the code has been developed, and a model of the target architecture to which the code is to be ported. PORTHOS automatically checks whether every behaviour of the code on the target architecture is also allowed on the source platform. This guarantees that correctness of the program in terms of safety properties (in particular properties like mutual exclusion) carries over to the targeted hardware, and the program remains correct after porting.

Portability requires an analysis method that is hardware-architecture-aware in the sense that a description of the memory models of source and target platforms has to be part of the input. A language for memory models, called CAT [4], has been developed only recently. In CAT, memory models are defined in terms of relations between memory operations of a program. There are some base relations (program order, reads from, coherence) that are common to all memory models. A memory model may define further so-called derived relations by restricting and composing base relations. The memory model specifies axioms in the form of acyclicity and irreflexivity constraints over relations. An execution is consistent if it satisfies all axioms. Our work builds on the CAT language.

There are three problems that make portability different from most common verification tasks.

(i) We have to deal with user-defined memory models. These models may define derived relations as least fixed points.
(ii) The formulation of portability involves an alternation (consistent + inconsistent) of quantifiers.
(iii) High-level code may be compiled into different low-level code depending on the architecture (see, e.g., Fig. 1).

Concerning the first problem, we implement in SMT the operations that CAT defines on relations. Notably, we propose an encoding for derived relations that are defined as least fixed points. Such least fixed points are prominently used in the Power memory model [8] and their computation was identified as a key problem in [48]. To quote the authors *[...] the proper fixpoint construction [...] is much more expensive than a fixed unrolling.* We show that, with our encoding, this is not the case. A naive approach would implement the Kleene iteration in SAT by introducing copies of the variables for each iteration step, resulting in a very large encoding. We show how to employ SAT + integer difference logic [19] to compactly encode the Kleene iteration process. Notably, every bounded model checking technique reasoning about complex memory models defined in CAT (e.g., Power) will face the problem of dealing with recursive definitions and can make use of our technique to solve it efficiently.

The second problem is to encode the quantifier alternation underlying the definition of portability. A porting bug is an execution that is consistent with the target but inconsistent with the source memory model. We capture this alternation with a single existential query. Consistency is specified in terms of acyclicity (and irreflexivity) of relations. Hence, an execution is inconsistent if a derived relation of the (source) memory model contains a cycle (or is not irreflexive).

The naive idea would be to model cyclicity by unsatisfiability. Instead, we reduce cyclicity to satisfiability by introducing auxiliary variables that guess the cycle.

The reader may criticise our definition of portability: one could claim that all that matters is whether safety is preserved, even if the executions differ. To be precise, a state-based notion of portability requires that every state computable under the target architecture is already computable on the source platform. We study state portability and come up with two results.

(a) Algorithmically, state portability is beyond SAT.
(b) Empirically, there is little difference between state portability and our notion.

The third problem is that the same high-level program is compiled to different assembly programs depending on the source and the target architectures. Even the number of registers and the semantics of the synchronisation primitives provided by those architectures usually differ. Consider the program from Fig. 1, written in C++11 and compiled to x86 and Power. The observation is this. Even if the assembly programs differ, one can map every assembly memory access to the corresponding read or write operation in the high-level code. In the example, clearly "`MOV [y],$1`" and "`stw r1,y`" correspond to "`y.store(memory_order_relaxed, 1)`". This allows us to relate low-level and high-level executions and to compare executions of both assembly programs by checking if they map to the same high-level execution. With this observation, our analysis can be extended by translating an input program into two corresponding assembly programs and making explicit the relation among the low-level and high-level executions. While this relation among executions is not studied in the present paper, details of how to construct it and how to incorporate it into our approach can be found in [38].

In summary, we make the following contributions.

1. We present the first SMT-based implementation of a core subset of CAT which can handle recursive definitions efficiently.
2. We formulate the portability problem based on the CAT language.
3. We develop a bounded analysis for portability. Despite the apparent alternation of quantifiers, our SMT encoding is a satisfiability query of polynomial size and optimal in the complexity sense.
4. We compare our notion of portability to a state-based notion and show that the latter does not afford a polynomial SAT encoding.
5. We present experiments showing that *(i)* in a large majority of cases both notions of portability coincide, and *(ii)* mutual exclusion algorithms are often not portable, particularly we perform the first analysis from TSO to Power.

2 Portability Analysis on an Example

Consider program **IRIW** in Fig. 1, written in C++11 and using the atomic operator `memory_order_relaxed` which provides no guarantees on how memory

```
thread t0                                  thread t1
y.store(memory_order_relaxed, 1)           x.store(memory_order_relaxed, 1)

thread t2                                  thread t3
r1 = x.load(memory_order_relaxed);         r1 = y.load(memory_order_relaxed);
r2 = y.load(memory_order_relaxed)          r2 = x.load(memory_order_relaxed)
```

x86 Assembly

thread t_0	thread t_1	thread t_2	thread t_3
MOV [y],$1	MOV [x],$1	MOV EAX,[x]	MOV EAX,[y]
		MOV EAX,[y]	MOV EAX,[x]

Power Assembly

thread t_0	thread t_1	thread t_2	thread t_3
li r1,1	li r1,1	lwz r1,x	lwz r1,y
stw r1,y	stw r1,x	lwz r3,y	lwz r3,x

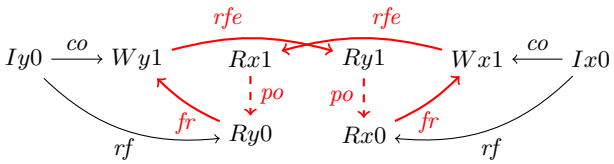

Fig. 1: Portability of program **IRIW** from TSO to Power.

accesses are ordered. When porting, the program is compiled to two different architectures. The corresponding low-level programs behave differently on x86 and on IBM's Power. On TSO, the memory model implemented by x86, each thread has a store buffer of pending writes. A thread can see its own writes before they become visible to other threads (by reading them from its buffer), but once a write hits the memory it becomes visible to all other threads simultaneously: TSO is a multi-copy-atomic model [18]. Power on the other hand does not guarantee that writes become visible to all threads at the same point in time. Think of each thread as having its own copy of the memory. With these two architectures in mind, consider the execution in Fig. 1. Thread t_2 reads $x = 1, y = 0$ and thread t_3 reads $x = 0, y = 1$, indicated by the solid edges *rfe* and *rf*. Since under TSO every execution has a unique global view of all operations, no interleaving allows both threads to read the above values of the variables. Under Power, this is possible. Our goal is to automatically detect such differences when porting a program from one architecture to another, here from TSO to Power.

Our tool PORTHOS applies to various architectures, and we not only have a language for programs but also a *language for memory models*. The semantics of a program on a memory model is defined axiomatically, following two steps [8,48]. We first associate with the program (and independent of the memory model) a set of executions which are candidates for the semantics. An execution is a graph (Fig. 1) whose nodes (events) are program instructions and whose edges are basic dependencies: the program order *po*, the reads-from relation *rf* (giving

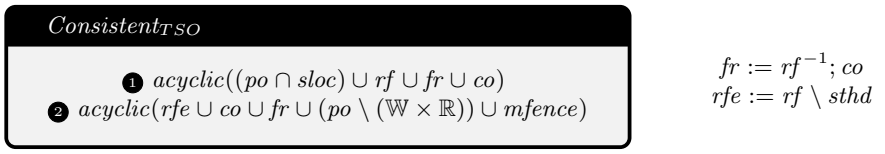

Fig. 2: TSO.

the write that a load reads from), and the coherence order *co* (stating the order in which writes take effect). The memory model then defines which executions are consistent and thus form the semantics of the program on that model.

We describe memory models in the recently proposed language CAT [4]. Besides the base relations, a model may define so-called derived relations. The consistency requirements are stated in terms of acyclicity and irreflexivity axioms over these (base and derived) relations. The CAT formalisation of TSO is given in Fig. 2. It forbids executions forming a cycle over $rfe \cup fr \cup (po \setminus (\mathbb{W} \times \mathbb{R}))$. The red edges in Fig. 1 yield such a cycle; the execution is not consistent with TSO. Power further relaxes the program order (Fig. 6), the relations denoted by the dotted lines are no longer considered for cycles and thus the execution is consistent. Hence, **IRIW** has executions consistent with Power but not with TSO and is therefore not portable.

Our contribution is a bounded analysis for portability implemented in the PORTHOS tool (`http://github.com/hernanponcedeleon/PORTHOS`). First, the program is unrolled up to a user-specified bound. Within this bound, PORTHOS is guaranteed to find all portability bugs. It will neither see bugs beyond the bound nor will it be able to prove a cyclic program portable. The unrolled program, together with the CAT models, is transformed into an SMT formula where satisfying assignments correspond to bugs.

A bug is an execution consistent with the target memory model $\mathcal{M}_T$ but inconsistent with the source $\mathcal{M}_S$. We express this combination of consistency and inconsistency with only one existential quantification. The key observation is that the derived relations, which may differ in $\mathcal{M}_T$ and $\mathcal{M}_S$, are fully defined by the execution. Hence, by guessing an execution we also obtain the derived relations (there is nothing more to guess). Checking consistency for $\mathcal{M}_T$ is then an acyclicity (or irreflexivity) constraint on the derived relations that immediately yields an SMT query. Inconsistency for $\mathcal{M}_S$ requires cyclicity. The trick is to explicitly guess the cycle. We introduce Boolean variables for every event and every edge that could be part of the cycle. In Fig. 1, if $Rx1$ is on the cycle, indicated by the variable $\mathtt{C}(Rx1)$ being set, then there should be one incoming and one outgoing edge also in the cycle. Besides the incoming edge shown in the graph, $Rx1$ could read from the initial value $Ix0$. Since there are two possible incoming edges but only one outgoing edge, we obtain $\mathtt{C}(Rx1) \Rightarrow ((\mathtt{C}_{rfe}(Wx1, Rx1) \vee \mathtt{C}_{rf}(Ix0, Rx1)) \wedge \mathtt{C}_{po}(Rx1, Ry0))$. If a relation is on the cycle, then also both end-points should be part of the cycle and the relation should belong to the execution: $\mathtt{C}_{po}(Rx1, Ry0) \Rightarrow (\mathtt{C}(Rx1) \wedge \mathtt{C}(Ry0) \wedge \mathtt{po}(Rx1, Ry0))$. Finally, at least one event has to be part of

the cycle: $\mathtt{C}(Ix0) \vee \mathtt{C}(Wx1) \vee \mathtt{C}(Rx1) \vee \mathtt{C}(Rx0) \vee \mathtt{C}(Iy0) \vee \mathtt{C}(Wy1) \vee \mathtt{C}(Ry1) \vee \mathtt{C}(Ry0)$. The execution in Fig. 1 contains the relations marked in red and forms a cycle which violates Axiom ❷ in TSO. The execution respects the axioms of Power (Fig. 6), showing the existence of a portability bug in **IRIW** from TSO to Power.

The other challenge is to capture relations that are defined recursively. The Kleene iteration process [43] starts with the empty relation and repeatedly adds pairs of events according to the recursive definitions. We encode this into (quantifier-free) integer difference logic [19]. For every recursive relation $\mathtt{r}$ and every pair of events (e_1, e_2), we introduce an integer variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the iteration step in which the pair entered the value of $\mathtt{r}$. A Kleene iteration then corresponds to a total ordering on these integer variables. Crucially, we only have one Boolean variable $\mathtt{r}(e_1, e_2)$ per pair rather than one per iteration step. We illustrate the encoding on a simplified version of the preserved program order for Power defined as $ppo := ii \cup ic$ (cf. Fig. 6 for the full definition). The relation is derived from the mutually recursive relations $ii := dd \cup ic$ and $ic := cd \cup ii$, where dd and cd represent data and control dependencies. Call $Rx1$ and $Ry0$ respectively e_1 and e_2. The encoding is

$$\mathtt{ii}(e_1, e_2) \Leftrightarrow (\mathtt{dd}(e_1, e_2) \wedge (\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{dd}}_{e_1,e_2})) \vee (\mathtt{ic}(e_1, e_2) \wedge (\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{ic}}_{e_1,e_2}))$$

$$\mathtt{ic}(e_1, e_2) \Leftrightarrow (\mathtt{cd}(e_1, e_2) \wedge (\Phi^{\mathtt{ic}}_{e_1,e_2} > \Phi^{\mathtt{cd}}_{e_1,e_2})) \vee (\mathtt{ii}(e_1, e_2) \wedge (\Phi^{\mathtt{ic}}_{e_1,e_2} > \Phi^{\mathtt{ii}}_{e_1,e_2})).$$

The pair (e_1, e_2) that belongs to relation dd in step $\Phi^{\mathtt{dd}}_{e_1,e_2}$ of the Kleene iteration can be added to relation ii at a later step $\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{dd}}_{e_1,e_2}$. As $ii := dd \cup ic$, the disjunction allows us to also add the elements of ic to ii. Since dd and cd are empty for **IRIW**, the relations ii and ic have to be identical. Identical non-empty relations will not yield a solution: the integer variables cannot satisfy $(\Phi^{\mathtt{ii}}_{e_1,e_2} > \Phi^{\mathtt{ic}}_{e_1,e_2})$ and $(\Phi^{\mathtt{ic}}_{e_1,e_2} > \Phi^{\mathtt{ii}}_{e_1,e_2})$ at the same time. Hence, the only satisfying assignment is the one where both ii and ic are the empty relation, which implies that ppo is empty. This is consistent with the preserved program order of Power for **IRIW**.

3 Programs and Memory Models

We introduce our language for programs and the core of the language CAT. The presentation follows [4,48] and we refer the reader to those works for details.

Programs. Our language for shared memory concurrent programs is given in Fig. 3. Programs consist of a finite number of threads from a while-language. The threads operate on assembly level, which means they explicitly read from the shared memory into registers, write from registers into memory, and support local computations on the registers. The language has various fence instructions (sync, lwsync, and isync on Power and mfence on x86) that enforce ordering and visibility constraints among instructions. We refrain from explicitly defining the expressions and predicates used in assignments and conditionals. They will depend on the data domain. For our analysis, we only require the domain to

$$
\begin{aligned}
\langle prog \rangle &::= \texttt{program}\ \langle thrd \rangle^* \\
\langle thrd \rangle &::= \texttt{thread}\ \langle tid \rangle\ \langle inst \rangle \\
\langle inst \rangle &::= \langle atom \rangle \mid \langle inst \rangle; \langle inst \rangle \\
&\mid \texttt{while}\ \langle pred \rangle\ \langle inst \rangle \\
&\mid \texttt{if}\ \langle pred \rangle\ \texttt{then}\ \langle inst \rangle \\
&\qquad\qquad \texttt{else}\ \langle inst \rangle \\
\langle atom \rangle &::= \langle reg \rangle \leftarrow \langle exp \rangle \mid \langle reg \rangle \leftarrow \langle loc \rangle \\
&\mid \langle loc \rangle := \langle reg \rangle \mid \langle mfence \rangle \\
&\mid \langle sync \rangle \mid \langle lwsync \rangle \mid \langle isync \rangle
\end{aligned}
$$

Fig. 3: Programming language.

$$
\begin{aligned}
\langle MCM \rangle &::= \langle assert \rangle \mid \langle rel \rangle \mid \langle MCM \rangle \wedge \langle MCM \rangle \\
\langle assert \rangle &::= acyclic(\langle r \rangle) \mid irreflexive(\langle r \rangle) \\
\langle r \rangle &::= \langle b \rangle \mid \langle r \rangle \cup \langle r \rangle \mid \langle r \rangle \cap \langle r \rangle \mid \langle r \rangle \setminus \langle r \rangle \\
&\mid \langle r \rangle^{-1} \mid \langle r \rangle^{+} \mid \langle r \rangle^{*} \mid \langle r \rangle; \langle r \rangle \\
\langle b \rangle &::= po \mid rf \mid co \mid ad \mid dd \mid cd \mid sthd \mid sloc \\
&\mid mfence \mid sync \mid lwsync \mid isync \\
&\mid id(\langle set \rangle) \mid \langle set \rangle \times \langle set \rangle \mid \langle name \rangle \\
\langle set \rangle &::= \mathbb{E} \mid \mathbb{W} \mid \mathbb{R} \\
\langle rel \rangle &::= \langle name \rangle := \langle r \rangle
\end{aligned}
$$

Fig. 4: Core of CAT [4].

admit an SMT encoding in a logic which has its satisfiability problem in NP. For the rest of the paper we will assume that programs are acyclic: any while statement is removed by unrolling the program to a depth specified by the user. Since verification is generally undecidable for while-programs [39], this under-approximation is necessary for cyclic programs.

Executions. The semantics of a program is given in terms of *executions*, partial orders where the events represent occurrences of the instructions and the ordering edges represent dependencies. The definition is given in Fig. 5. An execution consists of a set $\mathbb{X}$ of executed events and so-called *base* and *induced relations* satisfying the Axioms ❸-⓲. Base relations rf and co and the set $\mathbb{X}$ define an execution (they are the ones to be guessed). Induced relations can be extracted directly from the source code of the program. The axioms in Fig. 5 are common to all memory models and natively implemented by our tool. To state them, let $\mathbb{E}$ represents memory events coming from program instructions accessing the memory. Memory accesses are either reads or writes $\mathbb{E} := \mathbb{R} \cup \mathbb{W}$. By $\mathbb{R}_l$ and $\mathbb{W}_l$ we refer respectively to the reads and writes that access location l. The events of thread t form the set $\mathbb{E}_t$. Relations $sthd$ and $sloc$ are equivalences relating events belonging to the same thread ❸ and accessing the same location ❹. Relations po, ad, dd and cd represent program order and address/data/control dependencies. Axiom ❺ states that the *program order* po is an intra-thread relation which ❻ forms a total order when projected to events in the same thread (predicate $total(r, A)$ holds if r is a total order on the set A). *Address dependencies* are either read-to-read or read-to-write ❼, *data dependencies* are read-to-write ❽, and *control dependencies* originate from reads ❾. Fence relations are architecture specific and relate only events in program order ❿-⓭. Axiom ⓮, which we do not make explicit, requires the executed events $\mathbb{X}$ to form a path in the threads' control flow. By Axioms ⓯ and ⓰, the *reads-from relation* rf gives for each read a unique write to the same location from which the read obtains its value. Here, $r_1; r_2 := \{(x, y) \mid \exists z : (x, z) \in r_1 \text{ and } (z, y) \in r_2\}$ is the composition of the relations r_1 and r_2. We write $r^{-1} := \{(y, x) \mid (x, y) \in r\}$ for the inverse

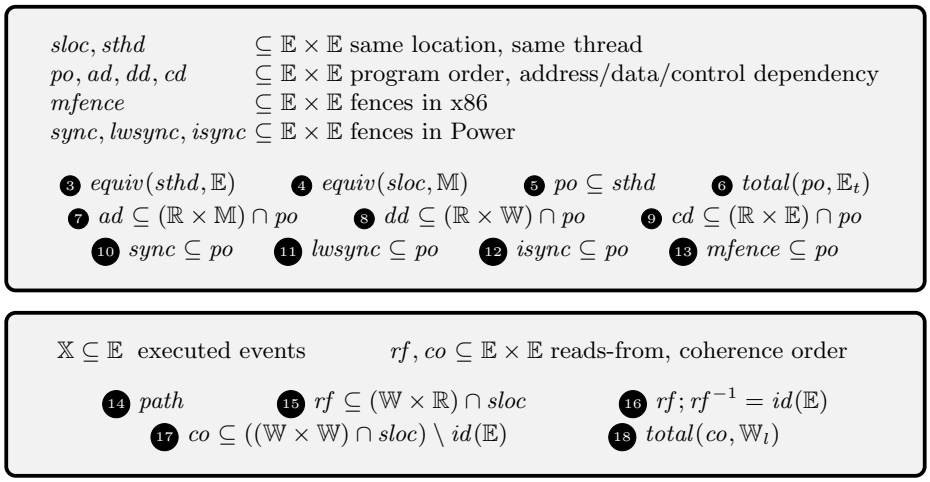

Fig. 5: Executions; adapted from [48].

of relation r. Finally, $id(A)$ is the identity relation on A. By Axioms 17 and 18, the *coherence relation co* relates writes to the same location, and it forms a total order for each location. We will assume the existence of an initial write event for each location which assigns value 0 to the location. This event is first in the coherence order.

Memory Consistency Models. We give in Fig. 4 a core subset of the CAT language for memory consistency models (MCMs). A *memory model* is a constraint system over so-called *derived relations*. Derived relations are built from the base and induced relations in an execution, hand-defined relations that refer to the different sets of events, and named relations that we will explain in a moment. The assertions are acyclicity and irreflexivity constraints over derived relations. CAT also supports recursive definitions of relations. We assume a set $\langle name \rangle$ of relation names (different from the predefined relations) and require that each name used in the memory model has associated a defining equation $\langle name \rangle := \langle r \rangle$. Notably, $\langle r \rangle$ may again contain relation names, making the system of defining equations recursive. The actual relations that are denoted by the names are defined to be the least solution to this system of equations. We can compute the least solution with a standard Kleene iteration [43] starting from the empty relations and iterating until the least fixed point is reached.

In Section 6 we study portability to Power; we use its formalization [8] in the core of CAT as given in Fig. 6. Power is a highly relaxed memory model that supports program-order relaxations depending on address and data dependencies, that is not multi-copy atomic, and that has a complex set of fence instructions. The axioms defining Power are uniproc 1 and the constraints 19 to 21. The model relies on the recursively defined relations ii, ci, ic, and cc.

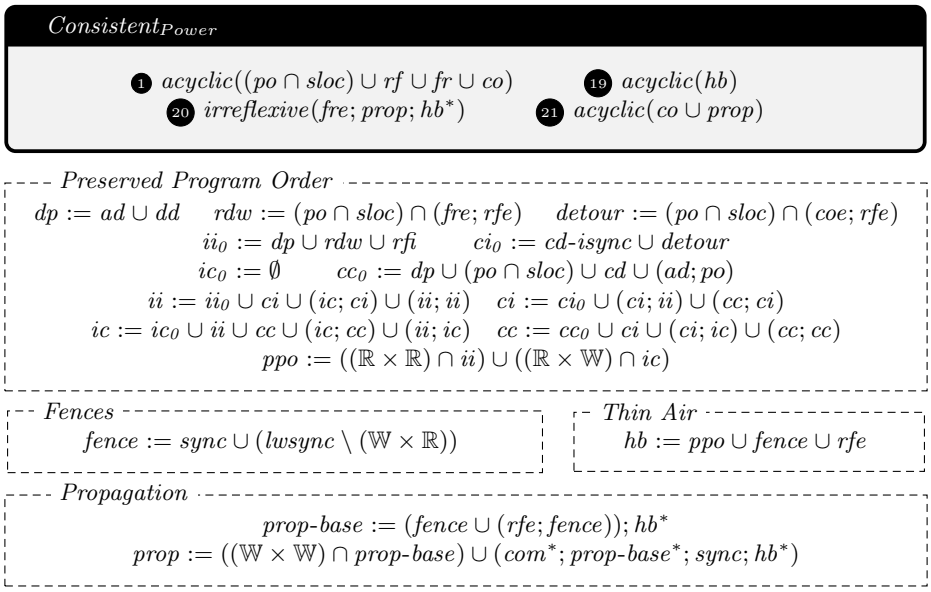

Fig. 6: Power [8].

4 Portability Analysis

Let $cons_{\mathcal{M}}(P)$ be the set of executions of program P consistent with $\mathcal{M}$. Given a program P and two MCMs $\mathcal{M}_S$ and $\mathcal{M}_T$, our goal is to find an execution X which is consistent with the target ($X \in cons_{\mathcal{M}_T}(P)$) but not with the source ($X \notin cons_{\mathcal{M}_S}(P)$). In such a case P is not portable from $\mathcal{M}_S$ to $\mathcal{M}_T$.

Definition 1 (Portability). *Let $\mathcal{M}_S$, $\mathcal{M}_T$ be two MCMs. A program P is* portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ *if $cons_{\mathcal{M}_T}(P) \subseteq cons_{\mathcal{M}_S}(P)$.*

Our method finds non-portable executions as satisfying assignments to an SMT formula. Recall that an execution is uniquely represented by the set $\mathbb{X}$ and the relations rf and co, which need to be guessed by the solver. All other relations are derived from these guesses, the source code of the program, and the MCMs in question. Therefore, we also have to encode the derived relations of the two MCMs defined in the language of Fig. 4. As the last part, we encode the assertions expressed in the language of Fig. 4 on these relations in such a way that the guessed execution is allowed by $\mathcal{M}_T$ (all the assertions stated for $\mathcal{M}_T$ hold) while the same execution is not allowed by $\mathcal{M}_S$ (at least one of the axioms of $\mathcal{M}_S$ is violated). The full SMT formula is of the form $\phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}_T} \wedge \phi_{\neg\mathcal{M}_S}$. Here, ϕ_{CF} and ϕ_{DF} encode the control flow and data flow of the executions, $\phi_{\mathcal{M}_T}$ encodes the derived relations and all assertions of $\mathcal{M}_T$, and $\phi_{\neg\mathcal{M}_S}$ encodes the derived relations of $\mathcal{M}_S$ and a violation of at least one of the assertions of the source memory model. The control-flow and data-flow encodings are standard for bounded model checking [17]. The rest of the section focuses on how to encode the derived relations needed for representing both MCMs, how to encode assertions

for the target memory model and how to encode an assertion violation in the source memory model. The encoding for assertions in the target memory model and the encoding for most of the relations is similar to [6], the most notable difference being that they do not discuss how to handle mutually recursively defined relations while we do so in an efficient way.

Encoding Derived Relations. For any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we use a Boolean variable $\mathtt{r}(e_1, e_2)$ representing the fact that $e_1 \xrightarrow{r} e_2$ holds. We similarly use fresh Boolean variables to represent the derived relations, using the encoding to force their values as follows. For the union (resp. intersection) of two relations, at least one of them (resp. both of them) should hold; set difference requires that the first relation holds and the second one does not; for the composition of relations we iterate over a third event and check if it belongs to the range of the first relation and the domain of the second. Computing a reverse relation requires reversing the events. We define the transitive closure of r recursively where the base case tc_0 holds if events are related according to r and the recursive case uses a relation composition. This is computed with the iterative squaring technique using the relation composition. Finally reflexive and transitive closure checks if the events are the same or are related by r^+. The encodings are summarized below.

$$\mathtt{r}_1 \cup \mathtt{r}_2(e_1, e_2) \Leftrightarrow \mathtt{r}_1(e_1, e_2) \vee \mathtt{r}_2(e_1, e_2) \qquad \mathtt{r}_1 \cap \mathtt{r}_2(e_1, e_2) \Leftrightarrow \mathtt{r}_1(e_1, e_2) \wedge \mathtt{r}_2(e_1, e_2)$$
$$\mathtt{r}_1 \setminus \mathtt{r}_2(e_1, e_2) \Leftrightarrow \mathtt{r}_1(e_1, e_2) \wedge \neg \mathtt{r}_2(e_1, e_2) \qquad \mathtt{r}^{-1}(e_1, e_2) \Leftrightarrow \mathtt{r}(e_2, e_1)$$
$$\mathtt{r}_1 ; \mathtt{r}_2(e_1, e_2) \Leftrightarrow \bigvee_{e_3 \in \mathbb{E}} \mathtt{r}_1(e_1, e_3) \wedge \mathtt{r}_2(e_3, e_2) \qquad \mathtt{r}^*(e_1, e_2) \Leftrightarrow \mathtt{r}^+(e_1, e_2) \vee (e_1 = e_2)$$

$$\mathtt{r}^+(e_1, e_2) \Leftrightarrow \mathtt{tc}_{\lceil \log |\mathbb{E}| \rceil}(e_1, e_2), \textit{where}$$
$$\mathtt{tc}_0(e_1, e_2) \Leftrightarrow \mathtt{r}(e_1, e_2), \textit{and}$$
$$\mathtt{tc}_{i+1}(e_1, e_2) \Leftrightarrow \mathtt{r}(e_1, e_2) \vee (\mathtt{tc}_i ; \mathtt{tc}_i(e_1, e_2)).$$

Recall that some of the relations (e.g., *ii* and *ic* of Power) can be defined mutually recursively, and that we are using the least fixed point (smallest solution) semantics for cyclic definitions. A classical algorithm for solving such equations is the Kleene fixpoint iteration. The iteration starts from the empty relations as initial approximation and on each round computes a new approximation until the (least) fixed point is reached. Such an iterative algorithm can be easily encoded into SAT. The problem of such an encoding is the potentially large number of iterations needed, and thus the resulting formula size can grow to be very large. A more clever way to encode this is an approach that has been already used in earlier work on encoding mutually recursive monotone equation systems with nested least and greatest fixpoints [30]. The encoding of this paper uses an extension of SAT with integer difference logic (IDL), a logic that is still NP complete. A SAT encoding is also possible but incurs an overhead in the encoding size: if the SMT encoding is of size $O(n)$, the SAT encoding is of size $O(n \log n)$ [30]. We chose IDL since our experiments showed the encoding to be the most time consuming of the tasks.

Here, the basic idea is to guess a certificate that contains the iteration number in which a pair would be added to the relation in the Kleene iteration. For

this we use additional integer variables and enforce that they locally follow the propagations made by the fixed point iteration algorithm. Thus, for any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we introduce an integer variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the round in which $\mathtt{r}(e_1, e_2)$ would be set by the Kleene iteration algorithm. Using these new variables we guess the execution of the Kleene fixed point iteration algorithm, and then locally check that every guess that was made is also a valid propagation of the fixed point iteration algorithm. To give an example, consider a definition where $r_1 := r_2 \cup r_3$ and $r_2 := r_1 \cup r_4$. The encoding is as follows

$$\mathtt{r}_1(e_1, e_2) \Leftrightarrow (\mathtt{r}_2(e_1, e_2) \wedge (\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})) \vee (\mathtt{r}_3(e_1, e_2) \wedge (\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_3}_{e_1,e_2}))$$
$$\mathtt{r}_2(e_1, e_2) \Leftrightarrow (\mathtt{r}_1(e_1, e_2) \wedge (\Phi^{\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_1}_{e_1,e_2})) \vee (\mathtt{r}_4(e_1, e_2) \wedge (\Phi^{\mathtt{r}_2}_{e_1,e_2} > \Phi^{\mathtt{r}_4}_{e_1,e_2})).$$

A pair (e_1, e_2) is added to r_1 by the Kleene iteration in step $\Phi^{\mathtt{r}_1}_{e_1,e_2}$. It comes from either r_2 or r_3. If it came from r_2 then it is of course also in r_2 and it was added to r_2 in an earlier iteration $\Phi^{\mathtt{r}_2}_{e_1,e_2}$ and thus $(\Phi^{\mathtt{r}_1}_{e_1,e_2} > \Phi^{\mathtt{r}_2}_{e_1,e_2})$. It is similar if it came from r_3. The only satisfying assignment for the encoding is one where both r_1 and r_2 are the union of r_3 and r_4.

Encoding Target MCM Assertions. For the target architecture we need to encode all acyclicity and irreflexivity assertions of the memory model. For handling acyclicity we again use non-Boolean variables in our SMT encoding for compactness reasons. One can encode that a relation is acyclic by adding a numerical variable $\Psi_e \in \mathbb{N}$ for each event e in the relation we want to be acyclic. Then acyclicity of relation r is encoded as $\mathit{acyclic}(r) \Leftrightarrow \bigwedge_{e_1,e_2 \in \mathbb{E}} (\mathtt{r}(e_1, e_2) \Rightarrow (\Psi_{e_1} < \Psi_{e_2}))$. Notice that we can impose a total order with all $\Psi_{e_1} < \Psi_{e_2}$ constraints iff there is no cycle. Our encoding is the same as the SAT + IDL encoding in [28] where more discussion of SAT modulo acyclicity can be found. The irreflexive constraint is simply encoded as: $\mathit{irreflexive}(r) \Leftrightarrow \bigwedge_{e \in \mathbb{E}} \neg \mathtt{r}(e, e)$.

Encoding Source MCM Assertions. For the source architecture we have to encode that one of the derived relations does not fulfill its assertions. On the top level this can be encoded as a simple disjunction over all the assertions of the source memory model, forcing at least one of the irreflexivity or acyclicity constraints to be violated.

For the irreflexivity violation, we can reuse the same encoding as for the target memory model simply as $\neg \mathit{irreflexive}(r)$. What remains to be encoded is $\mathit{cyclic}(r)$, which requires the relation r to be cyclic. Here, we give an encoding that uses only Boolean variables. We add Boolean variables $\mathtt{C}(e)$ and $\mathtt{C}_{\mathtt{r}}(e_1, e_2)$, which guess the edges and nodes constituting the cycle. We ensure that for every event in the cycle, there should be at least one incoming edge and at least one outgoing edge that are also in the cycle:

$$c_n = \bigwedge_{e_1 \in \mathbb{E}} (\mathtt{C}(e_1) \Rightarrow (\bigvee_{e_2 \xrightarrow{r} e_1} \mathtt{C}_{\mathrm{r}}(e_2, e_1) \wedge \bigvee_{e_1 \xrightarrow{r} e_2)} \mathtt{C}_{\mathrm{r}}(e_1, e_2))).$$

If an edge is guessed to be in a cycle, the edge must belong to relation r, and both events must also be guessed to be on the cycle:

$$c_e = \bigwedge_{e_1,e_2 \in \mathbb{E}} (\mathtt{C_r}(e_1,e_2) \Rightarrow (\mathtt{r}(e_1,e_2) \wedge \mathtt{C}(e_1) \wedge \mathtt{C}(e_2))).$$

A cycle exists, if these formulas hold and there is an event in the cycle:

$$\mathit{cyclic}(r) \Leftrightarrow (c_e \wedge c_n \wedge \bigvee_{e \in \mathbb{E}} \mathtt{C}(e)).$$

5 State Portability

Portability from $\mathcal{M}_S$ to $\mathcal{M}_T$ requires that there are no new executions in $\mathcal{M}_T$ that did not occur in $\mathcal{M}_S$. One motivation to check portability is to make sure that safety properties of $\mathcal{M}_S$ carry over to $\mathcal{M}_T$. Safety properties only depend on the values that can be computed, not on the actual executions. Therefore, we now study a more liberal notion of so-called *state portability*: $\mathcal{M}_T$ may admit new executions as long as they do not compute new states. Admitting more executions means we require less synchronization (fences) to consider a ported program correct, and thus state portability promises more efficient code. This notion has been used in [31].

The main finding in this section is negative: a polynomial encoding of state portability to SAT does not exist (unless the polynomial hierarchy collapses). Phrased differently, state portability does not admit an efficient bounded analysis (like our method for portability). Fortunately, our experiments indicate that new executions often compute new states. This means portability is not only a sufficient condition for state portability but, in practice, the two are equivalent. Combined with the better algorithmics of portability, we do not see a good motivation to move to state portability. Proofs of all stated results can be found in [38]. We remind the reader that we restrict our input to acyclic programs (that can be obtained from while-programs with bounded unrolling); for while-programs, verification tasks are generally undecidable [39].

A state is a function that assigns a value to each location and register. An execution X computes the state $\mathit{state}(X)$ defined as follows: a location receives the value of the last write event (according to co) accessing it; for a register, its value depends on the last event in po that writes to it. The relationship between the notions is as in Lemma 1.

Definition 2 (State Portability). *Let $\mathcal{M}_S$, $\mathcal{M}_T$ be MCMs. Program P is* state portable from $\mathcal{M}_S$ to $\mathcal{M}_T$ *if* $\mathit{state}(\mathit{cons}_{\mathcal{M}_T}(P)) \subseteq \mathit{state}(\mathit{cons}_{\mathcal{M}_S}(P))$.

Lemma 1. *(1) Portability implies state portability. (2) State portability does not imply portability.*

For Lemma 1.(2), consider a variant of **IRIW** (Fig. 1) where all written values are 0. The program is trivially state portable from Power to TSO, but like **IRIW**, not portable.

We turn to the hardness argumentation. To check state portability, every $\mathcal{M}_T$-computable state seems to need a formula checking whether some $\mathcal{M}_S$-consistent execution computes it. The result would be an exponential blow-up or a quantified Boolean formula, which is not practical. But can this exponential blow-up or quantification be avoided by some clever encoding trick? The answer is no! Theorem 1 shows that state portability is in a higher class of the polynomial hierarchy than portability. It is indeed harder to check than portability.

The polynomial hierarchy [42] contains complexity classes between NP and PSPACE. Each class is represented by the problem of checking validity of a Boolean formula with a fixed number of quantifier alternations. We need here the classes co-NP $= \Pi_1^P \subseteq \Pi_2^P$. The *tautology problem* (validity of a closed Boolean formula with a universal quantifier $\forall x_1 \dots x_n : \psi$) is a Π_1^P-complete problem. The higher class Π_2^P allows for a second quantifier: validity of a formula $(\forall x_1 \dots x_n \exists y_1 \dots y_n : \psi)$ is a Π_2^P-complete problem. Theorem 1 refers to a class of common memory models that we define in a moment. Moreover, we assume that the given pair of memory models $\mathcal{M}_S$ and $\mathcal{M}_T$ is *non-trivial* in the sense that $cons_{\mathcal{M}_T}(P) \subseteq cons_{\mathcal{M}_S}(P)$ fails for some program, and similar for state portability.

Theorem 1. *Let $\mathcal{M}_S, \mathcal{M}_T$ be a non-trivial pair of common MCMs. (1) Portability from $\mathcal{M}_S$ to $\mathcal{M}_T$ is Π_1^P-complete. (2) State portability is Π_2^P-complete.*

By Theorem 1.(2), state portability cannot be solved efficiently. The first part says that our portability analysis is optimal. We focus on this lower bound to give a taste of the argumentation: given a non-trivial pair of memory models, we know there is a program that is not portable. Crucially, we do not know the program but give a construction that works for any program. The proof of Theorem 1.(2) is along similar lines but more involved.

Definition 3. *We call an MCM* common[1] *if*

(i) the inverse operator is only used in the definition of fr,
(ii) the constructs sthd, sloc, and $\langle set \rangle \times \langle set \rangle$ are only used to restrict (in a conjunction) other relations,
(iii) it satisfies uniproc (Axiom ❶) , and
(iv) every program is portable from this MCM to SC.

We explain the definition. When formulating an MCM, one typically forbids well-chosen cycles of base relations (and *fr*). To this end, derived relations are introduced that capture the paths of interest, and acyclicity constraints are imposed on the derived relations. The operators inverse and $\langle set \rangle \times \langle set \rangle$ may do the opposite, they add relations that do not correspond to paths of base relations (and *fr*). Besides stating what is common in MCMs, Properties *(i)* and *(ii)* help

[1] Notice that all memory models considered in [8] and in this paper are common ones.

us compose programs (cf. next paragraph). Uniproc is a fundamental property without which an MCM is hard to program. Since the purpose of an MCM is to capture SC relaxations, we can assume MCMs to be weaker than SC. Properties *(iii)* and *(iv)* guarantee that the program P_ψ given below is portable between any common MCMs.

The crucial property of common MCMs is the following. For every pair of events e_1, e_2 in a derived relation, (1) there are (potentially several) sequences of base relations (and *fr*) that connect e_1 and e_2, and (2) the derived relation only depends on these sequences. The property ensures that if we append a program P' to a location-disjoint program P, any executions composed from consistent executions of P and P' is also consistent.

It remains to prove Π_1^P-hardness of portability by constructing a program that is portable iff a formula ψ is a tautology. We first introduce the program P_ψ that generates some assignment and checks if it satisfies the Boolean formula $\psi(x_1 \dots x_m)$ (over the variables $x_1 \dots x_m$). The program $P_\psi := t_1 \parallel t_2$ consists of the two threads t_1 and t_2 defined below. Note that we cannot directly write a constant i to a location, so we first assign i to register $r_{c,i}$.

thread t_1	*thread* t_2
$r_{c,0} \leftarrow 0;\ r_{c,1} \leftarrow 1;\ r_{c,2} \leftarrow 2$	$r_{c,1} \leftarrow 1;$
$x_1 := r_{c,0} \dots x_m := r_{c,0};$	$x_1 := r_{c,1} \dots x_m := r_{c,1};$
$r_1 \leftarrow x_1 \dots r_m \leftarrow x_m;$	
if $\psi(r_1 \dots r_m)$ **then**	
$\quad y := r_{c,2};$	
else $\ y := r_{c,1};$	

We reduce checking whether $\forall x_1 \dots x_m : \psi(x_1 \dots x_m)$ holds to portability of a program $P_{\forall\psi}$. The idea for $P_{\forall\psi}$ is this. First P_ψ is run, it guesses and evaluates an assignment for ψ. If ψ is not satisfied ($y = 1$), then some non-portable program P_{np} is executed. The program $P_{\forall\psi}$ is portable iff the non-portable part is never executed. This is the case iff ψ is satisfied by all assignments.

Let $\mathcal{M}_S$, $\mathcal{M}_T$ be common and non-trivial. By non-triviality, there is a program $P_{np} = t'_1 \parallel \dots \parallel t'_k$ that is **n**ot **p**ortable from $\mathcal{M}_S$ to $\mathcal{M}_T$. We can assume P_{np} has no registers or locations in common with P_ψ. Program $P_{\forall\psi}$ prepends P_ψ to the first two threads of P_{np}. Once $y = 1$, P_{np} starts running. Formally, let t_1 and t_2 be the threads in P_ψ and let $t_i := \mathit{skip}$ for $3 \le i \le k$. We define $P_{\forall\psi} := t''_1 \parallel \dots \parallel t''_k$ with $t''_i := t_i;\ r \leftarrow y;\ \mathbf{if}(r = 1)\ \mathbf{then}\ t'_i$.

We show that $P_{\forall\psi}$ is portable iff ψ is satisfied for every assignment.

6 Experiments

The encoding from Section 4 has been implemented in a tool called PORTHOS. We evaluate PORTHOS on benchmark programs and a wide range of well-known MCMs. For SC, TSO, PSO, RMO and Alpha (henceforth called traditional architectures) we use the formalizations from [3]; for Power the one in Fig. 6.

Benchmark	SC-TSO	SC-Power	TSO-Power
Bakery	✗	✗	✗
Bakery x86	✔	✗	✗
Bakery Power	✔	✔	✔
Burns	✗	✗	✗
Burns x86	✔	✗	✗
Burns Power	✔	✔	✔
Dekker	✗	✗	✗
Dekker x86	✔	✗	✗
Dekker Power	✔	✔	✔
Lamport	✗	✗	✗
Lamport x86	✔	✗	✗
Lamport Power	✔	✔	✔
Peterson	✗	✗	✗
Peterson x86	✔	✗	✗
Peterson Power	✔	✔	✔
Szymanski	✗	✗	✗
Szymanski x86	✔	✗	✗
Szymanski Power	✔	✔	✔

				Deadness	
	✗✗	✔✔	✗✔	✔✔	✗✔
SC-TSO	27	898	75	933	40
SC-PSO	27	777	196	836	137
SC-RMO	27	737	236	780	193
SC-Alpha	27	846	127	887	86
TSO-PSO	0	833	67	883	27
TSO-RMO	0	760	240	798	202
TSO-Alpha	0	877	133	912	88
PSO-RMO	0	831	169	844	156
PSO-Alpha	0	968	32	973	27
RMO-Alpha	0	999	1	999	1
Alpha-RMO	0	856	144	864	136
	0.98%	85.29%	13.73%	88.26%	10.73%
SC-Power	1477	898	52	936	14
TSO-Power	917	1132	378	1166	344
PSO-Power	502	1880	45	1892	33
RMO-Power	40	2227	160	2239	148
Alpha-Power	0	2427	0	2427	0
	24.20%	70.57%	5.23%	71.35%	4.45%

Table 1: (Left) Bounded portability analysis of mutual exclusion algorithms: portable (✔), non-portable (✗). **(Right)** Portability vs. State Portability on litmus tests.

We divide our results into three categories: portability of mutual exclusion algorithms, empirical comparison between portability and state portability, and performance of the tool.

Portability of Mutual Exclusion Algorithms. Most of the tools that are MCM-aware [8,35,45,48,49] accept only litmus tests as inputs. PORTHOS, however, can analyze cyclic programs with control flow branching and merging by unrolling them into acyclic form. In order to show the broad applicability of our method, we tested portability of several mutual exclusion algorithms: Lamport's bakery [32], Burns' protocol [15], Dekker's [23], Lamport's fast mutex [33], Peterson's [37] and Szymanski's [44]. The benchmarks also include previously known fenced versions for TSO taken from [12] (marked as x86) and new versions we introduced using Power fences (marked as Power). The loops were unrolled once in all the experiments to obtain an acyclic program, and the discussion in what follows is for the portability analysis of this acyclic program.

While these algorithms have been proven correct for SC, it is well known that they do not guarantee mutual exclusion when ported to weaker architectures. The effects of relaxing the program order have been widely studied; there are techniques that even place fences automatically to guarantee portability, but they assume SC as the source architecture [5,12]. In Table 1 (left) we do not only confirm that fenceless versions of the benchmarks are not portable from SC to TSO and fenced versions of them are, we also show that those fences are not enough to guarantee mutual exclusion when porting from TSO to Power. We have used PORTHOS to find portability bugs when porting from TSO to Power and manually added fences to forbid such executions (see benchmarks marked as Power). To the best of our knowledge these are the first results

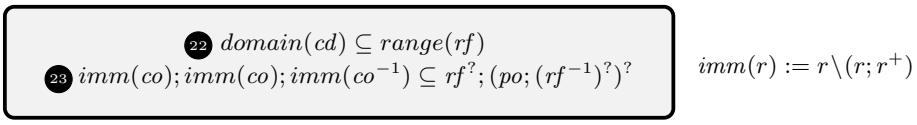

Fig. 7: Syntactic Deadness [48].

about portability of mutual exclusion algorithms from memory models weaker than SC to the Power architecture.

Portability vs State Portability. We empirically compare both notions of portability by using PORTHOS (which implements portability) and the HERD7 tool (`http://diy.inria.fr/herd`) which reasons about state reachability. HERD7 systematically constructs all consistent executions of the program and exhaustively enumerates all possible computable states. Such enumeration can be very expensive for programs with lots of computable states, e.g., for programs with a very large level of concurrency. Since HERD7 only allows to reason about one memory model at a time, for each test we run the tool twice (once for each MCM) and compare the set of computable states. The program is not state portable if the target MCM generates computable states that are not computable states of the source MCM.

Our experiments contain two test suites: TS_1 contains 1000 randomly generated litmus tests in x86 assembly (to test traditional architectures) and TS_2 contains 2427 litmus tests in Power assembly taken from [36]. Each test contains between 2 and 4 threads and between 4 and 20 instructions. Table 1 (right) reports the number of non-portable (w.r.t. both definitions) litmus tests (✗✗), the number of portable and state portable litmus tests (✔✔) and the number of litmus tests that are not portable but are still state portable (✗✔). In the last case the new executions allowed by the target memory model do not result in new computable states of the program. We show that in many cases both notions of portability coincide. On traditional architectures, TS_1 contains very few non state portable tests (0.98%). Here, a non portable program is state portable in only 13.73% of the cases. For TS_2 from traditional architectures to Power, the number of non state portable litmus tests rises to 24.20%, while only in 5.24% of the cases the two notions of portability do not match because the new executions do not result in a new computable state for the program.

In order to remove some executions that do not lead to new computable states, PORTHOS optionally supports the use of syntactic deadness which has been recently proposed in [48]. Dead executions are either consistent or lead to not computable states. Formally an execution X is dead if $X \notin cons_{\mathcal{M}}(P)$ implies that $state(X) \neq state(Y)$ for all $Y \in cons_{\mathcal{M}}(P)$. Instead of looking for any execution which is not consistent for the source architecture, we want to restrict the search to non-consistent and dead executions of $\mathcal{M}_S$. This is equivalent to checking state portability. As shown by Wickerson et al. [48], dead executions can be approximated with constraints 22 and 23 given in Fig. 7 where $r^{?}$ is the reflexive closure of r. These constraints can be easily encoded into SAT. Our tool has an implementation which rules out quite a few executions not comput-

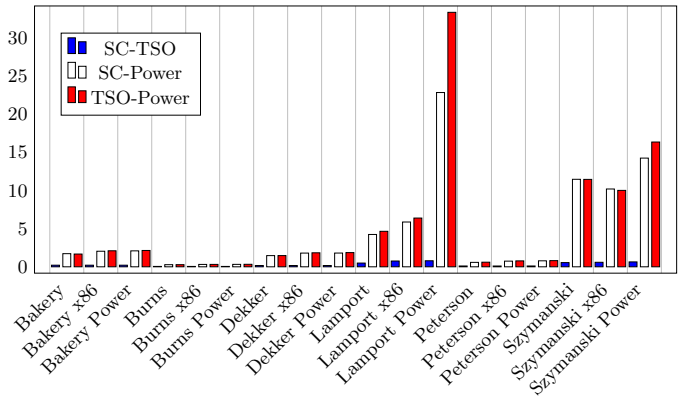

Fig. 8: Solving times (in secs.) for portability of mutual exclusion algorithms.

ing new states. The last two columns of Table 1 (right) show that by restricting the search to (syntactic) dead executions, the ratio of litmus tests the tool reports as non portable, but are actually state portable (due to syntactic dead executions that are not semantically dead) is reduced to 10.73% for traditional architectures and to 4.44% for Power.

The experiments above show that in most of the cases both notions of portability coincide, especially when using dead executions or porting to Power. To test state portability, our method can be complemented with an extra query to check if the final state of the counter-example execution is also reachable in the source model by another execution. As shown in Section 5, the price to obtain such a result is to go one level higher in the polynomial hierarchy which affects the performance. However, once an execution is found that disproves portability, one could check if the execution implies non state portability with a single existential query.

Performance. We evaluate the solving times of our tool on the mutual exclusion benchmarks as shown in Fig. 8. Our prototype encoding implementation is done in Python; the encoding times have a minimum of 13 seconds and a maximum of 303 seconds. The encodings involving Power are usually more time consuming than traditional models since Power has both transitive closures and least fixed points in its encoding. For the mutual exclusion algorithms, the solving times are actually much lower than the encoding times of our prototype implementation. We expect that the encoding times could be vastly improved by a careful C/C++ implementation of the encoding.

We acknowledge that for small litmus test, the running times of Herd7 outperform our prototype implementation. However, as soon as the programs become bigger, Herd7 does not perform as well as porthos. We believe this is due to the use of efficient search techniques in the SMT solver. In contrast, the number of executions Herd7 has to enumerate explicitly grows exponentially with the test size.

7 Related Work

Semantics and verification under weak memory models have been the subject of study at least since 2007. Initially, the behavior of x86 and TSO has been clarified [13,41], then the Power architecture has been addressed [36,40], now ARM is being tackled [26]. The study also looks beyond hardware, in particular C++11 received considerable attention [10,11]. Research in semantics goes hand in hand with the development of verification methods. They come in two flavors: program logics [46,47] and algorithmic approaches [1,2,6,8,9,12,14,20,21]. Notably, each of these methods and tools is designed for a specific memory model and hence is not directly able to handle porting tasks.

The problem of verifying consistency under weak memory models has been extensively studied. Multiple formalisations and variations of the problem and their complexity have been analyzed [16,24,25]. A prominent approach is testing where an execution is (partially) given and consistency is tested for a specified model [27,29]. In this line we showed that state portability (formulated as a bounded analysis for cyclic programs) is Π_2^p-complete. This means there is no hope for a polynomial encoding into SAT (unless the polynomial hierarchy collapses). In contrast, our execution-based notion of portability is co-NP complete (we look for a violation to portability), which in particular means that our portability analysis is optimal in the complexity sense. Our experiments show that in most of the cases both notions of portability coincide.

A problem less general than portability is solved in [12] where non-portable traces from SC to TSO are characterized. The problem is reduced to state reachability under the SC semantics in an instrumented program and a minimal number of fences is synthesized to enforce portability. One step further, one can enforce portability not only to TSO, but also to weaker memory models [22]. The OFFENCE tool [7] does this, but can only analyze litmus test and is limited to restoring SC. Checking the existence of critical cycles (i.e. portability bugs) on complex programs has been tackled in [5], where such cycles are broken by automatically introducing fences. The cost of different types of fences is considered and the task is encoded as an optimization problem. The MUSKETEER tool analyzes C programs and has shown to scale up to programs with thousands of lines of code, but the implementation is also restricted to the case were the source model is SC. Fence insertion can also be used to guarantee safety properties (rather than restoring SC behaviors). The FENDER and DFENCE tools [31,34] can verify real-world C code, but they are restricted to TSO, PSO, and RMO.

8 Conclusion and Outlook

We introduce the first method that tests portability between any two axiomatic memory models defined in the CAT language. The method reduces portability analysis to satisfiability of an SMT formula in SAT + integer difference logic. We propose efficient solutions for two crucial tasks: reasoning about two user-defined MCMs at the same time and encoding mutually recursively defined relations (needed for Power) into SMT. The latter technique can be re-used by any

bounded model checking technique reasoning about complex memory models such as Power.

Our complexity analysis and experimental results both suggest that our definition of portability is preferable over the state-based notion of portability. The complexity results show that checking for state-based portability cannot be done with a single SMT solver query, unlike the approach to portability analysis suggested in this paper. We also show that our method is not restricted to litmus tests and present an automated tool-based portability analysis of mutual exclusions algorithms from several axiomatic memory models to Power.

Acknowledgements

We thank John Wickerson for his explanations about dead executions, Luc Maranget for several discussions about CAT models, and Egor Derevenetc for providing help with the mutual exclusion benchmarks. This work has been partially developed under contracting of Liebherr Aerospace Lindenberg GmbH and supported by the Academy of Finland project 277522. Florian Furbach was supported by the DFG project R2M2: Robustness against Relaxed Memory Models.

References

1. Parosh A. Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.
2. Parosh A. Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, volume 9780 of *LNCS*, pages 134–156. Springer, 2016.
3. Jade Alglave. *A Shared Memory Poetics*. Thèse de doctorat, L'université Paris Denis Diderot, 2010.
4. Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.
5. Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, volume 8559 of *LNCS*, pages 508–524. Springer, 2014.
6. Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
7. Jade Alglave and Luc Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.
8. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.
9. Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.
10. Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

11. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.
12. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.
13. Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.
14. Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.
15. James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171 – 184, 1993.
16. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671, 2005.
17. Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.
18. William W. Collier. *Reasoning about parallel architectures*. Prentice Hall, 1992.
19. Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS*, volume 3253 of *LNCS*, pages 263–276. Springer, 2004.
20. Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, volume 7935 of *LNCS*, pages 84–104. Springer, 2013.
21. Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, volume 8931 of *LNCS*, pages 449–466. Springer, 2015.
22. Egor Derevenetc and Roland Meyer. Robustness against power is pspace-complete. In *ICALP*, volume 8573 of *LNCS*, pages 158–170. Springer, 2014.
23. Edsger W. Dijkstra. Cooperating sequential processes. In *The Origin of Concurrent Programming*, pages 65–138. Springer-Verlag New York, Inc., 2002.
24. Constantin Enea and Azadeh Farzan. On atomicity in presence of non-atomic writes. In *TACAS*, volume 9636 of *LNCS*, pages 497–514. Springer, 2016.
25. Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV*, volume 5123 of *LNCS*, pages 52–65. Springer, 2008.
26. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL*, pages 608–621. ACM, 2016.
27. Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63, 2015.
28. Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *JELIA*, pages 137–151, 2014.
29. Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26:1208–1244, 1997.
30. Keijo Heljanko, Misa Keinänen, Martin Lange, and Ilkka Niemelä. Solving parity games by a reduction to SAT. *J. Comput. Syst. Sci.*, 78(2):430–440, 2012.
31. Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

32. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
33. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.
34. Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.
35. Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2010.
36. Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *LNCS*, pages 495–512. Springer, 2012.
37. Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
38. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for axiomatic memory models. PORTHOS: One tool for all models. *CoRR*, abs/1702.06704, 2017.
39. Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
40. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.
41. Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391. ACM, 2009.
42. Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.
43. Viggo Stoltenberg-Hansen, Edward R. Griffor, and Ingrid Lindstrom. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.
44. Boleslaw K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS*, pages 621–626. ACM, 1988.
45. Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*, pages 341–350. ACM, 2010.
46. Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.
47. Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.
48. John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204. ACM, 2017.
49. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE Computer Society, 2004.