Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Robustness against Relaxed Memory Models

Autor:
Egor Derevenetc

Datum der Disputation: 24.04.2015

Vorsitzender:	Prof. Dr. Klaus Schneider
1. Berichterstatter:	Prof. Dr. Roland Meyer
2. Berichterstatter:	Prof. Dr. Rupak Majumdar

Dekan des Fachbereichs Informatik:
Prof. Dr. Klaus Schneider

D 386

Abstract

Sequential Consistency (SC) is the memory model traditionally applied by programmers and verification tools for the analysis of multithreaded programs. SC guarantees that instructions of each thread are executed atomically and in program order. Modern CPUs implement memory models that relax the SC guarantees: threads can execute instructions out of order, stores to the memory can be observed by different threads in different order. As a result of these relaxations, multithreaded programs can show unexpected, potentially undesired behaviors, when run on real hardware.

The *robustness* problem asks if a program has the same behaviors under SC and under a relaxed memory model. Behaviors are formalized in terms of *happens-before* relations — dataflow and control-flow relations between executed instructions. Programs that are robust against a memory model produce the same results under this memory model and under SC. This means, they only need to be verified under SC, and the verification results will carry over to the relaxed setting.

Interestingly, robustness is a suitable correctness criterion not only for multithreaded programs, but also for parallel programs running on computer clusters. Parallel programs written in Partitioned Global Address Space (PGAS) programming model, when executed on cluster, consist of multiple processes, each running on its cluster node. These processes can directly access memories of each other over the network, without the need of explicit synchronization. Reorderings and delays introduced on the network level, just as the reorderings done by the CPUs, may result into unexpected behaviors that are hard to reproduce and fix.

Our first contribution is a generic approach for solving robustness against relaxed memory models. The approach involves two steps: combinatorial analysis, followed by an algorithmic development. The aim of combinatorial analysis is to show that among program computations violating robustness there is always a computation in a certain normal form, where reorderings are applied in a restricted way. In the algorithmic development we work out a decision procedure for checking whether a program has violating normal-form computations.

Our second contribution is an application of the generic approach to widely implemented memory models, including Total Store Order (TSO) used in Intel x86 and Sun SPARC architectures, the memory model of Power architecture, and the PGAS memory model. We reduce robustness against TSO to SC state reachability for a modified input program. Robustness against Power and PGAS is reduced to language emptiness for a novel class of automata — multiheaded automata. The reductions lead to new decidability results. In particular, robustness is PSpace-complete for all the considered memory models.

Acknowledgements

First and foremost, I thank my scientific advisor Prof. Dr. Roland Meyer for being a mentor and a friend, setting high research standards, and showing by personal example how to pursue them. I am immensely grateful for his time, knowledge, hints, and advices, which run all through the research that led to this thesis.

I deeply acknowledge the Fraunhofer Institute for Industrial Mathematics (ITWM) and personally the director of the Competence Center for High Performance Computing and Visualization (CC-HPC) Dr. Franz-Josef Pfreundt for the scholarship and financial support of my trips to conferences and summer schools.

I thank Dr. Mirko Rahn for organizing my coming to Kaiserslautern, being constantly positive and helpful, and introducing me to the world of modern high-performance computing.

I was privileged to collaborate with the experts in program verification, relaxed memory models, and formal languages: Dr. Mohamed Faouzi Atig, Prof. Dr. Ahmed Bouajjani, Georgel Calin, Carl Leonardsson, and Prof. Dr. Rupak Majumdar. This work contains traces of days-long discussions of Power memory model with Ahmed, Carl, Faouzi, and Roland. Colleagues, thank you for your time and input!

I express gratitude to Dr. Jade Alglave for her precious clarifications on Power memory model.

I thank my ITWM colleagues Dr. Daniel Grünewald, Dr. Martin Kühn, Dr. Rui Machado, and Dr. Mirko Rahn, for sharing their expertise in GPI, GASPI, and programming for clusters in general.

I am grateful to everybody who read the draft of the thesis and gave feedback.

I thank my family and friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Relaxed Memory Models

In order to deliver maximum performance, modern CPUs execute instructions out of order. The reorderings typically respect the data and control dependencies between instructions within one thread of execution.[1] Consequently, single-threaded programs preserve the sequential semantics on these CPUs. Multi-threaded programs, however, can observe these reorderings and, as a result, show unexpected behaviors.

The guarantees provided by an architecture with respect to the visible ordering of operations are specified in the *memory consistency model*, or, simply, the *memory model* of this architecture. Perhaps the most intuitive memory model is Sequential Consistency (SC) [56]. It defines that instructions of each thread are executed in order and stores to the memory become immediately visible to all the threads. This memory model is assumed by most programmers and verification tools. Modern processor architectures, on the contrary, adopt models weaker than SC.

Intel x86 [47] and SPARC [84] processors implement the Total Store Order (TSO) [72, 84] memory model. The model reflects the use of store buffers. When a thread executes a store instruction, it adds a store operation comprising the address and the value to be written to this address to the local store buffer of this thread. (Actually, the store buffer belongs to the CPU core executing the thread. However, without loss of generality we can always assume that each thread runs on its own CPU core.) Later these operations are non-deterministically popped from the buffer and performed on the memory in FIFO order. Loads read either from the last buffered store to the same address, or, if no such store exists, from memory.

The SPARC architecture [84] defines another two memory models: Partial Store Order (PSO) and Relaxed Memory Order (RMO). PSO extends TSO by allowing reorderings of stores to different addresses; it can be formalized similar to TSO in terms of store buffers: each thread has one buffer per address. RMO relaxes PSO by allowing out-of-order loads. Support for PSO and RMO is declared optional [84], and recent SPARC CPUs seem to implement solely

[1] DEC Alpha [80] architecture is a notable exception admitting reorderings of dependent load instructions [67].

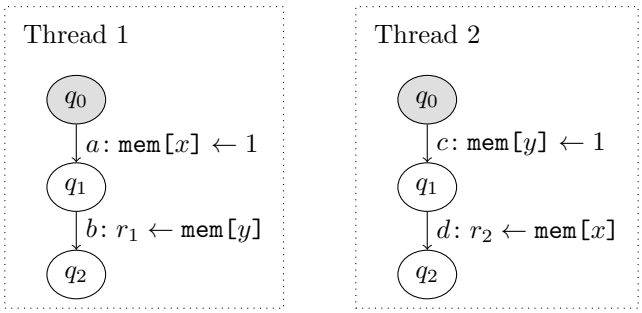

Figure 1.1: Store Buffering (SB) program. Initially, $x = y = 0$.

TSO [67].

Power [32] and ARM [12] architectures have memory models that are even more relaxed [11, 65, 66, 78]. Unlike the SPARC memory models, Power and ARM do not feature store atomicity: one store can become visible to different threads at different times, stores to different memory locations can be seen in different order by different threads. Nevertheless, Power guarantees that all threads see stores to the same memory location in the same order. Although the currently proposed model for ARM includes the same guarantee, it is explicitly noted that it is violated on existing hardware [11].

In all the mentioned relaxed models different threads at the same moment of time may observe different memory states. For example, on TSO a thread observes the memory state being a combination of the global memory state and the contents of this thread's store buffer. As a result, programs running on architectures with relaxed memory models can show behaviors that are impossible under SC, where all threads always agree on the memory contents.

Example 1.1. Consider the SB program shown in Figure 1.1. It implements a simplified version of the Dekker's mutual exclusion protocol for two threads [35]. The first thread signals that it wants to enter the critical section by storing value 1 to variable x. Next, it checks whether the second thread wants to enter the critical section. For this it loads the value of variable y to register r_1. It is assumed that, if $r_1 = 0$, then the thread enters the critical section by a transition from q_2 (omitted). The second thread is symmetric. Under SC it is guaranteed that at most one thread reads value 0, and mutual exclusion holds. On a TSO architecture the stores may be buffered and executed on memory only after loads are completed. Consequently, both threads can load 0, and mutual exclusion fails.

Example 1.2. Consider the Message Passing (MP) program shown in Figure 1.2. In this program the first thread sends some useful data to the second thread. For this, it writes the data to variable x. Then, it sets the flag variable y to 1, to signal that the data is ready. The second thread reads the flag variable y. It is assumed that, if the second thread has observed the new value of y, it will also read the data written by the first thread to variable x. The program indeed works as expected under SC and TSO memory models. However, it will fail on Power machines, because, first, their memory model allows out-of-order execution of loads from different addresses, second, it allows stores to different

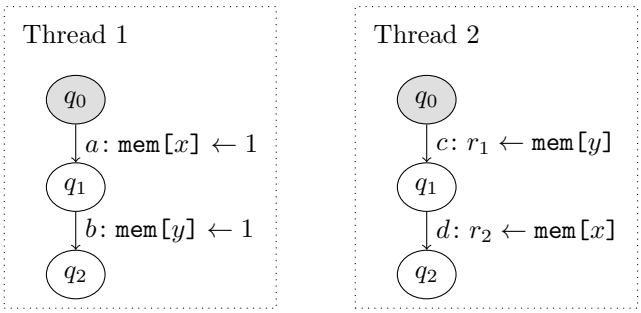

Figure 1.2: Message Passing (MP) program. Initially, $x = y = 0$.

addresses to be observed by different threads in different order. Any of these features is sufficient to make the program misbehave.

From the above examples it becomes clear that verification problems for concurrent programs must be posed in the context of a certain memory model. In the next section we give an overview of the major verification problems for concurrent programs and show interrelations between them.

1.2 Verification Problems

Multiple verification problems were formulated for concurrent programs running on relaxed memory models. Probably the most basic one is *state reachability*. It asks whether a given program reaches a certain (bad) state under a given memory model. Checking whether an assertion fails is a classic example of a state reachability problem. State reachability is long-known to be PSPACE-complete for programs with SC semantics [52]. It was recently shown that under TSO and PSO the problem is non-primitive recursive (lower and upper bound) [14].

The reachability problem, as a correctness criterion, has two disadvantages. The first impediment to using it is the necessity to mark the bad program states, i.e., to write a specification, which implies additional work for software engineers. The second disadvantage is the high computational complexity of the problem. As mentioned above, reachability is very difficult already for TSO. Moreover, in Section 4.5 we will show that state reachability under Power is undecidable.

An alternative problem to reachability is *robustness*. It asks whether a given program has the same behaviors under SC and under a relaxed memory model. The definition depends on the notion of behavior used. A program is called *trace-robust* against a relaxed memory model, if all its computations under this model have the same data and control dependencies as some SC computations. It is called *state-robust*, if its threads can reach the same control states under SC and under a relaxed memory model.

Example 1.3. The program SB (Figure 1.1) is not trace-robust against TSO, PSO, RMO, Power. Indeed, under TSO the loads in both threads can read the initial value zero (see Example 1.1), which is impossible under SC. PSO, RMO, Power memory models are weaker than TSO and allow all the TSO behaviors.

Example 1.4. The program MP (Figure 1.2) is trace-robust against TSO, but not against PSO, RMO, and Power. The latter models allow the second thread to observe the store to y before the store to x and reach the state $r_1 = 1, r_2 = 0$.

Formally, both programs are state-robust, because they do not use the values that they load, and, therefore, their threads reach the same control states. However, both programs can be rendered not state-robust by adding additional checks. For example, in the SB program the threads can check that they did not both read 0 from x and y and, if they did, enter a special state. This state will be reachable under all considered relaxed memory models, but not under SC, which will make the program not state-robust against these relaxed memory models. See Example 2.11 for details.

State-robustness can be naively solved by enumerating all the states reachable under a relaxed memory model and checking whether each of them can be reached under SC. Actually, we will show that this is an optimal solution: Theorem 2.12 shows that state-robustness against a memory model is as hard as state reachability for this model. Taking into account that state reachability is non-primitive-recursive for TSO and PSO [14] and undecidable for Power (Theorem 4.25), state-robustness becomes quite unattractive as a correctness criterion.

Trace-robustness is an algorithmically easier problem than its state-based counterpart. We will show that trace-robustness is PSPACE-complete for all the memory models considered in the thesis. Moreover, trace-robustness implies state-robustness (Theorem 2.19), i.e., trace-robust programs reach the same states under SC and a relaxed memory model. In the thesis we focus on trace-robustness and call it *robustness* in the following.

Once a program was shown to be non-robust, the next logical step is to make it robust. Robustness can be enforced by inserting special instructions that forbid or constrain certain kinds of reorderings. These instructions are commonly called *memory fences* or *barriers*. The problem of *enforcing robustness* consists in determining an in some sense optimal set of fences guaranteeing robustness. Optimality can be defined either in terms of the number of inserted fences or in the performance costs introduced by them. Clearly, the problem of enforcing robustness is no easier that the robustness problem itself, since a program is robust iff the smallest fence set guaranteeing robustness is empty.

Data-race freedom (DRF) [3] is another useful correctness criterion for concurrent programs. A *data race* is a pair of concurrent memory accesses to the same address with one of the accesses being a store. Most memory models provide the so-called *DRF guarantee*: data-race-free programs do not show non-SC behaviors under them (i.e., they are robust). High-level languages with this guarantee include Java [40, 13], C [48], and C++ [49]. Moreover, C and C++ consider data races (on non-atomic variables) as errors and specify racy programs to have undefined behavior. Saraswat et al. [77] proposed a framework for defining memory models of high-level languages and established that all the models defined in this framework provide the DRF guarantee. Among CPU models, the DRF guarantee was formally proven for TSO and PSO memory models [22, 75], although it should hold for all the memory models considered in the thesis. Unfortunately, implementations of synchronization primitives, lock-free data structures, and miscellaneous high-performance algorithms are racy on purpose [27, 34, 35, 38, 42, 51, 57, 58, 74]. Consequently, data-race

freedom is a too strong correctness criterion for such programs.

All the previously mentioned verification problems apply to programs with a fixed number of threads. However, concurrent algorithms and data structures are often designed to be used by unboundedly many threads. Verification of such data structures is addressed by the *parameterized* versions of the verification problems. Instead of analyzing programs running a single instance of each thread, they consider programs that admit any number of instances of each thread running concurrently. Parameterized state reachability under SC is commonly known to be decidable, by a reduction to coverability in Petri nets (Lemma 2.8), but generally ExpSpace-hard (Lemma 2.9). In Section 6.3 we show that parameterized robustness against TSO is decidable and as hard as parameterized state reachability under SC.

1.3 Partitioned Global Address Space

Problems caused by reorderings of memory accesses on the hardware level arise not only in the world of multithreaded programs, but also in the world of parallel applications for clusters. Software for computer clusters was traditionally developed within the message passing paradigm, typically using a Message Passing Interface (MPI) [37] library for communication. Partitioned Global Address Space (PGAS) is an alternative programming model for clusters that gained a considerable recent attention [19, 29, 30, 39, 43, 63, 68, 70].

A distinctive feature of PGAS is the emphasis on one-sided communication. Processes running on different nodes of the cluster can directly access memories of each other. These accesses generally do not require synchronization between the processes running on communicating nodes. This is contrary to the message passing paradigm, where the sender and the receiver are always explicitly synchronized.

One-sided communication is often implemented on top of a network providing Remote Direct Memory Access (RDMA). Applications can entirely delegate data transfers to an RDMA-enabled network interface controller. The controller will transfer the given block of data between the given nodes on its own, without putting any load on the CPUs.

As in most packet-switching networks, the packets containing the data being transferred may arrive to the destination node and be written to its memory out of order. As in the case of CPU memory models, this may result in programs having odd behaviors which are hard to reproduce and debug. Interestingly, all the verification problems described in the previous section can be formulated and make sense for PGAS applications.

1.4 Contributions and Outline

The first contribution of the thesis is a generic approach for solving robustness. This approach is described in Chapter 3 and consists of two steps. The first step is a combinatorial analysis. In this step we show that, if a program has computations violating robustness, it has a violating computation in a special *normal form*, where memory accesses are reordered in a certain restricted way. Consequently, checking robustness amounts to finding violating normal-form

computations. The second step is an algorithmic development, where we reduce robustness to language emptiness. We do it as follows. First, we show that normal-form computations can be generated by multiheaded automata — a novel class of automata developed in the context of robustness. Then, we filter violating normal-form computations using an intersection with finite automata. Altogether, the program is robust iff the intersection is empty.

The second contribution of the thesis is the application of the generic approach to several widely used memory models.

In Chapter 4 we consider the memory model of the IBM Power architecture. Following our generic approach, we reduce robustness against Power to language emptiness for multiheaded automata. This reduction leads to a new complexity result: robustness against Power is PSPACE-complete, a result also presented in [33]. For contrast, in Section 4.5 we additionally prove that state reachability under Power is undecidable.

In Chapter 5 we show how the results obtained for Power can be applied to the whole hierarchy of SPARC memory models: RMO, PSO, TSO.

In Chapter 6 we consider the TSO memory model in more detail. By combinatorial analysis we show that TSO additionally enjoys a *locality* property: it is sufficient to look only for violating normal-form computations where a single thread does reorderings. We call these computations *TSO witnesses*. Next, we show how to modify (instrument) the original program in order to let it detect TSO witnesses. This gives us a PSPACE decision procedure for robustness against TSO, as well as decidability for the parameterized robustness problem. We explain how to use the decision procedure to efficiently compute a minimal set of fence instructions that must be inserted to enforce robustness. The results presented in this chapter are also published in [20].

In Chapter 7 we discuss the implementation of the algorithms from Chapter 6 in our tool Trencher and present the results of testing this tool on well-known concurrent algorithms and data structures.

In Chapter 8 we consider the robustness problem for programs running on computer clusters. We propose a unified model for programs using PGAS APIs for communication and show how to express the semantics of popular PGAS APIs in terms of this model. Similar to Power, we derive a normal form of violating computations and reduce robustness against PGAS to language emptiness for multiheaded automata, a result presented in [28]. Finally, we note that robustness (as well as reachability) becomes undecidable in the parameterized PGAS setup.

We begin with the introduction of necessary definitions, Chapter 2. The graph of dependencies between the chapters is shown in Figure 1.3.

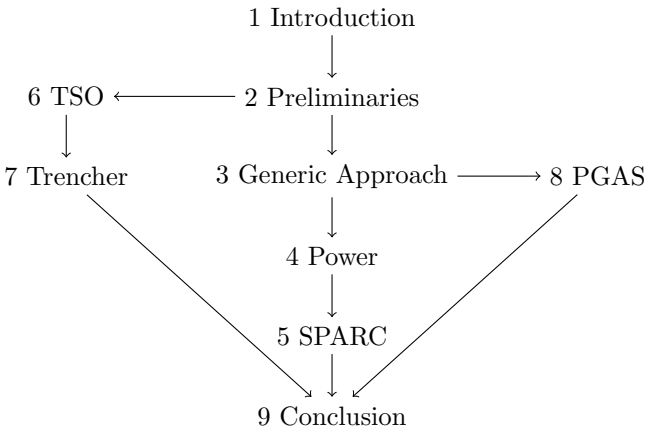

Figure 1.3: Graph of dependencies between the chapters of the thesis.

Chapter 2

Preliminaries

2.1 Automata

In the thesis we define programs and their semantics in an automata-theoretic way. A (nondeterministic) *automaton* is a tuple $A = (S, \Sigma, \Delta, s_0, F)$, where S is a set of states, Σ is an alphabet (a set of *symbols*), $\Delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a set of transitions, $s_0 \in S$ is an initial state, and $F \subseteq S$ is a set of final states. We call the automaton *finite* if S and Σ are finite. We write $s_1 \xrightarrow{a} s_2$ if $t = (s_1, a, s_2) \in \Delta$ and denote $\mathsf{src}(t) := s_1$, $\mathsf{dst}(t) := s_2$, $\mathsf{lab}(t) = a$. We naturally extend $\to$ to sequences Σ^*. The *language* of the automaton is $\mathcal{L}(A) := \{\sigma \in \Sigma^* \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}$. We say that a state $s_1 \in S$ is *reachable from a state* $s_2 \in S$ if $s_2 \xrightarrow{\sigma} s_1$ for some $\sigma \in \Sigma^*$. We say that a state $s \in S$ is *reachable in automaton* A if it is reachable from s_0.

A *word* σ *in alphabet* Σ is a finite sequence $\sigma := a_1 \dots a_n \in \Sigma^*$. We define its length as $|\sigma| := n$, $\sigma[i] := a_i$, $\mathsf{first}(\sigma) := a_1$, and $\mathsf{last}(\sigma) := a_n$. The length of the empty word ε is zero. We call a word α a *subsequence* of the word σ, if $\alpha = a_{i_1} \dots a_{i_m}$, where $1 \le i_1 < i_2 < \dots < i_m \le n$. Given $\alpha = a_1 \dots a_n$ and $\beta = b_1 \dots b_m$, we denote their *concatenation* as $\alpha \cdot \beta := a_1 \dots a_n b_1 \dots b_m$. We call α a *prefix* of σ and write $\alpha \sqsubseteq \sigma$ if $\sigma = \alpha \cdot \beta$ for some β. We call β a *suffix* of σ and write $\beta \sqsupseteq \sigma$ if $\sigma = \alpha \cdot \beta$ for some α. We call γ a *subword* of σ if $\sigma = \alpha \cdot \gamma \cdot \beta$ for some prefix α and suffix β. We say that a is before b in σ and write $a <_\sigma b$ if $\sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$. We write $\mathsf{succ}(\sigma)$ for the successor relation on σ. Given an alphabet $\Sigma' \subseteq \Sigma$, the *projection* $\sigma{\downarrow}\Sigma'$ *of* σ *onto* Σ' is the longest subsequence of σ in alphabet Σ'.

Lemma 2.1. *Language emptiness for a finite automaton is* NL*-complete.*

Proof. Language emptiness is equivalent to the *st*-non-connectivity problem for directed graphs: the language is empty iff no final state of the automaton is reachable from its initial state. The *st*-connectivity problem is well-known to be NL-complete. Taking into account that NL = co-NL (a corollary of Immerman-Szelepcsényi Theorem [46, 82]), we get the statement of the lemma. □

2.2 Petri Nets

We reduce parameterized versions of robustness and reachability for some memory models to the coverability problem for Petri nets. In this section we provide the necessary definitions.

A *Petri net* is a triplet $N = (P, T, W)$ where P is a finite set of *places*, T is a finite set of *transitions* with $P \cap T = \emptyset$, and $W \colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is a *weight function*. A *marking* is a function that assigns a natural number to each place: $M \colon P \to \mathbb{N}$. A *marked Petri net* is a pair (N, M_0) of a Petri net and an *initial marking* M_0. A transition $\mathsf{t} \in T$ is *enabled in marking* M if $M(\mathsf{p}) \geq W(\mathsf{p}, \mathsf{t})$ for all $\mathsf{p} \in P$. The *firing relation* $[\rangle \subseteq \mathbb{N}^P \times T \times \mathbb{N}^P$ contains a tuple (M_1, t, M_2) iff transition t is enabled in M_1 and for all $\mathsf{p} \in P$ we have $M_2(\mathsf{p}) = M_1(\mathsf{p}) - W(\mathsf{p}, \mathsf{t}) + W(\mathsf{t}, \mathsf{p})$. We also write $M_1[\mathsf{t}\rangle M_2$. We naturally extend the firing relation to sequences of transitions.

We say that a marking M is *reachable* in a marked Petri net (N, M_0) if there is a transition sequence $\sigma \in \Delta^*$, such that $M_0[\sigma\rangle M$. A marking M is *coverable* if there is a reachable marking M' so that $M'(\mathsf{p}) \geq M(\mathsf{p})$ for all $\mathsf{p} \in P$.

The *Petri net coverability* problem consists in deciding whether a given marking M is coverable in a given Petri net (N, M_0). The problem was shown to be ExpSpace-hard by Lipton [60] and decidable in ExpSpace by Rackoff [76].

Lemma 2.2 ([60, 76])**.** *Petri net coverability is* ExpSpace*-complete.*

2.3 Programs

A *program* is a finite sequence of threads: $\mathcal{P} = \mathcal{T}_1 \dots \mathcal{T}_n$. A *thread* is an automaton $\mathcal{T}_{\mathsf{tid}} := (Q_{\mathsf{tid}}, \mathsf{CMD}, \mathcal{I}_{\mathsf{tid}}, q_{\mathsf{tid}0}, Q_{\mathsf{tid}})$ with a finite set of control states Q_{tid}, all of them being final, initial state $q_{\mathsf{tid}0}$, and a set of transitions $\mathcal{I}_{\mathsf{tid}}$ called *instructions* and labeled with *commands* CMD defined below. Each thread has an id from $\mathsf{TID} := [1..|\mathcal{P}|]$.

Let $\mathsf{DOM} = \mathsf{ADDR}$ be a finite domain of values and addresses containing the value 0. Let REG be a finite set of registers that take values from DOM. The set of commands CMD includes loads, stores, local assignments, and conditionals (`assume`):

$$\begin{aligned}\langle cmd \rangle \;::=\; & \langle reg \rangle \leftarrow \texttt{mem[}\langle expr \rangle\texttt{]} \mid \texttt{mem[}\langle expr \rangle\texttt{]} \leftarrow \langle expr \rangle \\ \mid\; & \langle reg \rangle \leftarrow \langle expr \rangle \mid \texttt{assume(}\langle expr \rangle\texttt{)}\end{aligned}$$

In the further parts of the thesis we will extend CMD with architecture-specific commands. The set of expressions EXPR is defined over constants from DOM, registers from REG, and (unspecified) functions FUN over $\mathsf{DOM} \cup \{\bot\}$. We assume that these functions return $\bot$ iff any of the arguments is $\bot$.

As the *size of program* $\mathcal{P}$ we take the length of its binary representation plus the cardinality of the data domain $|\mathsf{DOM}|$.

2.4 Program Semantics

In the thesis we tend to define semantics of programs in an operational way. Given a program $\mathcal{P}$ and a particular memory model mm, we define its semantics under this model as an automaton $X_{\mathsf{mm}}(\mathcal{P}) := (S_{\mathsf{mm}}, \mathrm{E}_{\mathsf{mm}}, \Delta_{\mathsf{mm}}, s_{\mathsf{mm}0}, F_{\mathsf{mm}})$.

The states of this automaton correspond to the states of the running program. The transitions match the execution steps of the program. We label transitions with *events* E_{mm}. We call a sequence of events $\sigma \in \mathrm{E}_{\mathsf{mm}}^*$ a *computation*. The set of all *mm-computations of program* $\mathcal{P}$ is $\mathsf{C}_{\mathsf{mm}}(\mathcal{P}) := \mathcal{L}(X_{\mathsf{mm}}(\mathcal{P}))$.

2.4.1 Sequential Consistency

Sequentially consistent (SC) semantics is probably the most intuitive semantics a program can have. It was introduced by Lamport [56] and defined as follows:

> ... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Formally, we define a state $s \in S_{\mathsf{sc}}$ of a program $\mathcal{P}$ running under SC as a tuple $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem})$, where counter configuration $\mathsf{sn}\colon \mathsf{TID} \to \mathbb{N}$ gives, for each thread, the id that will be assigned to the next instruction executed in this thread, $\mathsf{pc}(\mathsf{tid}) \in Q_{\mathsf{tid}}$ gives the control state of the thread tid, and $\mathsf{mem}\colon \mathsf{TID} \times \mathsf{REG} \cup \mathsf{ADDR} \to \mathsf{DOM}$ gives, for each address and each register, the value stored at this address or in this register. The initial state is $s_{\mathsf{sc}0} := (\mathsf{sn}_0, \mathsf{pc}_0, \mathsf{mem}_0)$, where $\mathsf{sn}_0 := \lambda\mathsf{tid}.1$, all the control states are initial: $\mathsf{pc}_0(\mathsf{tid}) := q_{\mathsf{tid}\,0}$, and the memory is filled with zeroes: $\mathsf{mem}_0(\mathsf{a}) := 0$ for all $\mathsf{a} \in \mathsf{TID} \times \mathsf{REG} \cup \mathsf{ADDR}$. All states are final: $F_{\mathsf{sc}} := S$.

The SC transition relation Δ_{sc} consists of all the transitions defined by the rules in Table 2.1. As $\widehat{e}$ we denote the value of an expression e in thread tid. Given a function $f\colon X \to Y$, $x' \in X$, and $y' \in Y$, we define $f' := f[x' := y']$ by $f'(x) := f(x)$ for $x \in X \setminus \{x'\}$ and $f'(x') := y'$. The first rule describes a load from memory. The second rule describes a store to memory. The third rule defines the semantics of a local assignment. The fourth rule specifies that a conditional is executable whenever the value of the argument expression is not zero. In the chapters devoted to relaxed memory models we will extend CMD with architecture-specific memory barriers. We assume that their SC semantics is equivalent to that of the `assume(1)` instruction.

Example 2.3. The following computation $\sigma \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$ is an SC computation of program SB (Figure 1.1):

$$\sigma := abcd \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P}),$$

where

- $a := (1, 1, q_1 \xrightarrow{\texttt{mem}[x]\leftarrow 1} q_2, x)$,
- $b := (1, 2, q_2 \xrightarrow{r_1\leftarrow\texttt{mem}[y]} q_3, y)$,
- $c := (2, 1, q_1 \xrightarrow{\texttt{mem}[y]\leftarrow 1} q_2, y)$,
- $d := (2, 2, q_2 \xrightarrow{r_2\leftarrow\texttt{mem}[x]} q_3, x)$.

Note that any prefix $\tau \sqsubseteq \sigma$ is also a valid SC computation of this program.

$$\frac{\mathsf{cmd} = r \leftarrow \texttt{mem}[e], \quad \mathsf{a} := \widehat{e}}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \mathsf{mem}(\mathsf{a})])}$$

$$\frac{\mathsf{cmd} = \texttt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}}, \quad \mathsf{a} := \widehat{e_{\mathsf{a}}}, \quad \mathsf{v} := \widehat{e_{\mathsf{v}}}}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[\mathsf{a} := \mathsf{v}])}$$

$$\frac{\mathsf{cmd} = r \leftarrow e}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \widehat{e}])}$$

$$\frac{\mathsf{cmd} = \texttt{assume}(e), \quad \widehat{e} \neq 0}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem})}$$

Table 2.1: SC transition rules, assuming $\mathsf{pc}(\mathsf{tid}) = q$, an instruction $\mathsf{instr} = q \xrightarrow{\mathsf{cmd}} q'$, $\mathsf{pc}' := \mathsf{pc}[\mathsf{tid} := q']$, and $\mathsf{sn}' := \mathsf{sn}[\mathsf{tid} := \mathsf{sn}(\mathsf{tid}) + 1]$.

Fix a computation $\sigma \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$ and an event $\mathsf{e} \in \sigma$. As $\mathsf{tid}(\mathsf{e})$ we denote the thread id of the event, i.e., the first component of e. As $\mathsf{id}(\mathsf{e})$ we denote the serial number of the event, i.e., the second component of e. As $\mathsf{instr}(\mathsf{e})$ we denote the instruction that created the event, i.e., the third component of the event. For load and store events, we use $\mathsf{addr}(\mathsf{e})$ to refer to the last component of the event showing the address being accessed. For instance, in Example 2.3 we have $\mathsf{tid}(a) = 1$, $\mathsf{id}(a) = 1$, $\mathsf{instr}(a) = q_1 \xrightarrow{\texttt{mem}[x] \leftarrow 1} q_2$, and $\mathsf{addr}(a) = x$.

We say that event e *belongs to instruction* $(\mathit{tid}, \mathit{id})$ if $\mathsf{tid}(\mathsf{e}) = \mathsf{tid}$ and $\mathsf{id}(\mathsf{e}) = \mathsf{id}$.

2.4.2 Total Store Order

Total Store Order (TSO) is a popular and simple relaxed memory model used, e.g., in Intel x86 and Sun SPARC architectures [72, 84]. In this section we present a formalization of TSO semantics in terms of store buffers, as described by Owens et al. [72].

When a thread executes a store, it adds the store operation (the address and the value being written) to the FIFO buffer of this thread. The operation is non-deterministically popped from the buffer at some point later and executed on memory. Load operations first snoop into the buffer of the same thread. If there are no stores to the same address buffered, a load from memory happens. Otherwise, the load takes the value from the last buffered store to the same address (an *early read* situation). One can force flushing the buffered stores to the memory by inserting special memory barrier instructions: `mfence` on x86 or `membar #StoreStore | #StoreLoad` on SPARC.

The SPARC and x86 architectures provide special instructions for performing atomic operations. SPARC provides atomic load-store (`ldstub`), swap (`swap`), and compare and swap (`cas`) [84]. Atomic instructions on x86 are the usual instructions extended with the `lock` prefix. A `lock`ed instruction is executed as follows: the thread flushes the buffers, acquires exclusive access to the memory, executes the instruction itself, flushes the buffers again, and releases

$$\frac{\mathsf{cmd}{=}r \leftarrow \texttt{mem}[e], \quad \mathsf{a} := \widehat{e}, \quad \mathsf{buf}(\mathsf{tid}){\downarrow}(\mathbb{N} \times \{\mathsf{a}\} \times \mathsf{DOM}) = \beta \cdot (\mathsf{id}, \mathsf{a}, \mathsf{v})}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \mathsf{v}], \mathsf{buf})}$$

$$\frac{\mathsf{cmd} = r \leftarrow \texttt{mem}[e], \quad \mathsf{a} := \widehat{e}, \quad \mathsf{buf}(\mathsf{tid}){\downarrow}(\mathbb{N} \times \{\mathsf{a}\} \times \mathsf{DOM}) = \varepsilon}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \mathsf{mem}(\mathsf{a})], \mathsf{buf})}$$

$$\frac{\mathsf{cmd} = \texttt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}},\ \mathsf{a} := \widehat{e_{\mathsf{a}}},\ \mathsf{v} := \widehat{e_{\mathsf{v}}},\ \mathsf{id} := \mathsf{sn}(\mathsf{tid}),\ \beta := \mathsf{buf}(\mathsf{tid})}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{id},\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{buf}[\mathsf{tid} := \beta \cdot (\mathsf{id}, \mathsf{a}, \mathsf{v})])}$$

$$\frac{\mathsf{buf}(\mathsf{tid}) = (\mathsf{id}, \mathsf{a}, \mathsf{v}) \cdot \beta}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{id},\mathsf{flush})} (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}[\mathsf{a} := \mathsf{v}], \mathsf{buf}[\mathsf{tid} := \beta])}$$

$$\frac{\mathsf{cmd} = \texttt{mfence}, \quad \mathsf{buf}(\mathsf{tid}) = \varepsilon}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{buf})}$$

$$\frac{\mathsf{cmd} = r \leftarrow e}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \widehat{e}], \mathsf{buf})}$$

$$\frac{\mathsf{cmd} = \texttt{assume}(e), \quad \widehat{e} \neq 0}{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf}) \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{buf})}$$

Table 2.2: TSO transition rules, assuming $\mathsf{pc}(\mathsf{tid}) = q$, an instruction $\mathsf{instr} = q \xrightarrow{\mathsf{cmd}} q'$, $\mathsf{pc}' := \mathsf{pc}[\mathsf{tid} := q']$, and $\mathsf{sn}' := \mathsf{sn}[\mathsf{tid} := \mathsf{sn}(\mathsf{tid}) + 1]$. As $\widehat{e}$ we denote the value of expression e in thread tid and memory configuration mem.

the exclusive access to the memory [72].

Consider a multithreaded program $\mathcal{P}$, as defined in Section 2.3. We add to the set CMD of the program commands the x86 `mfence` instruction. The TSO semantics of program $\mathcal{P}$ is an automaton $X_{\mathsf{tso}}(\mathcal{P}) := (S_{\mathsf{tso}}, \mathrm{E}_{\mathsf{tso}}, \Delta_{\mathsf{tso}}, s_{\mathsf{tso}0}, F_{\mathsf{tso}})$. The state $s \in S_{\mathsf{tso}}$ is a tuple $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf})$, where $\mathsf{sn}\colon \mathsf{TID} \to \mathbb{N}$ is a vector of counters used for identifying events, $\mathsf{pc}(\mathsf{tid}) \in Q_{\mathsf{tid}}$ is the control state of the thread tid, $\mathsf{mem}\colon \mathsf{TID} \times \mathsf{REG} \cup \mathsf{ADDR} \to \mathsf{DOM}$ is the memory configuration, and $\mathsf{buf}\colon \mathsf{TID} \to (\mathbb{N} \times \mathsf{ADDR} \times \mathsf{DOM})^*$ is the configuration of the store buffers, storing sequence of address-value pairs, together with the identifiers of the matching store events. The initial state is $s_0 := (\mathsf{sn}_0, \mathsf{pc}_0, \mathsf{mem}_0, \mathsf{buf}_0)$, where $\mathsf{sn}_0 := \lambda\mathsf{tid}.1$, all threads are in the initial control states $\mathsf{pc}_0(\mathsf{tid}) := q_{\mathsf{tid}0}$, the memory is filled with zeroes: $\mathsf{mem}_0(\mathsf{a}) := 0$ for all $\mathsf{a} \in \mathsf{TID} \times \mathsf{REG} \cup \mathsf{ADDR}$, and the buffers are empty: $\mathsf{buf}_0(\mathsf{tid}) := \varepsilon$.

The transition relation is the smallest relation defined by the rules from Table 2.2. The rules repeat, up to notation and support for `locked` instructions, Figure 1 from [72]. The first two rules implement loads from the buffer and from the memory respectively. By the third rule, store instructions enqueue write operations to the buffer. The fourth rule non-deterministically dequeues and executes them on memory. The fifth rule defines that memory fences can only be executed when the buffer is empty. The last two rules refer to local assignments

and assertions. We omitted `lock`ed instructions to keep the constructions and proofs simple. Their handling is straightforward, similar to `mfence`, and does not affect the results. Our robustness checking tool Trencher presented in Chapter 7 supports `lock`ed instructions.

The final states are all the states with the buffers empty: $F_{\mathsf{tso}} := \{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \lambda\mathsf{tid}.\varepsilon) \in S_{\mathsf{tso}}\}$. We need the requirement of empty buffers for the simplicity of future definitions. Note that one can reach a final state from any state by flushing the buffers.

The set of *TSO computations* of a program is $\mathsf{C}_{\mathsf{tso}}(\mathcal{P}) := \mathcal{L}(X_{\mathsf{tso}}(\mathcal{P}))$.

Example 2.4. The following computation is a TSO computation of the SB program (Figure 1.1):

$$\tau := abc \cdot \mathsf{flush}(c) \cdot d \cdot \mathsf{flush}(a),$$

where

- $a := (1, 1, q_1 \xrightarrow{\texttt{mem}[x]\leftarrow 1} q_2, x)$,
- $b := (1, 2, q_2 \xrightarrow{r_1\leftarrow\texttt{mem}[y]} q_3, y)$,
- $c := (2, 1, q_1 \xrightarrow{\texttt{mem}[y]\leftarrow 1} q_2, y)$,
- $\mathsf{flush}(c) := (2, 1, \mathsf{flush})$,
- $d := (2, 2, q_2 \xrightarrow{r_2\leftarrow\texttt{mem}[x]} q_3, x)$,
- $\mathsf{flush}(a) := (1, 1, \mathsf{flush})$.

In this computation the first thread delays the flush of the store to variable x until the end of the computation. As a result, both threads read the initial values from variables x and y, which is impossible under SC.

Fix a computation $\sigma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ and an event $\mathsf{e} \in \sigma$. As $\mathsf{tid}(\mathsf{e})$ we denote the thread id of the event, i.e., the first component of e. As $\mathsf{id}(\mathsf{e})$ we denote the serial number of the event, i.e., the second component of e. As $\mathsf{instr}(\mathsf{e})$ we denote the instruction that created the event. For non-flush events, $\mathsf{instr}(\mathsf{e})$ is the third component of the tuple e. For flush events, $\mathsf{instr}(\mathsf{e})$ is equal to that of the *matching* store event (the event in σ having the same id and tid). Moreover, for load, store, and flush events we use $\mathsf{addr}(\mathsf{e})$ to refer to the address being accessed (the last component of the tuple for load and store events, the address of the matching store event for flush events). For instance, in Example 2.4 we have $\mathsf{tid}(a) = 1$, $\mathsf{id}(a) = 1$, $\mathsf{instr}(a) = q_1 \xrightarrow{\texttt{mem}[x]\leftarrow 1} q_2$, $\mathsf{addr}(a) := x$.

2.5 State Reachability

In the thesis we will use state reachability as a target for several reductions. This section provides the complexity and decidability results we will need.

Problem 2.5 (State reachability under mm)**.** Given a relaxed memory model mm, a program $\mathcal{P}$, a thread tid, and a control state $q \in Q_{\mathsf{tid}}$, to check whether the program reaches under mm a final state s with the control state of tid being q.

We defer formal definition of what is a control state of a given thread in a running program to the definition of the corresponding memory model semantics. However, for SC and TSO, the state reachability problem asks whether a state $(\mathsf{sn}, \mathsf{pc}, \mathsf{mem})$, respectively $(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{buf})$, with $\mathsf{pc}(\mathsf{tid}) = q$ is reachable.

The state reachability problem can be formulated for the parameterized setting, when the number of running threads (e.g., library clients) is not fixed a priori. In this setting, instead of considering a single program $\mathcal{P}$, we consider a family of programs $\{\mathcal{P}(I) \mid I \in \mathbb{N}^{\mathsf{TID}}\}$. A *program instance* $\mathcal{P}(I)$ of *parameterized program* $\mathcal{P}$ consists of $I(\mathsf{tid})$ copies of thread $\mathcal{T}_{\mathsf{tid}}$, $\mathsf{tid} \in \mathsf{TID}$. Parameterized state reachability asks if at least one instance of the parameterized program can reach a certain state.

Problem 2.6 (Parameterized state reachability under mm)**.** Given a relaxed memory model mm, a program $\mathcal{P}$, a thread tid and a control state $q \in Q_{\mathsf{tid}}$, to check whether there is $I \in \mathbb{N}^{\mathsf{TID}}$, such that $\mathcal{P}(I)$ reaches under mm a final state s with the control state of at least one copy of thread tid being q.

On the assumption of a finite data domain DOM, the following decidability and complexity results hold.

Lemma 2.7 ([52])**.** *State reachability under SC is* PSPACE*-complete.*

Lemma 2.8. *Parameterized state reachability under SC is decidable.*

Proof. Let $\mathcal{P}$ be a parameterized program with finite data domain DOM. We reduce parameterized SC state reachability to the coverability problem for a Petri net $N = (P, T, W)$. We construct the Petri net as follows.

For each pair of address and value $(\mathsf{a}, \mathsf{v}) \in \mathsf{ADDR} \times \mathsf{DOM}$ we create a place $\mathsf{p}_{\mathsf{a},\mathsf{v}}$. These places represent the state of the global memory: a marking M with $M(\mathsf{p}_{\mathsf{a},\mathsf{v}}) = 1$ corresponds to a state with $\mathsf{mem}(\mathsf{a}) = \mathsf{v}$. For each thread tid, control state $q \in Q_{\mathsf{tid}}$, $\overline{\mathsf{v}} \in \mathsf{DOM}^{\mathsf{REG}}$, we create a place $\mathsf{p}_{\mathsf{tid},q,\overline{\mathsf{v}}}$. $M(\mathsf{p}_{\mathsf{tid},q,\overline{\mathsf{v}}})$ gives the number of instances of thread tid in control state q with the valuation of registers being $\overline{\mathsf{v}}$.

For each thread tid we create a transition $\mathsf{t}_{\mathsf{tid}}$ that spawns instances of this thread in the initial state. We set $W(\mathsf{t}_{\mathsf{tid}}, \mathsf{p}_{q_{\mathsf{tid}0},0^{\mathsf{REG}}}) := 1$. Next we create transitions that simulate the instructions in each thread. We explain the construction for load instructions. The other instructions are handled similarly.

Consider a load instruction $\mathsf{instr} := q_1 \xrightarrow{r \leftarrow \mathtt{mem}[e_{\mathsf{a}}]} q_2$. For each $\overline{\mathsf{v}} \in \mathsf{DOM}^{\mathsf{REG}}$, $\mathsf{v} \in \mathsf{DOM}$ we add a fresh transition $\mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}$. Let a be the value of e_{a} for register valuation $\overline{\mathsf{v}}$. We set $W(\mathsf{p}_{\mathsf{tid},q_1,\overline{\mathsf{v}}}, \mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}) := W(\mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}, \mathsf{p}_{\mathsf{tid},q_2,\overline{\mathsf{v}}[r:=\mathsf{v}]}) := 1$. We set $W(\mathsf{p}_{\mathsf{a},\mathsf{v}}, \mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}) := W(\mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}, \mathsf{p}_{\mathsf{a},\mathsf{v}}) := 1$. Transition $\mathsf{t}_{\mathsf{tid},\mathsf{instr},\overline{\mathsf{v}},\mathsf{v}}$ is enabled iff there is an instance of the thread tid in control state q_1, register valuation $\overline{\mathsf{v}}$, and memory contains value v at address a. Firing the transition only updates the state of the thread instance: its program counter is set to label q_2, and the value of register r is set to v.

We define the initial marking by $M_0(\mathsf{p}_{\mathsf{a},0}) := 1$ for all $\mathsf{a} \in \mathsf{DOM}$, $M_0(\mathsf{p}) := 0$ for all other places $\mathsf{p} \in P$. A state with at least one copy of thread tid being in state q is SC-reachable in the parameterized program iff at least one of the markings M with $M(\mathsf{p}_{\mathsf{tid},q,\overline{\mathsf{v}}}) = 1$, $\overline{\mathsf{v}} \in \mathsf{DOM}^{\mathsf{REG}}$, is coverable in the constructed Petri net.

Altogether, by Lemma 2.2 and the above reduction, the parameterized SC state reachability problem is decidable. □

Lemma 2.9. *Parameterized state reachability under SC is* ExpSpace*-hard already for programs with* $|\mathsf{DOM}| \geq 3$.

Proof. Fix a Petri net $N := ((P, T, W), M_0)$ and a marking M. We reduce coverability problem for the Petri net N and marking M to the SC state reachability problem in a parameterized program $\mathcal{P}$. The program $\mathcal{P}$ consists of the threads $\{\mathcal{T}_\mathsf{p} \mid \mathsf{p} \in P\}$ and a supervisor thread $\mathcal{T}$. Each thread $\mathcal{T}_\mathsf{p}$ has the initial control state q_0 and a control state q_1. The number of instances of thread $\mathcal{T}_\mathsf{p}$ in control state q_1 will indicate the marking of the place p. The supervisor thread $\mathcal{T}$ controls transitions of the threads $\mathcal{T}_\mathsf{p}$ from q_0 to q_1 and back.

Thread $\mathcal{T}$ will need $2{\cdot}|P|$ different messages to communicate with the threads $\mathcal{T}_\mathsf{p}$: messages A_p to signal that thread $\mathcal{T}_\mathsf{p}$ must enter q_1 and messages B_p to signal that the state must be left. These messages can be represented as bit vectors of length $\lceil \log(2 \cdot |P|) \rceil$ and transmitted using the following protocol. In order to send a bit vector, $\mathcal{T}$ stores the bits of this vector to address x, waiting after each store until the variable x again becomes 0, i.e., sending a bit $\top$ looks as follows:

$$q_{x1} \xrightarrow{\texttt{mem[}x\texttt{]}\leftarrow\top} q_{x2} \xrightarrow{r\leftarrow\texttt{mem[}x\texttt{]}} q_{x3} \xrightarrow{\texttt{assume(}r\texttt{=0)}} q_{x4}.$$

In order to receive a bit vector, $\mathcal{T}_\mathsf{p}$ loads x, checks that it is not 0, and then stores 0 to x, i.e., receiving one bit into register r looks as follows:

$$q_{x1} \xrightarrow{r\leftarrow\texttt{mem[}x\texttt{]}} q_{x2} \xrightarrow{\texttt{assume(}r\texttt{)}\neq 0} q_{x3} \xrightarrow{\texttt{mem[}x\texttt{]}\leftarrow 0} q_{x4}.$$

In order to make sure that only one thread $\mathcal{T}_\mathsf{p}$ is currently receiving a bit vector, we need a flag y. Before starting to receive a bit vector, the thread atomically changes the value at y from 0 to 1 using compare and swap (we can assume that it is provided). After receiving a message destined to this thread, the thread writes 0 to y. If the thread has received a message destined to another thread (or a message that asks for a transition it cannot perform), it keeps 1 in y, thus blocking further communication. Note that if a receiving thread tries to read a bit before it is actually written or a sending thread tries to check the acknowledgement before the receiving thread does it, the communication blocks as well.

The threads $\mathcal{T}_\mathsf{p}$ have a path from q_0 to q_1 that involves receiving a bit vector representing A_p and a path from q_1 to q_0 that receives a bit vector representing B_p.

The supervisor thread from the initial state q_0 has a transition to q_1 that atomically changes the value at address z from 0 to 1 using compare-and-swap: this will limit the number of active supervisor instances to one. From the state q_1 it has a path to q_2 consisting, for each $\mathsf{p} \in P$, of a send of $M_0(\mathsf{p})$ message A_p (to reach the initial marking). From the control state q_2, for each transition $\mathsf{t} \in T$, we add a loop path from q_2 to q_2 that involves sending $W(\mathsf{p}, \mathsf{t})$ messages B_p for each $\mathsf{p} \in P$ followed by sending $W(\mathsf{t}, \mathsf{p})$ messages A_p for each $\mathsf{p} \in P$. Finally, we add a path from $q_2 \to q_3$ consisting of $M(\mathsf{p})$ transitions B_p for each $\mathsf{p} \in P$ and checking that the value of y is 0 in the end. The marking is coverable iff the supervisor thread can reach the control state q_3 under SC.

Note that in the construction we used only three addresses (x, y, z), each containing at most three different values. By the above reduction and Lemma 2.2, parameterized state reachability is ExpSpace-hard already for programs with $|\mathsf{DOM}| \geq 3$.

□

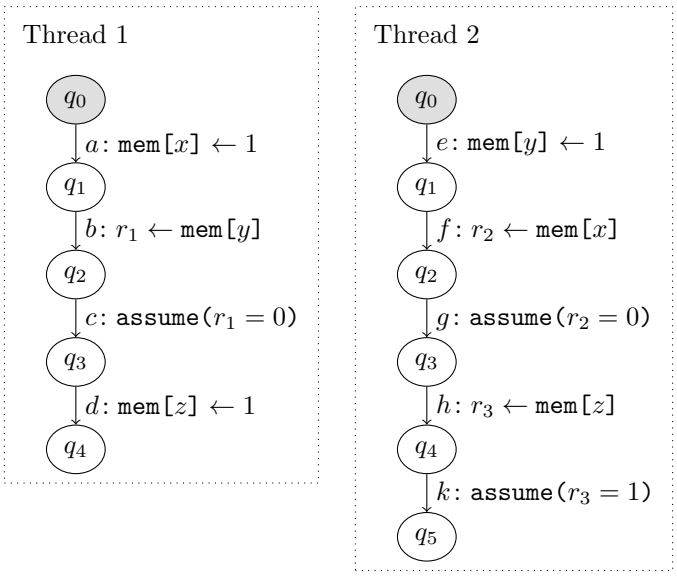

Figure 2.1: SB+ program, a non-state-robust version of the SB program from Figure 1.1. Initially, $x = y = 0$.

2.6 State-Robustness

The state-robustness problem consists in checking whether the threads of a program reach the same control states under SC and under a relaxed memory model.

Problem 2.10 (State-robustness against mm)**.** Given a relaxed memory model mm, a program $\mathcal{P}$, check, for each tid $\in$ TID and $q \in Q_{\mathsf{tid}}$ that if the program can reach under mm a final state s with the control state of tid being $q \in Q_{\mathsf{tid}}$, then it can reach a final state s' under SC with the control state of tid being q.

Example 2.11. Consider the SB+ program shown in Figure 2.1. It implements a mutual exclusion protocol, as explained in Example 1.1. We modified thread 1 to signal entering the critical section by setting variable z to 1. Thread 2, once it enters the critical section, checks whether z contains 1 and, if it so, enters state q_5. Since mutual exclusion fails under TSO, the program SB+ is not state-robust against TSO (and all weaker memory models): thread 2 can reach state q_5 under TSO, but not under SC.

Theorem 2.12. *State-robustness against* mm $\in \{$power, rmo, pso, tso, pgas$\}$ *is as hard as reachability under* mm*.*

Proof. The upper bound follows from a reduction of state-robustness to state reachability. One can enumerate all threads and all control states in each thread and check, for each control state, whether it is reachable under SC and under mm.

The lower bound follows from a reduction in the opposite direction. Consider a state reachability problem for a relaxed memory model mm, program $\mathcal{P}$, a thread tid and a control state $q_f \in Q_{\mathsf{tid}}$. We reduce the problem to state

robustness. We give the reduction for CPU memory models, the reduction for PGAS differs only in syntax.

In each thread of the original program we create a new initial state q_0' and create no-op transitions (e.g., labeled with `assume(1)`) from this state to all the states of this thread. Additionally, we create transitions

$$q_0' \xrightarrow{r \leftarrow \texttt{mem[cnt]}} q_{x1} \xrightarrow{\texttt{mem[cnt]} \leftarrow r+1} q_0, \tag{2.1}$$

where q_0 is the old initial state. We assume that r is a fresh register, cnt is a fresh address, and q_{x*} are fresh states not used in the original program (thread).

In thread tid we additionally create the following transitions:

$$q_f \xrightarrow{r \leftarrow \texttt{mem[cnt]}} q_{x2} \xrightarrow{\texttt{assume}(r = |\mathcal{P}|)} q_{x3} \xrightarrow{\texttt{mem[go]} \leftarrow 1} q_{x4}, \tag{2.2}$$

$$q_0' \xrightarrow{\texttt{assume(1)}} q_{x2}, \tag{2.3}$$

where go is a fresh address not used in the original program.

Finally, we extend the original program with additional threads that check whether address go contains 1 and, if it does, violate state-robustness, by e.g., executing the SB+ program (Figure 2.1).

Note that thread tid of the original program can reach control state q_f under mm iff thread tid of the modified program can reach control state q_{x3} under this model. Indeed, the modified program can perform transitions (2.1) in each thread, after which perform the exactly same transitions as the original program, reaching q_f. Due to executed (2.1), address cnt contains $|\mathcal{P}|$, and q_{x3} is reachable. The reverse implication is proven similarly.

Now assume the modified program is state-robust against mm. This means, the SB+ subprogram never runs, i.e., go is never set to 1, and thread tid in the modified program never reaches q_{x3}. Therefore, it does not reach q_f in the original program.

Assume the modified program is not state-robust against mm. This means, thread tid can reach q_{x3} under mm, but not under SC (all other states in the modified threads of the original program are reachable via a single transition from q_0'). Therefore, q_f is reachable by thread tid in the original program. □

2.7 Traces

Intuitively, a *trace* $T(\sigma)$ abstracts a program computation $\sigma \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$ to the dataflow and control-flow relations between executed instructions. Formally, the trace of σ is a directed graph $T(\sigma) := (V, \rightarrow_{po}, \rightarrow_{co}, \rightarrow_{src}, \rightarrow_{cf})$ with nodes V and four kinds of arcs. The nodes are instructions together with their thread identifiers and serial numbers (in order to distinguish instructions executed in different threads and the same instruction executed multiple times in the same thread), plus special nodes for the initial stores: $V \subseteq \bigcup_{\mathsf{tid} \in \mathsf{TID}} \{\mathsf{tid}\} \times \mathbb{N} \times \mathcal{I}_{\mathsf{tid}} \cup \{\mathsf{init}_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{ADDR}\}$. The *program order* $\rightarrow_{po}$ is the order in which instructions were executed in each thread. The *coherence order* $\rightarrow_{co}$ gives the global ordering of stores to each address. The *source order* $\rightarrow_{src}$ connects a store with the load that read from it. The *conflict order* $\rightarrow_{cf}$ shows, for a load, the store to the same address following the store from which the load took its value.

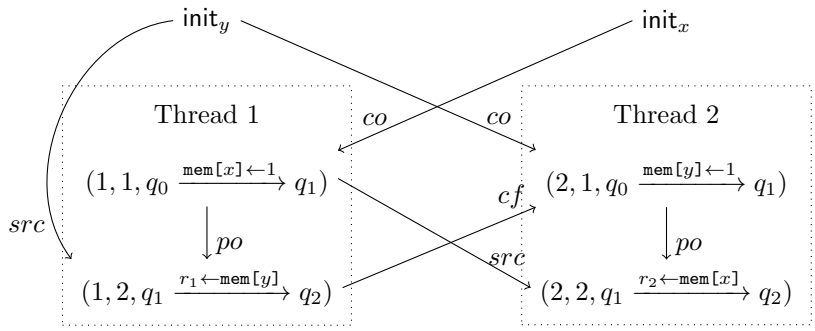

Figure 2.2: Trace of computation σ from Example 2.3.

2.7.1 SC Traces

For SC computations we formally define traces as follows. Fix a computation $\sigma \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$. Let $\mathsf{e}_{i_1} \dots \mathsf{e}_{i_m}$ be the longest subsequence of events in σ with $\mathsf{tid}(\mathsf{e}_{i_k}) = \mathsf{tid}$ for all $k \in [1..m]$. Then $\mathsf{e}_{i_1} \to_{po} \dots \to_{po} \mathsf{e}_{i_m}$. Here and further, for convenience, we abbreviate $(\mathsf{tid}(\mathsf{e}_1), \mathsf{id}(\mathsf{e}_1), \mathsf{instr}(\mathsf{e}_1)) \to (\mathsf{tid}(\mathsf{e}_2), \mathsf{id}(\mathsf{e}_2), \mathsf{instr}(\mathsf{e}_2))$ to $\mathsf{e}_1 \to \mathsf{e}_2$.

The coherence order $\to_{co}$ is the minimal relation satisfying the following. Let $\mathsf{e}_{i_1} \dots \mathsf{e}_{i_m}$ be the longest subsequence of store events in σ with $\mathsf{addr}(\mathsf{e}_{i_k}) = \mathsf{a}$ for all $k \in [1..m]$. Then $\mathsf{init}_{\mathsf{a}} \to_{co} \mathsf{e}_{i_1} \to_{co} \dots \to_{co} \mathsf{e}_{i_m}$, where $\mathsf{init}_{\mathsf{a}}$ is a special node for the initial store to address a.

The source order $\to_{src}$ is the minimal relation satisfying the following. Let $\sigma = \sigma_1 \cdot \mathsf{e}_1 \cdot \sigma_2 \cdot \mathsf{e}_2 \cdot \sigma_3$, e_1 is a store, e_2 is a load, $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2) = \mathsf{a}$, and there is no store event $\mathsf{e} \in \sigma_2$ with $\mathsf{addr}(\mathsf{e}) = \mathsf{a}$. Then $\mathsf{e}_1 \to_{src} \mathsf{e}_2$. If $\sigma = \sigma_1 \cdot \mathsf{e}_2 \cdot \sigma_2$, e_2 is a load, and there is no store $\mathsf{e}_1 \in \sigma_1$ with $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2)$, then $\mathsf{init}_{\mathsf{addr}(\mathsf{e}_2)} \to_{src} \mathsf{e}_2$.

Finally, the conflict order is the minimal relation satisfying the following. If $\mathsf{e}_1 \to_{src} \mathsf{e}_2$ and $\mathsf{e}_1 \to_{cf} \mathsf{e}_3$, then $\mathsf{e}_1 \to_{cf} \mathsf{e}_3$.

Example 2.13. Figure 2.2 shows the trace of computation σ from Example 2.3. The coherence order shows that the stores to x and y overwrite the values written by the initial stores. The source relation shows that the load from y in the first thread reads the value written by the initial store, and the load from x reads the value written by the store to x in the first thread. The conflict relation shows that the store of the second thread overwrites the value read by the load in the first thread.

2.7.2 TSO Traces

Now fix a TSO computation $\tau \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$. We define its trace $T(\tau) := (V, \to_{po}, \to_{co}, \to_{src}, \to_{cf})$ as follows.

Let $\mathsf{e}_{i_1} \dots \mathsf{e}_{i_m}$ be the longest subsequence of non-flush events in τ with $\mathsf{tid}(\mathsf{e}_{i_k}) = \mathsf{tid}$ for all $k \in [1..m]$. Then the program order $\to_{po}$ is the successor relation for this subsequence: $\mathsf{e}_{i_1} \to_{po} \dots \to_{po} \mathsf{e}_{i_m}$.

The coherence order $\to_{co}$ is the minimal relation satisfying the following. Let $\mathsf{e}_{i_1} \dots \mathsf{e}_{i_m}$ be the longest subsequence of flush events in τ with $\mathsf{addr}(\mathsf{e}_{i_k}) = \mathsf{a}$

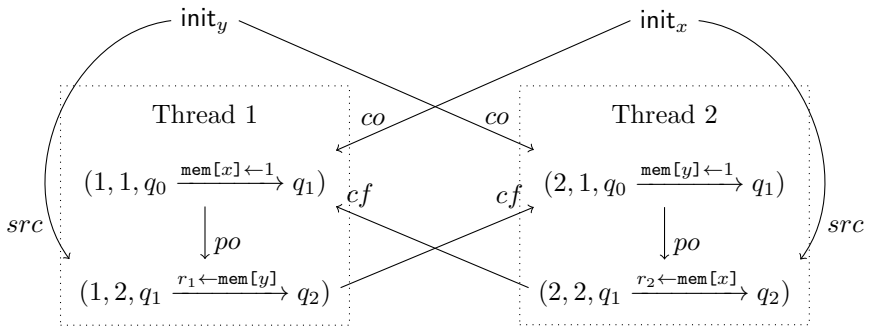

Figure 2.3: Trace of computation τ from Example 2.4.

for all $k \in [1..m]$. Then $\mathsf{init_a} \rightarrow_{co} \mathsf{e}_{i_1} \rightarrow_{co} \ldots \rightarrow_{co} \mathsf{e}_{i_m}$, where $\mathsf{init_a}$ is a special node for the initial store to address a.

The source order $\rightarrow_{src}$ requires case consideration. Consider a load event $\mathsf{e}_2 \in \tau$ with $\mathsf{tid}(\mathsf{e}_2) = \mathsf{tid}$, $\mathsf{addr}(\mathsf{e}_2) = \mathsf{a}$. The early read case: if $\tau = \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3 \cdot \tau_4$, where $\mathsf{tid}(\mathsf{e}_1) = \mathsf{tid}$, $\mathsf{addr}(\mathsf{e}_1) = \mathsf{a}$, e_1 is a matching store event of flush event e_3, there are no store events $\mathsf{e} \in \tau_2$ with $\mathsf{tid}(\mathsf{e}) = \mathsf{tid}$ and $\mathsf{addr}(\mathsf{e}) = \mathsf{a}$, then $\mathsf{e}_1 \rightarrow_{src} \mathsf{e}_2$. Otherwise, the load from memory case: if $\tau = \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3$, e_1 is a flush event with $\mathsf{addr}(\mathsf{e}_1) = \mathsf{a}$, and there are no flush events $\mathsf{e} \in \tau_2$ with $\mathsf{addr}(\mathsf{e}) = \mathsf{a}$, then $\mathsf{e}_1 \rightarrow_{src} \mathsf{e}_2$. Finally, if $\tau = \tau_1 \cdot \mathsf{e}_2 \cdot \tau_2$ and there is no flush event $\mathsf{e} \in \tau_1$ with $\mathsf{addr}(\tau_1) = \mathsf{a}$, then $\mathsf{init_a} \rightarrow_{src} \mathsf{e}_2$.

The conflict order $\rightarrow_{cf}$ shows which store overwrites the value read by a load. If $\mathsf{e}_1 \rightarrow_{src} \mathsf{e}_2$ and $\mathsf{e}_1 \rightarrow_{co} \mathsf{e}_3$, then $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_3$.

Example 2.14. Figure 2.3 shows the trace of computation τ from Example 2.4. The source order indicates that both loads read the initial values of variables x and y. The stores to x and y follow the initial stores to these variables in the coherence order and, consequently, conflict with the loads.

2.8 Robustness

Let $T_{\mathsf{mm}}(\mathcal{P})$ denote the set of traces of all computations of program $\mathcal{P}$ under memory model mm: $T_{\mathsf{mm}}(\mathcal{P}) := \{T(\sigma) \mid \sigma \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})\}$. The *robustness* problem consists in checking whether the program has the same set of traces under SC and under a relaxed memory model.

Problem 2.15 (Robustness against mm)**.** Given a relaxed memory model mm, and a program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}) = T_{\mathsf{mm}}(\mathcal{P})$.

We call a program *robust against a memory model* mm if the above equality holds. For all the memory models considered in the thesis the inclusion $T_{\mathsf{sc}}(\mathcal{P}) \subseteq T_{\mathsf{mm}}(\mathcal{P})$ trivially holds, so, robustness amounts to checking the reverse inclusion.

The union of the four relations in $T(\sigma)$ is commonly called the *happens-before* relation [55] of the computation: $\rightarrow_{hb}(\sigma) := \rightarrow_{po} \cup \rightarrow_{co} \cup \rightarrow_{src} \cup \rightarrow_{cf}$. Shasha and Snir have shown that a trace belongs to an SC computation iff the happens-before relation is acyclic.

Lemma 2.16 ([79])**.** *Consider a computation $\sigma \in \mathsf{C}_{mm}(\mathcal{P})$. Then $T(\sigma) \in T_{\mathsf{sc}}(\mathcal{P})$ iff $\rightarrow_{hb}(\sigma)$ is acyclic.*

Example 2.17. The SC computation σ from Example 2.3 has a trace with acyclic happens-before relation (Figure 2.2).

Example 2.18. The computation τ from Example 2.4 has a trace with the cyclic happens-before relation (Figure 2.3). Indeed, the program is not robust against TSO: there is no SC computation where both loads could read the initial value 0.

Trace-robustness implies state-robustness, which means that one can verify trace-robust programs under SC and be sure that verification results carry over to the relaxed setting.

Theorem 2.19. *Trace-robustness against $\mathsf{mm} \in \{\mathsf{power}, \mathsf{rmo}, \mathsf{pso}, \mathsf{tso}, \mathsf{pgas}\}$ implies state-robustness against mm.*

Proof. If a program has the same set of traces under SC and under a memory model mm, then, by definition of $\rightarrow_{po}$ component of traces, all its threads reach the same control states under SC and mm, i.e., the program is state-robust against mm. □

Similar to the parameterized state reachability (Section 2.5), we can define the robustness problem for parameterized programs. It consists in checking whether each instance of a parameterized program is robust.

Problem 2.20 (Parameterized robustness against mm)**.** Given a relaxed memory model mm, and a parameterized program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}(I)) = T_{\mathsf{mm}}(\mathcal{P}(I))$ for all $I \in \mathbb{N}^{\mathsf{TID}}$.

Parameterized robustness is an interesting problem, useful, for example, for verification of concurrent libraries. These libraries generally cannot assume that the number of threads calling library functions is bounded. While the robustness problem asks whether a given library works correctly with the given finite set of clients, the parameterized robustness problem asks whether a library works correctly for any number of clients.

Chapter 3

Generic Approach to Robustness

In this chapter we present a generic approach to solving robustness of finite-state programs. This approach will be used in the next chapters to solve robustness against several real-world memory models.

The approach is based on Lemma 2.16: a program is not robust against a memory model if under this memory model it has computations with cyclic happens-before relation. This characterization immediately leads to a naive method for detecting non-robustness. One can enumerate program computations under a given memory model and, for each computation, check its happens-before relation for cyclicity. This method is the basis of, e.g., the monitoring algorithms by Burckhardt and Musuvathi [25] and Burnim et al. [26]. Unfortunately, this approach is generally unsuitable for proving robustness, as it requires enumerating all program computations, whose number is generally infinite.

The idea of our approach is to shift the problem from finding just *any* computation with cyclic happens-before relation to finding such a computation within a certain, restricted class of computations. Intuitively, the class must have the following two qualities. On one hand, it must be representative: the class must contain a computation with cyclic happens-before relation if the program has one. On the other hand, it should be as small as possible, to reduce the search space. We discover such a class using combinatorial analysis and then develop an algorithm for checking its emptiness for a given program.

Altogether, we solve robustness against a given memory model in two steps. In the first, combinatorial, step we show that, if a program has computations violating robustness, it has such a computation of a certain normal form. In the second, algorithmic, step we devise an algorithm for checking whether a program has violating normal-form computations.

3.1 Normal-Form Computations

We say that a computation $\tau \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$ is *in normal form of degree* n if there is a partitioning $\tau = \tau_1 \cdots \tau_n$, such that each instruction has its first event in τ_1

(NF-A) and events belonging[1] to different instructions occur in different parts of the computation in the same order (NF-B):

NF-A $\mathsf{e}_1 \in \tau$ implies existence of a matching $\mathsf{e}_2 \in \tau_1$ with $\mathsf{tid}(\mathsf{e}_1) = \mathsf{tid}(\mathsf{e}_2)$ and $\mathsf{id}(\mathsf{e}_1) = \mathsf{id}(\mathsf{e}_2)$.

NF-B For $j \in \{1, 2\}$ let e_j, e'_j be events belonging to instructions $(\mathsf{tid}_j, \mathsf{id}_j)$. If $\mathsf{e}_1, \mathsf{e}_2 \in \tau_s$ and $\mathsf{e}'_1, \mathsf{e}'_2 \in \tau_{s'}$, then $\mathsf{e}_1 <_{\tau_s} \mathsf{e}_2$ iff $\mathsf{e}'_1 <_{\tau_{s'}} \mathsf{e}'_2$.

Clearly, a normal-form computation of degree n is also a computation of degree $n + 1$ or any other higher degree.

Example 3.1. Computation τ from Example 2.4 is a normal-form computation of degree 2. Indeed, it can be partitioned as $\tau := \tau_1 \cdot \tau_2$, where $\tau_1 := abc \cdot \mathsf{flush}(c) \cdot d$ and $\tau_2 := \mathsf{flush}(a)$. The only event $\mathsf{flush}(a)$ in τ_2 has a matching event a in τ_1, so, NF-A holds. Also, τ_2 consists of only one event, therefore, NF-B trivially holds.

We show that among computations with cyclic happens-before relation there is a normal-form computation of a certain small degree. For this, we choose a computation with cyclic happens-before relation and transform it to the normal form as follows.

Consider a shortest computation $\tau \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$ with cyclic happens-before relation. One can show that we can undo the last executed instruction in one of the threads and obtain a feasible computation. Formally, let $\tau \setminus (\mathsf{tid}, \mathsf{id})$ be the computation obtained from σ by deleting all events belonging to instruction $(\mathsf{tid}, \mathsf{id})$. We show that there is $(\mathsf{tid}, \mathsf{id})$, such that $\tau' := \mathsf{tid} \setminus (\mathsf{tid}, \mathsf{id}) \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$ and $|\tau'| < |\tau|$. If $\tau = \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdots \tau_n$, where $\mathsf{e}_1 \dots \mathsf{e}_{n-1}$ are the events belonging to instruction $(\mathsf{tid}, \mathsf{id})$, then $\tau' := \tau_1 \cdots \tau_n$.

Since τ was the shortest computation with cyclic happens-before relation, happens-before relation of τ' is acyclic. By Lemma 2.16, there is an SC computation $\sigma \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$ with $T(\sigma) = T(\tau')$. We reorder events in each part of the computation τ in the way in which they follow in σ:

$$\tau'' := \sigma{\downarrow}\tau_1 \cdot \mathsf{e}_1 \cdot \sigma{\downarrow}\tau_2 \cdot \mathsf{e}_2 \cdots \sigma{\downarrow}\tau_n.$$

Here we implicitly assume that σ is extended to include all the events from τ. For example, in case of TSO it contains flush events right after the matching store events.

Finally, we show that $\tau'' \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$, it has the same trace as the original computation, $T(\tau'') = T(\tau)$, and is a normal-form computation, with the partitioning

$$\tau'' := (\sigma{\downarrow}\tau_1 \cdot \mathsf{e}_1) \cdot (\sigma{\downarrow}\tau_2 \cdot \mathsf{e}_2) \cdots (\sigma{\downarrow}\tau_n).$$

The number of parts n is bounded by the number of events that an execution of a single instruction can generate plus one. This number, for all memory models considered in the thesis, does not exceed the number of threads in the program plus a constant.

Example 3.2. Computation τ from Example 2.4 is a shortest TSO computation of SB program with cyclic happens-before relation. We can delete either event

[1] See page 11 for the definition of *belongs to an instruction*.

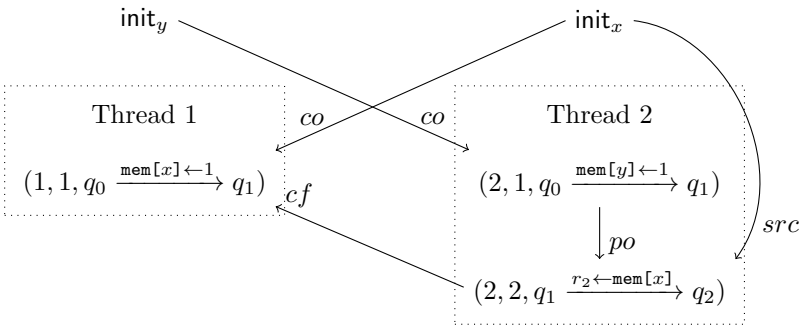

Figure 3.1: Trace of computations τ' and σ from Example 3.2.

b belonging to the last executed instruction in thread 1 or, symmetrically, event d of thread 2. Assume we delete b. Then $\tau_1 := a$, $\mathsf{e}_1 := b$, $\tau_2 := c \cdot \mathsf{flush}(c) \cdot d \cdot \mathsf{flush}(a)$. Then $\tau' := \tau_1 \cdot \tau_2$ has the same trace as sequentially consistent $\sigma := c \cdot \mathsf{flush}(c) \cdot da \cdot \mathsf{flush}(a)$, shown in Figure 3.1. Consequently, $\tau'' := (\sigma \downarrow \tau_1) \cdot \mathsf{e}_1 \cdot (\sigma \downarrow \tau_2) = (a) \cdot b \cdot (c \cdot \mathsf{flush}(c) \cdot d \cdot \mathsf{flush}(a))$, i.e., coincides with τ.

3.2 From Robustness to Language Emptiness

According to the previous section, a program is robust iff it does not have normal-form computations (of some small degree) with cyclic happens-before relation. In order to check robustness of a program, we construct the language of all normal-form computations and intersect it with a regular language checking cyclicity of happens-before relation. The program is robust iff the intersection is empty.

3.2.1 Multiheaded Automata

Normal-form computations do not bound the distance between events belonging to the same instruction. For example, under TSO a buffered store can remain in the buffer unboundedly long, and there can be an any number of such stores. Therefore, normal-form computations cannot be generated by classic finite automata: finite automata cannot keep information about unboundedly many concurrently executed instructions. However, we can use the fact that relative dispositions of events belonging to the same instructions are the same in each part of the computation (NF-B). The idea is to generate events belonging to one instruction in one shot, however, in different parts of the computation. For this, we enhance finite automata with the ability to generate multiple parts of a computation in parallel. The new class of automata is called *multiheaded automata*.

A multiheaded automaton generates a word $\sigma_1 \cdots \sigma_n$ by simultaneously generating its parts σ_i. The automaton has a head for each part, and the transition relation specifies the head producing an event. Formally, an *n-headed automaton over* Σ is an automaton operating on the extended alphabet $[1..n] \times \Sigma$: $A := (S, [1..n] \times \Sigma, \Delta, s_0, F)$. For a word $\sigma := (a_1, b_1) \dots (a_m, b_m)$ we de-

fine $\mathsf{take1st}(\sigma) := a_1 \dots a_m$ and $\mathsf{take2nd}(\sigma) := b_1 \dots b_m$. We naturally extend $\mathsf{take1st}()$ and $\mathsf{take2nd}()$ to sets of words. For $\sigma \in ([1..n] \times \Sigma)^*$ we define $\mathsf{word}(\sigma) := \mathsf{take2nd}(\sigma \downarrow (\{1\} \times \Sigma) \cdots \sigma \downarrow (\{n\} \times \Sigma))$. The *language of* A is $\mathcal{L}(A) := \{\mathsf{word}(\sigma) \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}$.

Multiheaded automata are closed under regular intersection, and emptiness is decidable in non-deterministic logarithmic space.

Lemma 3.3. *Consider an n-headed automaton U and a finite automaton V over a common alphabet Σ. There is an n-headed automaton W with $\mathcal{L}(W) = \mathcal{L}(U) \cap \mathcal{L}(V)$ with the number of states $|S_W| \leq |S_U| \cdot |S_V|^{2n} + 1$.*

Proof. Let $U = (S_U, \Sigma, \Delta_U, s_{U0}, F_U)$ and $V = (S_V, \Sigma, \Delta_V, s_{V0}, F_V)$. We set $W := (S_W, \Sigma, \Delta_W, s_{W0}, F_W)$. Let $\Omega := S_V^{[1..n]}$. Then, the set of states is $S_W := \{s_{W0}\} \uplus (S_U \times \Omega \times \Omega)$. The set of final states is $F_W := \{(s_U, \omega_1, \omega_2) \mid s_U \in F_U,\ \omega_1(n) \in F_V,\ \text{and } \omega_1(k) = \omega_2(k+1) \text{ for all } k \in [1..n-1]\}$. The automaton has the following transitions:

- $s_{W0} \xrightarrow{\varepsilon} (s_{U0}, \omega, \omega)$ for each $\omega \in \Omega$ with $\omega(1) = s_{V0}$,
- $(s_U, \omega_1, \omega_2) \xrightarrow{k,a} (s'_U, \omega'_1, \omega_2)$ if $s_U \xrightarrow{k,a} s'_U$, $\omega_1(k) \xrightarrow{a} \omega'_1(k)$, and $\omega_1(i) = \omega'_1(i)$ for $i \neq k$,
- $(s_U, \omega_1, \omega_2) \xrightarrow{\varepsilon} (s'_U, \omega_1, \omega_2)$ if $s_U \xrightarrow{\varepsilon} s'_U$,
- $(s_U, \omega_1, \omega_2) \xrightarrow{\epsilon} (s_U, \omega'_1, \omega_2)$ if $\omega_1(k) \xrightarrow{\varepsilon} \omega'_1(k)$ and $\omega_1(i) = \omega'_1(i)$ for $i \neq k$.

Consider $\alpha = \alpha_1 \cdots \alpha_n \in \mathcal{L}(U) \cap \mathcal{L}(V)$, where α_k is produced by the k^{th} head of U. By the ε-transition from the initial state, W guesses, for each k, the state $\omega(k)$ that the automaton V will reach after processing the prefix $\alpha_1 \cdots \alpha_{k-1}$ of α. The other transitions effectively execute the automaton U synchronously with n copies of the automaton V, each matching its own α_k subword of α, starting from the guessed initial state $\omega(k)$. The set of final states F_W makes sure that the guess was done correctly, which means the k^{th} copy of V has reached the initial state of the $k+1^{\text{th}}$ copy, and the n^{th} copy has reached a final state in F_V. □

Lemma 3.4. *Language emptiness for n-headed automata is* NL*-complete.*

Proof. See the proof of Lemma 2.1. □

Multiheaded automata are equal in expressiveness to right-linear scattered context grammars [41], assuming these grammars allow an initial word consisting of multiple non-terminals. Multiheaded automata are incomparable with context-free grammars. Indeed, they can be used to represent a language $a^n b^n c^n$, which is well-known to be non-context-free. On the other hand, $\sigma \cdot \sigma^{rev}$ is context-free but not accepted by a multiheaded automaton.

Example 3.5. Normal-form TSO computations of degree 2 can be generated by a multiheaded automaton of degree 2 defined as follows. The automaton, similar to $X_{\mathsf{tso}}(\mathcal{P})$ from Section 2.4.2, keeps in the state the current control state of each thread and the global memory configuration. However, instead of keeping the buffer contents, it remembers for every thread the last buffered value written to each address. The automaton handles all non-store instructions as before,

they produce events only in part 1, to make the computation satisfy NF-A. Store instructions produce two events: a store event in part 1 and a flush event in part 1 (store is not buffered) or part 2 (store is buffered) non-deterministically. Of course, the automaton must comply with the FIFO ordering and never produce flush events in part 1 if it already generated flush events for the same thread in part 2. This also guarantees that the computation satisfies NF-B. Computation τ from Example 2.4 would be generated by this automaton via a sequence of transitions $s_0 \xrightarrow{1,a} s_1 \xrightarrow{2,\mathsf{flush}(a)} s_2 \xrightarrow{1,b} s_3 \xrightarrow{1,c} s_4 \xrightarrow{1,\mathsf{flush}(c)} s_5 \xrightarrow{1,d} s_6$.

3.2.2 Checking Cyclicity of the Happens-Before Relation

Once we described the language of all normal-form computations of a program with a multiheaded automaton, we are going to check whether this language contains a computation with cyclic happens-before relation. This check combines several observations.

The first observation is the fact that, if a computation has a cycle, it has a *beautiful* cycle, where each thread contributes only once. Formally, we call a happens-before cycle beautiful, if it has the following form:

$$\begin{aligned}(\mathsf{tid}_1, i_1, \mathsf{instr}_1) \rightarrow^*_{po} (\mathsf{tid}_1, i'_1, \mathsf{instr}'_1) \rightarrow_{hop} \ldots \\ \rightarrow_{hop} (\mathsf{tid}_n, i_n, \mathsf{instr}_n) \rightarrow^*_{po} (\mathsf{tid}_n, i'_n, \mathsf{instr}'_n) \rightarrow_{hop} (\mathsf{tid}_1, i_1, \mathsf{instr}_1).\end{aligned}$$

Here, $\rightarrow_{hop} := (\rightarrow_{co} \cup \rightarrow_{src} \cup \rightarrow_{cf})$ and $\mathsf{tid}_k \neq \mathsf{tid}_l$ for $k \neq l$. We call $\theta := \mathsf{tid}_1 \ldots \mathsf{tid}_n$ the *profile* of the cycle.

Example 3.6. The happens-before cycle shown in Figure 2.3 is beautiful and has cycle profile $\theta := 1, 2$ (or $2, 1$).

Lemma 3.7. *A computation $\tau \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$ has a happens-before cycle iff it has a beautiful happens-before cycle.*

Proof. Consider an arbitrary happens-before cycle. It has the following form:

$$\begin{aligned}(\mathsf{tid}_1, i_1, \mathsf{instr}_1) \rightarrow^*_{po} (\mathsf{tid}_1, i'_1, \mathsf{instr}'_1) \rightarrow_{hop} \ldots \\ \rightarrow_{hop} (\mathsf{tid}_n, i_n, \mathsf{instr}_n) \rightarrow^*_{po} (\mathsf{tid}_n, i'_n, \mathsf{instr}'_n) \rightarrow_{hop} (\mathsf{tid}_1, i_1, \mathsf{instr}_1).\end{aligned}$$

Assume $\mathsf{tid}_l = \mathsf{tid}_m$ for some $l < m$. Fix these l and m. Then either $(\mathsf{tid}_l, i_l, \mathsf{instr}_l) \rightarrow^*_{po} (\mathsf{tid}_m, i_m, \mathsf{instr}_m)$ or $(\mathsf{tid}_m, i_m, \mathsf{instr}_m) \rightarrow^*_{po} (\mathsf{tid}_l, i_l, \mathsf{instr}_l)$. In the former case, τ has the following happens-before cycle which is shorter:

$$\begin{aligned}(\mathsf{tid}_1, i_1, \mathsf{instr}_1) \rightarrow^*_{po} (\mathsf{tid}_1, i'_1, \mathsf{instr}'_1) \rightarrow_{hop} \ldots \\ \rightarrow_{hop} (\mathsf{tid}_l, i_l, \mathsf{instr}_l) \rightarrow^*_{po} (\mathsf{tid}_m, i'_m, \mathsf{instr}'_m) \rightarrow_{hop} \ldots \\ \rightarrow_{hop} (\mathsf{tid}_n, i_n, \mathsf{instr}_n) \rightarrow^*_{po} (\mathsf{tid}_n, i'_n, \mathsf{instr}'_n) \rightarrow_{hop} (\mathsf{tid}_1, i_1, \mathsf{instr}_1).\end{aligned}$$

In the latter case, there is the following happens-before cycle which is also shorter:

$$(\mathsf{tid}_m, i_m, \mathsf{instr}_m) \rightarrow^*_{po} (\mathsf{tid}_l, i'_l, \mathsf{instr}'_l) \rightarrow_{hop} \ldots \rightarrow_{hop} (\mathsf{tid}_m, i_m, \mathsf{instr}_m).$$

Repeating the shortening procedure for the new cycle until there is no $l \neq m$ with $\mathsf{tid}_l = \mathsf{tid}_m$, we get a beautiful cycle. □

The second observation is the fact that the multiheaded automaton has to generate events of each thread in program order (NF-A). We can modify the multiheaded automaton to mark two events in each thread, one after another, and these events are guaranteed to be in the program order.

The third observation is the fact that the existence of a single happens-before arc between two given events can be checked by a finite automaton. For example, in case of TSO, to check, whether a load a conflicts with a store b, it is sufficient to check that both events have the same address, a is before the flush of b, and there are no flush events with the same address in between.

Altogether, the algorithm for checking robustness enumerates all possible cycle profiles. For each profile it constructs the augmented multiheaded automaton picking pairs of events in program order, intersects it with the regular finite automata (at most $|\mathcal{P}|$ of them) checking the existence of arcs between the marked events in different threads, and checks the emptiness of this intersection.

Example 3.8. The happens-before cycle shown in Figure 2.3 can be detected as follows. The algorithm chooses a cycle profile $\theta := 1, 2$. The multiheaded automaton marks $\mathsf{flush}(a)$ and b events in thread 1 in program order, $\mathsf{flush}(c)$ and d events in thread 2 in program order. Two finite automata detect the conflicts $d \rightarrow_{cf} \mathsf{flush}(a)$ and $b \rightarrow_{cf} \mathsf{flush}(c)$.

Chapter 4

Robustness against Power

Power [32] is a RISC architecture developed by IBM and several other companies. Power has a highly relaxed memory model. First, the model permits threads to execute independent instructions out of order. Second, the threads can observe stores to different addresses in different order. In this chapter we study robustness against Power memory model.

There are few works that address robustness against (a fragment of) Power. Alglave and Maranget [10] presented a tool that overapproximates the set of happens-before cycles in a given x86 or Power assembler program and inserts memory barriers to forbid these cycles. Alglave et al. [7] combine this approach with integer linear programming to compute a minimal set of barriers that must be inserted into a C program to eliminate all potential happens-before cycles under a given memory model.

Although the above methods can compute a set of fences that makes the program robust, they cannot be used to show robustness. In this chapter we present an algorithm for deciding robustness against Power. The algorithm is an emptiness check for multiheaded automata. We reduce robustness to the emptiness check as follows. First, we show that if a program is not robust, it has a normal-form computation with cyclic happens-before relation. Second, we show how to describe the set of all normal-form computations using multiheaded automata. Finally, we use an intersection with finite automata to filter only those normal-form computations, which have cyclic happens-before relation. The reduction gives us a PSpace procedure for checking robustness against Power. The problem is PSpace-complete, by a reduction of SC reachability to robustness. These results are also published in [33].

In Section 4.5 we digress from the robustness topic and show that state reachability under Power is generally undecidable. This fact makes robustness, combined with verification under SC, a preferable alternative to direct verification under Power.

Related work. Power memory model is rather complex, and there are multiple works devoted to formally defining it. Alglave et al. [11] give an extensive overview of related publications. We would like to highlight two of them: the operational model by Sarkar et al. [78] and the axiomatic model by Mador-Haim et al. [65]. The two models were heavily tested against the hardware. Nevertheless, the operational model is known to forbid certain behaviors that are

possible on real hardware[1] and in the axiomatic model[2] [11]. This is why in this chapter we stick to a *corrected operational model*, obtained from the original one by replacing *from a different write* by *from a coherence-order-earlier write* (two occurrences) in Section 4.5 of [78]. The corrected operational model includes all the behaviors of the original model, as well as the behaviors observed on the hardware and forbidden by the original model. The corrected operational model is believed to strictly and tightly over-approximate Power [6].

Atig et al. [14] showed that state reachability under memory models allowing write-to-read, read-to-read and read-to-write relaxations is undecidable. Although Power allows the above reorderings, the proof from [14] does not apply to Power directly. The reason is that the authors use a programming model with blocking reads: a read combines a load with a conditional that checks the loaded value. Power forbids reordering of stores with program-order-earlier conditionals and, consequently, with the loads on which this conditionals depend. The construction from Section 4.5 avoids these dependencies by implementing conditionals using local computations on registers, an idea suggested by Dr. Mohamed Faouzi Atig [16].

Existing tools for solving state reachability under (a fragment of) Power use the bounded model checking approach [8, 9]. Consequently, they can detect state reachability, but cannot generally prove that a program is safe.

4.1 Power Semantics

In this section we recall the corrected version of the model from [78].

The state of a running program consists of the runtime states of threads and the state of a storage subsystem. The runtime state of a thread includes information about the instructions being executed by the thread. In order to start executing an instruction, the thread must *fetch* it. The thread can fetch any instruction whose source control state is equal to the destination state of the last fetched instruction. Then, the thread must perform any computation required by the semantics of this instruction. For example, for a load the thread must compute the address being accessed, then read the value at this address, and place it into the target register. The last step of executing an instruction is *committing* it. Committing an instruction requires committing all its *dependencies*. For example, before committing a load the thread must commit all its *address dependencies* — the instructions which define the values of registers used in the address expression — and *control dependencies* — the program-order-earlier (fetched earlier than the load) conditional instructions. Moreover, all loads and stores accessing the same address must be committed in the order in which they were fetched.

The storage subsystem keeps track, for each address, of the global ordering of stores to this address — the *coherence order* — and the last store to this address *propagated* to each thread. When a thread commits a store, this store is assigned a position in the coherence order which we identify by a rational number — the *coherence key*. We choose rational numbers (rather than naturals) to be able to insert a store between any two stores in the coherence order. The key must be greater than the coherence key of the last store to the same address propagated

[1] `http://diy.inria.fr/cats/pldi-power/#lessvs`
[2] `http://diy.inria.fr/cats/cav-power/`

to this thread. The committed store is immediately propagated to its own thread. At some point later this store can be propagated to any other thread, as long as it is coherence-order-later (has a greater coherence key) than the last store to the same address propagated to that thread. When a thread loads a value from a certain address, it gets the value written by the last store to this address propagated to the thread. A thread can also forward the value being written by a not yet committed store to a later load reading the same address. This situation is called an *early read*.

An important property of Power is that it maintains the illusion of sequential consistency for single-threaded programs. This means that reorderings on the thread level must not lead to situations when, e.g., a program-order-later load reads a coherence-order-earlier store than the one read by a program-order-earlier load from the same address. In [78] these restrictions are enforced by the mechanism of restarting operations. We put these conditions into the requirements on final states of the running program instead.

Power provides several barrier instructions used for enforcing ordering of operations: `sync`, `lwsync`, `isync`. When a `sync` or `lwsync` instruction is committed, the *group A* set of stores is captured. It consists of all the stores that were propagated to the thread performing the barrier at the moment of barrier commit. Once all the group-A stores have been propagated to a thread, the `sync` or `lwsync` can be propagated to this thread. Once a `sync` is propagated to all threads, it is considered *acknowledged*.

Symmetrically, when a thread commits a store, the group-A set of `sync` and `lwsync` barriers is captured. It consists of all the barriers that were propagated to the thread committing the store at the moment of commit. A store can be propagated to a thread only after all group-A barriers have been propagated to this thread.

Committing a `sync` or `lwsync` requires all previous loads, stores, `sync`, `lwsync`, and `isync` instructions to be committed. Committing a load or a store requires all previous `sync`, `lwsync`, `isync` instructions to be committed and `sync`s to be acknowledged. Committing an `isync` requires all preceding loads and stores to have their addresses computed.

Finally, loading a value from memory or from an earlier store requires all previous `isync`s to be committed and `sync`s to be acknowledged.

Altogether, the set of commands for Power is

$$\begin{aligned}\langle cmd\rangle \;::=\; & \langle reg\rangle \leftarrow \texttt{mem[}\langle expr\rangle\texttt{]} \mid \texttt{mem[}\langle expr\rangle\texttt{]} \leftarrow \langle expr\rangle \\ \mid\; & \langle reg\rangle \leftarrow \langle expr\rangle \mid \texttt{assume(}\langle expr\rangle\texttt{)} \\ \mid\; & \texttt{sync} \mid \texttt{lwsync} \mid \texttt{isync}\end{aligned}$$

In this chapter we also assume that DOM is finite.

Formally, we define the semantics of program $\mathcal{P}$ on Power by a *Power automaton* $X_{\mathsf{power}}(\mathcal{P}) := (S_{\mathsf{power}}, \mathrm{E}, \Delta_{\mathsf{power}}, s_{\mathsf{power}_0}, F_{\mathsf{power}})$. We define the transitions labels (events) E together with the transitions.

State space

A state of the Power automaton is a pair $s_{\mathsf{power}} = (\mathsf{ts}, s_Z) \in S_{\mathsf{power}}$ with runtime thread states $\mathsf{ts}\colon \mathsf{TID} \to S_Y$ and storage subsystem state $s_Z \in S_Z$.

A runtime thread state $s_Y = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded}) \in S_Y$ includes a finite sequence of fetched instructions $\mathsf{fetched} \in \mathcal{I}^*$, a set of indices of committed

instructions $\mathsf{committed} \subseteq [1..|\mathsf{fetched}|]$, and a function $\mathsf{loaded}\colon [1..|\mathsf{fetched}|] \to \{\bot\} \cup \{\mathsf{init}_\mathsf{a} \mid \mathsf{a} \in \mathsf{ADDR}\} \cup \mathsf{TID} \times \mathbb{N}$ giving the store read by a load. We use init_a to denote the initial store of value 0 to address a. The initial state of a running thread is $s_{Y0} := (\varepsilon, \emptyset, \lambda i.\bot)$.

A state of the storage subsystem $s_Z = (\mathsf{co}, \mathsf{prop}, \mathsf{propsyncs}, \mathsf{groupastores}, \mathsf{groupasyncs}) \in S_Z$ includes

- $\mathsf{co}\colon \mathsf{TID} \times \mathbb{N} \cup \{\mathsf{init}_\mathsf{a} \mid \mathsf{a} \in \mathsf{ADDR}\} \to \mathbb{Q}$ — a mapping from a store instruction (its thread id and index in the list of fetched instructions) to its position in the coherence order,
- $\mathsf{prop}\colon \mathsf{TID} \times \mathsf{ADDR} \to \{\mathsf{init}_\mathsf{a} \mid \mathsf{a} \in \mathsf{ADDR}\} \cup \mathsf{TID} \times \mathbb{N}$ — a mapping from a thread id and an address to the last store to this address propagated to this thread,
- $\mathsf{propsyncs}\colon \mathsf{TID} \to 2^{\mathsf{TID} \times \mathbb{N}}$ — a mapping from a thread id to the set of `sync`s and `lwsync`s propagated to this thread,
- $\mathsf{groupastores}\colon \mathsf{TID} \times \mathbb{N} \to \{\bot\} \cup (\mathsf{ADDR} \to \{\mathsf{init}_\mathsf{a} \mid \mathsf{a} \in \mathsf{ADDR}\} \cup \mathsf{TID} \times \mathbb{N})$ — a mapping from a `sync` or `lwsync` to its group-A stores,
- $\mathsf{groupasyncs}\colon \mathsf{TID} \times \mathbb{N} \to 2^{\mathsf{TID} \times \mathbb{N}}$ — a mapping from a store to its group-A `sync`s and `lwsync`s.

The initial state of the storage subsystem is $s_{Z0} := (\lambda\mathsf{tid}.\lambda i.0, \lambda\mathsf{tid}.\lambda\mathsf{a}.\mathsf{init}_\mathsf{a}, \lambda\mathsf{tid}.\emptyset, \lambda\mathsf{tid}.\lambda i.\emptyset, \lambda\mathsf{tid}.\lambda i.\emptyset)$.

The initial state of automaton $X_{\mathsf{power}}(\mathcal{P})$ is $s_{\mathsf{power}_0} := (\lambda\mathsf{tid}.s_{Y0}, s_{Z0})$.

Transition relation

Fix a state $s_{\mathsf{power}} = (\mathsf{ts}, s_Z)$ with $s_Z = (\mathsf{co}, \mathsf{prop}, \mathsf{propsyncs}, \mathsf{groupastores}, \mathsf{groupasyncs})$ and a thread id $\mathsf{tid} \in \mathsf{TID}$ with runtime state $\mathsf{ts}(\mathsf{tid}) = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$.

Let $\mathsf{eval}(\mathsf{tid}, i, e)$ return the value in DOM of the expression e in the i'th fetched instruction of thread tid, or $\bot$ when the value is undefined. Formally $\mathsf{eval}(\mathsf{tid}, i, e) := \mathsf{v}$, where v is computed as follows. If $e \in \mathsf{DOM}$, then $\mathsf{v} := e$. If $e = \mathsf{f}(e_1 \dots e_n)$, then $\mathsf{v} := \mathsf{f}(\mathsf{eval}(\mathsf{tid}, i, e_1) \dots \mathsf{eval}(\mathsf{tid}, i, e_n))$. Otherwise, $e = r \in \mathsf{REG}$. Let $i' \in [1..i-1]$ be the greatest index, such that $\mathsf{fetched}[i']$ is a local assignment or a load to r. If there is no such index, we define $\mathsf{v} := 0$. If $\mathsf{lab}(\mathsf{fetched}[i']) = r \leftarrow e_\mathsf{v}$, then $\mathsf{v} := \mathsf{eval}(\mathsf{tid}, i', e_\mathsf{v})$. If $\mathsf{lab}(\mathsf{fetched}[i']) = r \leftarrow \mathtt{mem}[e_\mathsf{a}]$, then $\mathsf{v} := \bot$ if $\mathsf{loaded}[i'] = \bot$, $\mathsf{v} := 0$ if $\mathsf{loaded}[i'] = \mathsf{init}_*$, and $\mathsf{v} := \mathsf{val}(\mathsf{loaded}[i'])$ otherwise (see the definition of val below).

The expression $\mathsf{addr}(\mathsf{tid}, i)$ returns the value of the address argument of the i'th fetched instruction of thread tid and is defined as follows. We use the special value $\top$ if the instruction has no such argument. If $\mathsf{lab}(\mathsf{fetched}[i]) = r \leftarrow \mathtt{mem}[e_\mathsf{a}]$ or $\mathsf{lab}(\mathsf{fetched}[i]) = \mathtt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}$, then $\mathsf{addr}(\mathsf{tid}, i) := \mathsf{eval}(\mathsf{tid}, i, e_\mathsf{a})$. Otherwise, $\mathsf{addr}(\mathsf{tid}, i) := \top$. We overload $\mathsf{addr}(\mathsf{init}_\mathsf{a}) := \mathsf{a}$.

Similarly, the expression $\mathsf{val}(\mathsf{tid}, i)$ returns the value of the value argument of the i'th fetched instruction of thread tid and is defined as follows. If $\mathsf{lab}(\mathsf{fetched}[i]) = \mathtt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}$, $\mathsf{lab}(\mathsf{fetched}[i]) = r \leftarrow e_\mathsf{v}$, or $\mathsf{lab}(\mathsf{fetched}[i]) = \mathtt{assume}(e_\mathsf{v})$, then $\mathsf{val}(\mathsf{tid}, i) = \mathsf{eval}(\mathsf{tid}, i, e_\mathsf{v})$. Otherwise, $\mathsf{val}(\mathsf{tid}, i) := \top$.

The expressions $\mathsf{addrdep}(\mathsf{tid}, i)$, $\mathsf{datadep}(\mathsf{tid}, i)$, $\mathsf{ctrldep}(\mathsf{tid}, i)$ denote the sets of indices of instructions in thread tid being respectively address, data, and control dependencies of the i'th instruction. The first two can be formally defined in a recursive manner, similar to eval. Also, $\mathsf{ctrldep}(\mathsf{tid}, i) := \{i' \in [1..i-1] \mid \mathsf{lab}(\mathsf{fetched}[i']) = \texttt{assume}(e_\mathsf{v})\}$.

The expression $\mathsf{acked}(\mathsf{tid}, i)$ returns $\top$ if $(\mathsf{tid}, i) \in \mathsf{propsyncs}(\mathsf{tid}')$ for all $\mathsf{tid}' \in \mathsf{TID}$. It returns $\bot$ otherwise.

Let $\mathcal{T}_\mathsf{tid} = (Q_\mathsf{tid}, \mathsf{CMD}, \mathcal{I}_\mathsf{tid}, q_{\mathsf{tid}0}, Q_\mathsf{tid}) \in \mathcal{P}$. The transition relation Δ_power is the smallest relation defined by the rules below:

POW-FETCH Consider $\mathsf{instr} \in \mathcal{I}_\mathsf{tid}$ with $\mathsf{src}(\mathsf{instr}) = \mathsf{dst}(\mathsf{last}(\mathsf{fetched}))$ or $\mathsf{src}(\mathsf{instr}) = q_{\mathsf{tid}0}$ if $\mathsf{fetched} = \varepsilon$, then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{fetch},\mathsf{tid},\mathsf{instr})} (\mathsf{ts}[\mathsf{tid} := (\mathsf{fetched} \cdot \mathsf{instr}, \mathsf{committed}, \mathsf{loaded})], s_Z).$$

POW-LOAD Let $\mathsf{fetched}[i]$ be a load, $\mathsf{loaded}[i] = \bot$, $\mathsf{a} := \mathsf{addr}(\mathsf{tid}, i) \neq \bot$. Assume that for all $i' \in [1..i-1]$ holds: if $\mathsf{fetched}[i']$ is a $\texttt{sync}$ or $\texttt{isync}$, then $i' \in \mathsf{committed}$ and, if $\mathsf{fetched}[i']$ is a $\texttt{sync}$, $\mathsf{acked}(\mathsf{tid}, i') = \top$. Assume that if $i' \in [1..i-1]$ is $\texttt{lwsync}$, then for $i'' \in [1..i'-1]$ holds: if $\mathsf{fetched}[i'']$ is a load, then $i'' \in \mathsf{committed}$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{load},\mathsf{tid},i,\mathsf{a})} (\mathsf{ts}[\mathsf{tid} := (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded}[i := \mathsf{prop}(\mathsf{tid}, \mathsf{a})])], s_Z).$$

POW-EARLY Let $\mathsf{fetched}[i]$ be a load, $\mathsf{loaded}[i] = \bot$, and $\mathsf{a} := \mathsf{addr}(\mathsf{tid}, i) \neq \bot$. Let $i' \in [1..i-1]$ be the greatest index such that $\mathsf{fetched}[i']$ is a store with $\mathsf{a}' = \mathsf{addr}(\mathsf{tid}, i') \in \{\mathsf{a}, \bot\}$. Assume $\mathsf{a}' \neq \bot$, $\mathsf{val}(\mathsf{tid}, i') \neq \bot$, $i' \notin \mathsf{committed}$. Assume that for all $i' \in [1..i-1]$ holds: if $\mathsf{fetched}[i']$ is a $\texttt{sync}$ or $\texttt{isync}$, then $i' \in \mathsf{committed}$ and, if $\mathsf{fetched}[i']$ is a $\texttt{sync}$, $\mathsf{acked}(\mathsf{tid}, i') = \top$. Assume that if $i' \in [1..i-1]$ is $\texttt{lwsync}$, then for $i'' \in [1..i'-1]$ holds: if $\mathsf{fetched}[i'']$ is a load, then $i'' \in \mathsf{committed}$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{load},\mathsf{tid},i,\mathsf{a})} (\mathsf{ts}[\mathsf{tid} := (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded}[i := (\mathsf{tid}, i')])], s_Z).$$

POW-COMMIT Consider $i \in [1..|\mathsf{fetched}|] \setminus \mathsf{committed}$, where $\mathsf{fetched}[i]$ is not a store. Assume $\mathsf{addrdep}(\mathsf{tid}, i) \cup \mathsf{datadep}(\mathsf{tid}, i) \cup \mathsf{ctrldep}(\mathsf{tid}, i) \subseteq \mathsf{committed}$. Assume $\mathsf{a} := \mathsf{addr}(\mathsf{tid}, i) \neq \bot$ and $\mathsf{v} := \mathsf{val}(\mathsf{tid}, i) \neq \bot$. If $\mathsf{a} \neq \top$, assume $\{i' \in [1..i-1] \mid \mathsf{addr}(\mathsf{tid}, i') \in \{\mathsf{a}, \bot\}\} \subseteq \mathsf{committed}$. In case $\mathsf{fetched}[i]$ is a load, assume $\mathsf{loaded}[i] \neq \bot$. In case $\mathsf{fetched}[i]$ is an $\texttt{assume}()$, assume $\mathsf{v} \neq 0$. In case $\mathsf{fetched}[i]$ is a load, a store, $\texttt{sync}$, $\texttt{lwsync}$, or $\texttt{isync}$, assume for each $i' \in [1..i-1]$ with $\mathsf{lab}(\mathsf{fetched}[i']) \in \{\texttt{sync}, \texttt{lwsync}, \texttt{isync}\}$ holds $i' \in \mathsf{committed}$ and, if $\mathsf{fetched}[i']$ is a $\texttt{sync}$, $\mathsf{acked}(\mathsf{tid}, i') = \top$. In case $\mathsf{fetched}[i]$ is $\texttt{sync}$ or $\texttt{lwsync}$, assume for each $i' \in [1..i-1]$ with $\mathsf{addr}(\mathsf{tid}, i') \neq \top$ holds $i' \in \mathsf{committed}$. In case $\mathsf{fetched}[i]$ is $\texttt{isync}$, assume for each $i' \in [1..i-1]$ holds $\mathsf{addr}(\mathsf{tid}, i') \neq \bot$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{commit},\mathsf{tid},i)} (\mathsf{ts}[\mathsf{tid} := (\mathsf{fetched}, \mathsf{committed} \cup \{i\}, \mathsf{loaded})], s'_Z).$$

If $\mathsf{fetched}[i]$ is not $\texttt{sync}$ or $\texttt{lwsync}$, then $s'_Z := s_Z$. Otherwise, $s'_Z := (\mathsf{co}, \mathsf{prop}, \mathsf{propsyncs}, \mathsf{groupastores}', \mathsf{groupasyncs})$, where $\mathsf{groupastores}' :=$

$\mathsf{groupastores}[(\mathsf{tid}, i) := \mathsf{prop}(\mathsf{tid})]$; moreover, the transition is immediately followed by POW-PROP-SYNC transition propagating the barrier to thread tid.

POW-STORE Assume all the preconditions from the previous rule hold, but $\mathsf{fetched}[i]$ is a store. Choose a coherence key $\mathsf{k} \in \mathbb{Q}$ such that there is no $\mathsf{tid}' \in \mathsf{TID}$, $i' \in \mathbb{N}$ for which $\mathsf{co}(\mathsf{tid}', i') = \mathsf{k}$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{commit},\mathsf{tid},i,\mathsf{k},\mathsf{a})} (\mathsf{ts}[\mathsf{tid} := (\mathsf{fetched}, \mathsf{committed} \cup \{i\}, \mathsf{loaded})], s'_Z),$$

where $s'_Z := (\mathsf{co}', \mathsf{prop}, \mathsf{propsyncs}, \mathsf{groupastores}, \mathsf{groupasyncs}')$, $\mathsf{co}' := \mathsf{co}[(\mathsf{tid}, i) := \mathsf{k}]$, $\mathsf{groupasyncs}' := \mathsf{groupasyncs}[(\mathsf{tid}, i) := \mathsf{propsyncs}(\mathsf{tid})]$.

Moreover, this transition is immediately followed by a POW-PROP-STORE transition propagating the store to the thread where it was committed.

POW-PROP-STORE Consider $\mathsf{tid}' \in \mathsf{TID}$, $i' \in \mathbb{N}$ with $\mathsf{co}(\mathsf{tid}', i') \neq \bot$. Let $\mathsf{a} := \mathsf{addr}(\mathsf{tid}', i')$. Assume $\mathsf{co}(\mathsf{prop}(\mathsf{tid}, \mathsf{a})) < \mathsf{co}(\mathsf{tid}', i')$. Assume $s \in \mathsf{propsyncs}(\mathsf{tid})$ for all $s \in \mathsf{groupasyncs}(\mathsf{tid}', i')$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{prop},\mathsf{tid},\mathsf{tid}',i',\mathsf{a})} (\mathsf{ts}, s'_Z),$$

where $s'_Z := (\mathsf{co}, \mathsf{prop}[(\mathsf{tid}, \mathsf{a}) := (\mathsf{tid}', i')], \mathsf{propsyncs}, \mathsf{groupastores}, \mathsf{groupasyncs})$.

POW-PROP-SYNC Consider $i \in \mathsf{committed}$, where $\mathsf{fetched}[i]$ is a `sync` or `lwsync`. Fix $\mathsf{tid}' \in \mathsf{TID}$. Assume $(\mathsf{tid}, i) \notin \mathsf{propsyncs}(\mathsf{tid}')$. Assume $\mathsf{co}(\mathsf{groupastores}(\mathsf{tid}, i, \mathsf{a})) \leq \mathsf{co}(\mathsf{prop}(\mathsf{tid}', \mathsf{a}))$ for each $\mathsf{a} \in \mathsf{ADDR}$. Then:

$$(\mathsf{ts}, s_Z) \xrightarrow{(\mathsf{prop},\mathsf{tid},i,\mathsf{tid}')} (\mathsf{ts}, (\mathsf{co}, \mathsf{prop}, \mathsf{propsyncs}', \mathsf{groupastores}, \mathsf{groupasyncs}),$$

where $\mathsf{propsyncs}' := \mathsf{propsyncs}[\mathsf{tid}' := \mathsf{propsyncs}(\mathsf{tid}') \cup \{(\mathsf{tid}, i)\})]$.

Final states

The set of final states $F_{\mathsf{power}} \subseteq S_{\mathsf{power}}$ consists of all states $s_{\mathsf{power}} = (\mathsf{ts}, (\mathsf{co}, \mathsf{prop})) \in S_{\mathsf{power}}$, such that for each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{ts}[\mathsf{tid}] = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$ the following holds:

POW-FIN-COMM All instructions are committed: $\mathsf{committed} = [1..|\mathsf{fetched}|]$.

POW-FIN-LD Loads agree with the coherence order. Let $\mathsf{fetched}[i]$ be a load, and $\mathsf{fetched}[i']$ be an earlier load to the same address: $i' < i$, $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{addr}(\mathsf{tid}, i')$. Then $\mathsf{co}(\mathsf{loaded}[i']) \leq \mathsf{co}(\mathsf{loaded}[i])$.

POW-FIN-LD-ST Loads and stores in the same thread agree with the coherence order. Let $\mathsf{fetched}[i]$ be a load, let $\mathsf{fetched}[i']$ be an earlier store to the same address: $i' < i$, $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{addr}(\mathsf{tid}, i')$. Then $\mathsf{co}(\mathsf{tid}, i') \leq \mathsf{co}(\mathsf{loaded}[i])$.

The set of all *Power computations of program* $\mathcal{P}$ is $\mathsf{C}_{\mathsf{power}}(\mathcal{P}) := \mathcal{L}(X_{\mathsf{power}}(\mathcal{P}))$.

Example 4.1. $\sigma_{MP} = \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a,1) \cdot \mathsf{fetch}(b) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b,1) \cdot \mathsf{prop}(b,2) \cdot \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c)$ is a feasible Power computation of the program MP (Figure 1.2):

- $\mathsf{fetch}(a) := (\mathsf{fetch}, 1, a)$ — thread 1 fetches store instruction a.
- $\mathsf{commit}(a) := (\mathsf{commit}, 1, 1, 1, x)$ — thread 1 commits a with $\mathsf{k} = 1$.
- $\mathsf{prop}(a,1) := (\mathsf{prop}, 1, 1, 1, x)$ — a is propagated to its own thread.
- $\mathsf{fetch}(b) := (\mathsf{fetch}, 1, b)$ — thread 1 fetches store instruction b.
- $\mathsf{commit}(b) := (\mathsf{commit}, 1, 2, 2, y)$ — thread 1 commits b with $\mathsf{k} = 2$.
- $\mathsf{prop}(b,1) := (\mathsf{prop}, 1, 1, 2, x)$ — the store is propagated to its thread.
- $\mathsf{prop}(b,2) := (\mathsf{prop}, 2, 1, 2, x)$ — the store is propagated to thread 2.
- $\mathsf{fetch}(c) := (\mathsf{fetch}, 2, c)$ — thread 2 fetches load c.
- $\mathsf{fetch}(d) := (\mathsf{fetch}, 2, c)$ — thread 2 fetches load d.
- $\mathsf{load}(c) := (\mathsf{load}, 2, 1, y)$ — thread 2 reads value 1 written by b to y, because b was propagated to thread 2.
- $\mathsf{load}(d) := (\mathsf{load}, 2, 2, x)$ — thread 2 reads the initial value 0 of x, because a was not propagated to thread 2.
- $\mathsf{commit}(d) := (\mathsf{commit}, 2, 2)$ — thread 2 commits load d.
- $\mathsf{commit}(c) := (\mathsf{commit}, 2, 1)$ — thread 2 commits load c.

In the end, POW-FIN-COMM holds as all fetched instructions are indeed committed, and POW-FIN-LD and POW-FIN-LD-ST trivially hold, as none of the threads has two instructions accessing the same address.

Lemma 4.2. *Assume* $s_{power_0} \xrightarrow{\sigma} s_{power} \in F_{power}$. *Then* s_{power} *is uniquely determined.*

Proof. Given a state and an event e, there is at most one transition from this state labeled by e that may lead to a final state. This is clear for non-load events. For load events, this follows from Lemma 4.5 and Lemma 4.6: if a load event was produced by a load from memory transition, then condition (3) from Lemma 4.6 holds, then condition (1) from Lemma 4.5 cannot hold for any store, therefore, the load event cannot be produced by an early read transition. □

Lemma 4.3. *Let* $s_{power_0} \xrightarrow{\sigma} (ts, s_Z) \xrightarrow{e} (ts', s'_Z)$. *Let* $(fetched, committed, loaded) = ts(tid)$, $(fetched', committed', loaded') = ts'(tid)$ *for some* $tid \in TID$. *If* $loaded[i] \neq \bot$, *then* $loaded'[i] = loaded[i]$.

Proof. Follows from the $\mathsf{loaded}[i] = \bot$ requirement in POW-LOAD and POW-EARLY transitions. □

Lemma 4.4. *Let* $s_{power_0} \xrightarrow{\sigma} s_{power} \xrightarrow{e} s_{power}'$. *Assume* $eval(tid, i, e) = v \neq \bot$ *in* s_{power}. *Then* $eval(tid, i, e) = v$ *in* s_{power}'.

Proof. By definition of eval, Lemma 4.3, and the fact that functions in FUN are deterministic. □

Lemma 4.5. *Consider a computation $\sigma \in \mathsf{C}_{power}(\mathcal{P})$. Then a load (tid, i) reads a value from a store (tid, i') via an early read (POW-EARLY) transition iff (1) $\sigma = \sigma_1 \cdot (\mathsf{load}, \mathsf{tid}, i, \mathsf{a}) \cdot \sigma_2 \cdot (\mathsf{commit}, \mathsf{tid}, i', *, \mathsf{a}) \cdot \sigma_3$, $i' \in [1..i-1]$ and (2) σ_3 does not contain events matching $(\mathsf{commit}, \mathsf{tid}, [i'+1..i-1], *, \mathsf{a})$.*

Proof. From left to right. Assume the load (tid, i) reads from the store (tid, i') via an early read transition. Then (tid, i) must be the latest store to the same address in thread tid and must not be committed before load (i.e. committed after it), therefore (1) holds. If (2) does not hold, then (tid, i') is not the latest store to address a in thread tid before the load event, since stores to the same address are committed in the order of fetching. Contradiction.

From right to left. Let $s_{\mathsf{power}_0} \xrightarrow{\sigma_1} s_{\mathsf{power}} = (\mathsf{ts}, s_Z)$. Consider $\mathsf{ts}(\mathsf{tid}) = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$. Let $i'' < i$ be the greatest index, such that $\mathsf{fetched}[i'']$ is a store, $\mathsf{addr}(i'') \in \{\mathsf{a}, \bot\}$.

Assume $i' < i''$. If $\mathsf{addr}(i'') = \mathsf{a}$, we get a contradiction to (2), since stores to the same address are committed in the order of fetching. If $\mathsf{addr}(i'') = \bot$, then an early read is not possible in state s_{power}, and the load reads from the latest propagated store (POW-LOAD), which is coherence-order-before the store (tid, i'), which is program-order-before (tid, i). This situation is forbidden by POW-FIN-LD-ST.

By Lemma 4.4, $\mathsf{addr}(\mathsf{tid}, i') \in \{\mathsf{a}, \bot\}$, therefore, $i'' = i'$. Assume $\mathsf{addr}(\mathsf{tid}, i') = \bot$ or $\mathsf{val}(\mathsf{tid}, i') = \bot$. Then, again, a load from the latest propagated store takes place, which is impossible (see above). Therefore, $\mathsf{addr}(\mathsf{tid}, i') = \mathsf{a}$ and $\mathsf{val}(\mathsf{tid}, i') \neq \bot$.

Obviously, $i' \notin \mathsf{committed}$ holds, as each fetched instruction is committed only once, and (tid, i') is committed after the load takes place, see (1). All in all, all requirements specific for the early read from (tid, i') are met, therefore, an early read transition from state s_{power} is possible. As shown above, a load from memory transition from the same state leads to $\sigma \notin \mathsf{C}_{\mathsf{power}}(\mathcal{P})$, therefore, (tid, i) reads from the store (tid, i') via an early read transition. □

Lemma 4.6. *Consider a computation $\sigma \in \mathsf{C}_{power}(\mathcal{P})$. Then a load (tid, i) reads a value from a store (tid', i') via a load from memory (POW-LOAD) transition iff (1) $\sigma = \sigma_1 \cdot (\mathsf{prop}, \mathsf{tid}, \mathsf{tid}', i', \mathsf{a}) \cdot \sigma_2 \cdot (\mathsf{load}, \mathsf{tid}, i, \mathsf{a}) \cdot \sigma_3$, (2) σ_2 does not contain events matching $(\mathsf{prop}, \mathsf{tid}, *, *, \mathsf{a})$, and (3) σ_3 does not contains events matching $(\mathsf{commit}, \mathsf{tid}, [1..i-1], *, \mathsf{a})$.*

Proof. From left to right. Assume the load (tid, i) reads from the store (tid', i') via a load from memory transition. Then, the load has read from the latest store to address a propagated to thread tid, i.e., (1) and (2) hold. Assume (3) does not hold — σ_3 contains a commit $(\mathsf{commit}, \mathsf{tid}, i'', *, \mathsf{a})$ and $i'' < i$. Then, (tid, i) reads from the store (tid', i'), which is coherence-order-before the store (tid, i''), which is program-order-before (tid, i). This situation is forbidden by POW-FIN-LD-ST.

From right to left. By (1), (3), and Lemma 4.5, the load event was not generated by an early read transition. Therefore, the event was generated by a load from memory transition, and the load has taken the value from the latest propagated store to address a, which is, by (1) and (2), (tid', i'). □

4.2 Traces and Robustness

The trace of a computation $\sigma \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$ is a directed graph $T(\sigma) := (V, \rightarrow_{po}, \rightarrow_{co}, \rightarrow_{src}, \rightarrow_{cf})$ with nodes V and four kinds of arcs. The nodes are instructions together with their thread identifiers and serial numbers: $V \subseteq \{\mathsf{init}_{\mathsf{a}} \mid \mathsf{a} \in \mathsf{ADDR}\} \cup \bigcup_{\mathsf{tid} \in \mathsf{TID}} \{\mathsf{tid}\} \times \mathbb{N} \times \mathcal{I}_{\mathsf{tid}}$. The *program order* $\rightarrow_{po}$ is the order in which instructions were fetched in each thread. The *coherence order* $\rightarrow_{co}$ gives the global ordering of stores to each address. The *source order* $\rightarrow_{src}$ shows the store from which a load took its value. The *conflict order* $\rightarrow_{cf}$ shows, for a load, the stores following in the coherence order the store from which the load took its value. We define the *happens-before* relation as $\rightarrow_{hb} := \rightarrow_{po} \cup \rightarrow_{co} \cup \rightarrow_{src} \cup \rightarrow_{cf}$.

Formally, consider a computation $\sigma \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$. Let $s_{\mathsf{power}_0} \xrightarrow{\sigma} s_{\mathsf{power}}$ with $s_{\mathsf{power}} = (\mathsf{ts}, (\mathsf{co}, \mathsf{prop}, \mathsf{propsyncs}, \mathsf{groupastores}, \mathsf{groupasyncs}))$. By Lemma 4.2, s_{power} is uniquely determined. The trace $T(\sigma) := (V, \rightarrow_{po}, \rightarrow_{co}, \rightarrow_{src}, \rightarrow_{cf})$ is defined as follows. Assuming $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{ts}(\mathsf{tid}) = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$, $i \in [1..|\mathsf{fetched}|]$, and similarly for tid', we have:

$$
\begin{aligned}
V :=& \{(\mathsf{tid}, i, \mathsf{fetched}[i]) \mid \mathsf{tid} \in \mathsf{TID},\ i \in \mathbb{N}\},\\
\rightarrow_{po} :=& \{((\mathsf{tid}, i, \mathsf{fetched}[i]), (\mathsf{tid}, i+1, \mathsf{fetched}[i+1])) \mid\\
&\qquad i \in [1..|\mathsf{fetched}| - 1]\},\\
\rightarrow_{co} :=& \{((\mathsf{tid}, i, \mathsf{fetched}[i]), (\mathsf{tid}', i', \mathsf{fetched}[i'])) \mid\\
&\qquad \mathsf{addr}(\mathsf{tid}, i) = \mathsf{addr}(\mathsf{tid}', i') \text{ and } \mathsf{co}(\mathsf{tid}, i) < \mathsf{co}(\mathsf{tid}', i')\} \cup\\
&\quad \{(\mathsf{init}_a, (\mathsf{tid}', i', \mathsf{fetched}[i'])) \mid a = \mathsf{addr}(\mathsf{tid}', i')\},\\
\rightarrow_{src} :=& \{((\mathsf{tid}, i, \mathsf{fetched}[i]), (\mathsf{tid}', i', \mathsf{fetched}'[i'])) \mid\\
&\qquad (\mathsf{tid}, i) = \mathsf{loaded}'[i']\} \cup\\
&\quad \{(\mathsf{init}_a, (\mathsf{tid}', i', \mathsf{fetched}'[i'])) \mid \mathsf{init}_a = \mathsf{loaded}'(i')\},\\
\rightarrow_{cf} :=& \{(a, b) \mid \exists c\colon c \rightarrow_{src} a \text{ and } c \rightarrow_{co} b\}.
\end{aligned}
$$

We will also need address $\rightarrow_{addr}$ and data $\rightarrow_{data}$ dependence relations (defined as expected based on $\mathsf{addrdep}$ and $\mathsf{datadep}$).

$$
\begin{aligned}
\rightarrow_{addr} :=& \{((\mathsf{tid}, i, \mathsf{fetched}[i]), (\mathsf{tid}, i', \mathsf{fetched}'[i])) \mid i \in \mathsf{addrdep}(\mathsf{tid}, i')\},\\
\rightarrow_{data} :=& \{((\mathsf{tid}, i, \mathsf{fetched}[i]), (\mathsf{tid}, i', \mathsf{fetched}'[i])) \mid i \in \mathsf{datadep}(\mathsf{tid}, i')\}.
\end{aligned}
$$

Since $\rightarrow_{po}$ includes all the information from the $\mathsf{fetched}$ component of a thread state, $\rightarrow_{addr}$ and $\rightarrow_{data}$ can be reconstructed from $\rightarrow_{po}$ by inspecting the instructions labeling the nodes. They are therefore not included in the trace explicitly.

We instantiate the robustness problem (Section 2.8) for Power.

Problem 4.7 (Robustness against Power)**.** Given a program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}) = T_{\mathsf{power}}(\mathcal{P})$.

According to Lemma 2.16, checking robustness of a program amounts to checking whether the program has computations with cyclic happens-before relation.

Example 4.8. The trace of computation σ_{MP} (Figure 4.1) has a cyclic happens-before relation. By Lemma 2.16, this means that the program is not robust. Indeed, in no SC computation load d can read 0 whereas c has read 1.

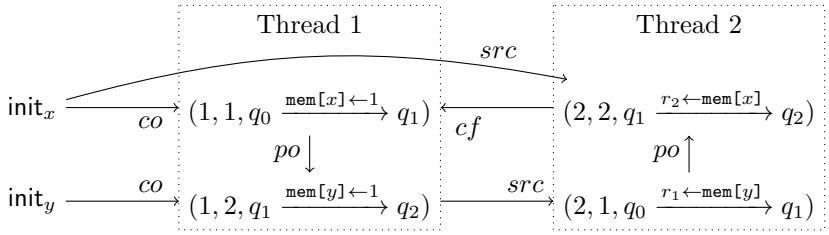

Figure 4.1: Trace of computation σ_{MP} from Example 4.1.

4.3 Normal-Form Computations

We say that a computation $\tau \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$ is *in normal form of degree n* if there is a partitioning $\tau = \tau_1 \cdots \tau_n$, such that all fetch events are in τ_1 (POW-NF-A) and events belonging to different instructions occur in different parts of the computation in the same order (POW-NF-B):

POW-NF-A $(\tau_2 \cdots \tau_n)\downarrow \mathsf{fetch} = \varepsilon$.

POW-NF-B Formally, for $j \in \{1, 2\}$ let e_j, e'_j be events belonging to instruction (tid_j, i_j). If $\mathsf{e}_1, \mathsf{e}_2 \in \tau_s$ and $\mathsf{e}'_1, \mathsf{e}'_2 \in \tau_{s'}$, then $\mathsf{e}_1 <_{\tau_s} \mathsf{e}_2$ iff $\mathsf{e}'_1 <_{\tau_{s'}} \mathsf{e}'_2$.

In the rest of this section we prove the following theorem.

Theorem 4.9. *A program is robust iff it has no normal-form computation of degree $|\mathcal{P}| + 3$ with cyclic happens-before relation.*

In order to keep the proofs in this and the next section readable and understandable, we omit the details related to Power barrier instructions in them. Consequently, the propsyncs, groupastores, and groupasyncs parts of s_Z component of the state are ignored. We come back to these instructions and show how to support them in Section 4.4.3.

Consider a computation $\sigma \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$. By $\sigma \setminus (\mathsf{tid}, i)$ we denote the computation obtained from σ by deleting all events belonging to the i'th fetched instruction in thread tid.

Lemma 4.10 (Cancellation)**.** *Consider a non-empty computation $\sigma \in \mathsf{C}_{power}(\mathcal{P})$. Then there is a (tid_x, i_x), such that $\sigma' = \sigma \setminus (\mathsf{tid}_x, i_x)$ satisfies $|\sigma'| < |\sigma|$ and $\sigma' \in \mathsf{C}_{power}(\mathcal{P})$.*

Proof. Consider the last fetched instruction in each thread. If among such instructions there is a non-store instruction, delete it: its result cannot be used by any other instruction. If all these instructions are stores, delete the one, on which (1) no load or store depends via $(\rightarrow_{src} \cup \rightarrow_{data})^+ \cdot \rightarrow_{addr}$, and (2) no condition depends via $(\rightarrow_{src} \cup \rightarrow_{data})^+$.

Towards a contradiction, assume there is no such store. Consider the last fetched (store) instruction in a thread tid_1: (tid_1, i_1). Case 1: there is a load or a store (tid_2, i'_2) whose address depends on (tid_1, i_1). Case 2: there is a condition (tid_2, i'_2) whose value depends on (tid_1, i_1). Consider the last fetched instruction in thread tid_2: (tid_2, i_2). It must be a store, and it must have been committed after (tid_1, i_1): a store can only be committed after all loads and

stores fetched before it have their addresses determined (Case 1) and after all preceding conditions are committed (Case 2).

Continuing the reasoning, for any last fetched instruction in a thread (tid_j, i_j) there is a last instruction in a different thread $(\mathsf{tid}_{j+1}, i_{j+1})$ which must have been committed later. Taking into account finiteness of the number of threads, we get a contradiction. □

Fix a program $\mathcal{P}$. Consider a shortest Power computation $\alpha \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$ with cyclic $\rightarrow_{hb}$. Let $(\mathsf{tid}_{\mathsf{x}}, i_{\mathsf{x}})$ be the instruction determined by Lemma 4.10. Let $\alpha := \alpha_1 \cdot \mathsf{x}_1 \cdot \alpha_2 \cdot \mathsf{x}_2 \cdots \alpha_n$, where $\{\mathsf{x}_1 \ldots \mathsf{x}_{n-1}\}$ are the events belonging to the i_{x}'th instruction fetched in thread $\mathsf{tid}_{\mathsf{x}}$. Then $\alpha \setminus (\mathsf{tid}_{\mathsf{x}}, i_{\mathsf{x}}) := \alpha' := \alpha_1 \cdot \alpha_2 \cdots \alpha_n$. Since α' is shorter than α, its $\rightarrow_{hb}$ is acyclic. Therefore, there is a computation $\beta \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$ with $T(\beta) = T(\alpha')$.

Computations β and α' consist of the same fetch, load, and commit events: fetch events are determined by $\rightarrow_{po}$; address component a of load and store commit events is determined by $\rightarrow_{addr}$, $\rightarrow_{data}$ (derivable from $\rightarrow_{po}$), and $\rightarrow_{src}$; since $\rightarrow_{co}$ is the same for both computations, we can assume that matching store commit events have the same value of coherence key k. Notably, β can have more propagate events than α' as Power semantics does not guarantee that all stores are propagated to all threads. Now we reorder events in each part α_j of α in the way they follow in β. This gives a computation $\gamma := \beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdot \beta{\downarrow}\alpha_2 \cdot \mathsf{x}_2 \cdots \beta{\downarrow}\alpha_n$. In the rest of the section we show that γ is a valid Power computation of program $\mathcal{P}$ and has the same trace as α.

Lemma 4.11. *For all $\mathit{tid} \in \mathit{TID}$ it holds that $\alpha{\downarrow}\mathit{fetch}{\downarrow}\mathit{tid} = \gamma{\downarrow}\mathit{fetch}{\downarrow}\mathit{tid}$.*

Proof. Since $T(\beta) = T(\alpha')$, by definition of α and properties of projection, for any $\mathsf{tid} \in \mathsf{TID}$ we have

$$\begin{aligned}
\alpha{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} &= \alpha_1{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \cdot \mathsf{x}_1{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \cdots \alpha_n{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \\
&= \cdots (\beta{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid}){\downarrow}(\alpha_i{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid}) \cdot \mathsf{x}_i{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \cdots \\
&= \beta{\downarrow}\alpha_1{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \cdot \mathsf{x}_1{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \cdots \beta{\downarrow}\alpha_n{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \\
&= (\beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdots \beta{\downarrow}\alpha_n){\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid} \\
&= \gamma{\downarrow}\mathsf{fetch}{\downarrow}\mathsf{tid}.
\end{aligned}$$

□

Lemma 4.12. *Consider some (tid, i) and (tid', i'). Let $P(\sigma) := \mathit{true}$ if requirements (1)–(2) from Lemma 4.5 or (1)–(3) from Lemma 4.6 hold for σ, and $P(\sigma) := \mathit{false}$ otherwise. Then, if $P(\alpha)$ then $P(\gamma)$.*

Proof. The proof is a case consideration: which of the two conditions (Lemma 4.5 requirements or Lemma 4.6 requirements) hold for σ, which of them hold for α, and whether the distinguished load and commit events are located in the same part α_j. We consider two of the eight cases. The other are similar.

Assume requirements (1)–(2) from Lemma 4.5 hold for α and requirements (1)–(3) from Lemma 4.6 hold for sequentially consistent computation β. If load and commit events are in the same part, then $\alpha = \alpha_1 \cdot \mathsf{x}_1 \cdots (\alpha'_j \cdot b \cdot \alpha''_j \cdot c \cdot d \cdot \alpha'''_j) \cdot \mathsf{x}_j \cdots \alpha_n$, $\beta = \beta_1 \cdot c \cdot d \cdot \beta_2 \cdot b \cdot \beta_3$, where $b = (\mathsf{load}, \mathsf{tid}, i, \mathsf{a})$, $c = (\mathsf{commit}, \mathsf{tid}, i')$, $d = (\mathsf{prop}, \mathsf{tid}, \mathsf{tid}, i', \mathsf{a})$, $i' < i$. Consequently, $\gamma = \beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdots \beta{\downarrow}\alpha_j \cdot \mathsf{x}_j \cdots \beta{\downarrow}$

$\alpha_n = \beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdots (\beta_1{\downarrow}\alpha_j \cdot c \cdot d \cdot \beta_2{\downarrow}\alpha_j \cdot b \cdot \beta_3{\downarrow}\alpha_j) \cdot \mathsf{x}_j \cdots \beta{\downarrow}\alpha_n$ — looks like a read from memory situation. We check requirements (1)–(3) of Lemma 4.6 then. First, $\beta_2{\downarrow}\alpha_j$ must have no prop events to thread tid with the address a — holds as β_2 does not have them. Second, $\beta_3{\downarrow}\alpha_j$ must have no commits of earlier stores in thread tid — holds as β_3 does not have them. Third, $\beta{\downarrow}\alpha_l = (\beta_1 \cdot \beta_2 \cdot \beta_3){\downarrow}\alpha_l$, $l \in [i+1..n]$ must have no commit events for stores with indices $[1..i-1]$, the same address a and thread id tid. Consider $\beta_1{\downarrow}\alpha_l$ — if it has such an event e, then two stores to the same address, e and c, are committed in different order in α' and β, which is impossible due to $T(\alpha') = T(\beta)$. Consider $\beta_2 \downarrow \alpha_l$ — it does not have such an event, because β_2 does not have prop events to address a, therefore, it does not have commits of own stores there too. Consider $\beta_3{\downarrow}\alpha_l$ — it does not have such an event, because β_3 does not. Finally, none of x_l events, $l \in [i+1..n-1]$, must be a commit of earlier writes in thread tid — holds, as these events belong to the last fetched instruction of a thread.

Consider the case when load and commit events are in different parts, i.e., $\alpha = \alpha_1 \cdot \mathsf{x}_1 \cdots (\alpha'_j \cdot b \cdot \alpha''_j) \cdots (\alpha'_k \cdot c \cdot d \cdot \alpha''_k) \cdots \alpha_n$, $\beta = \beta_1 \cdot c \cdot d \cdot \beta_2 \cdot b \cdot \beta_3$, where b, c, d are defined as before and $i' < i$. Then, $\gamma = \beta \downarrow \alpha_1 \cdot \mathsf{x}_1 \cdots \beta \downarrow \alpha_j \cdot \mathsf{x}_j \cdots \beta{\downarrow}\alpha_k \cdots \beta{\downarrow}\alpha_n = \beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdots \beta_1{\downarrow}\alpha_j \cdot \beta_2{\downarrow}\alpha_j \cdot b \cdot \beta_3{\downarrow}\alpha_j \cdot \mathsf{x}_j \cdots \beta_1{\downarrow}\alpha_k \cdot c \cdot d \cdot \beta_2{\downarrow}\alpha_k \cdot \beta_3{\downarrow}\alpha_k \cdot \mathsf{x}_k \cdots \beta{\downarrow}\alpha_n$ — looks like an early read case. Therefore, one must check that $\beta_2{\downarrow}\alpha_k \cdot \beta_3{\downarrow}\alpha_k \cdot \mathsf{x}_k \cdots \beta{\downarrow}\alpha_n$ has no commit events matching $(\mathsf{commit}, \mathsf{tid}, [i'+1..i-1], *, \mathsf{a})$. Consider $\beta_2{\downarrow}\alpha_k$ — it does not have such events, because they would be immediately followed by a prop event to thread tid and address a, which contradicts requirement (2) of Lemma 4.6. Consider $\beta_3 \downarrow \alpha_k$ — it does not have such events, because β_3 does not have them by requirement (3) of Lemma 4.6. Consider $\beta{\downarrow}\alpha_l$, $l \in [j+1..n]$ — it does not have such events, because α_l do not have them by requirement (2) of Lemma 4.5. Finally, x_l, $l \in [j+1..n-1]$ belong to the last fetched instruction of a thread, therefore do not contain the described commit events. □

Lemma 4.13 (Reinsertion)**.** $\gamma \in \mathsf{C}_{power}(\mathcal{P})$.

Proof. We proceed by induction. Assume (1) $\gamma = \gamma_1 \cdot \mathsf{e} \cdot \gamma_2$, (2) $s_{\mathsf{power}_0} \xrightarrow{\gamma_1} s_{\mathsf{power}}$, and (3) all loads satisfied in γ_1 have read from the same stores as in α. We show that $s_{\mathsf{power}_0} \xrightarrow{\gamma_1 \cdot \mathsf{e}} s_{\mathsf{power}}{}'$ and all loads satisfied in $\gamma_1 \cdot \mathsf{e}$ have read from the same stores as in α. Let $s_{\mathsf{power}} = (\mathsf{ts}, s_Z)$ and $\mathsf{ts}(\mathsf{tid}) = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$. Consider the event e.

($\mathbf{fetch}, \mathbf{tid}, i$) A transition labeled by e from state s_{power} is feasible due to Lemma 4.11 and the fact that feasibility of a fetch transition is conditioned solely on the previous fetch transition with the same thread id.

($\mathbf{load}, \mathbf{tid}, i, \mathbf{a}$) For the transition to be feasible, $\mathsf{addr}(i) = \mathsf{a}$ must hold. In order to have $\mathsf{addr}(\mathsf{tid}, i) \neq \bot$, all loads in thread tid, on which $\mathsf{addr}(\mathsf{tid}, i)$ depends, must be satisfied. Note that these loads are the same in α and γ due to Lemma 4.11. Since $\alpha \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$, these load events occurred before e in α. Let e' be one of these load events. If $\mathsf{e}' \in \alpha_i$ and $\mathsf{e} \in \alpha_j$, $i < j$, or $\mathsf{e}' \in \{\mathsf{x}_i \mid i \in [1..n-1]\}$, or $\mathsf{e} \in \{\mathsf{x}_i \mid i \in [1..n-1]\}$, then e' and e are located in γ in the same order. If $\mathsf{e}', \mathsf{e} \in \alpha_i$, then $\mathsf{e}', \mathsf{e} \in \beta$. Since the $\rightarrow_{po}$ components of $T(\alpha)$ and $T(\beta)$ match up to a single deleted arc, e' and e are located in β (therefore, in $\beta{\downarrow}\alpha_i$ and γ) in this

order. By inductive assumption (3) and the fact that functions in FUN are deterministic, $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ holds.

Assume the load (tid, i) has read from a store (tid', i') in α. Then, by Lemmas 4.5, 4.6, 4.12, either conditions (1)–(3) of Lemma 4.6 hold, or conditions (1)–(2) of Lemma 4.5 hold. In the former case, $(\mathsf{prop}, \mathsf{tid}, \mathsf{tid}', i', \mathsf{a})$ is the last prop event to tid with address a, therefore, a load from memory transition reading (tid', i') is feasible from state s_{power}. In the latter case, (tid', i') is the latest non-committed store to address a, and an early read transition reading (tid', i') is possible. The proof that $\mathsf{addr}(\mathsf{tid}', i') \neq \bot$ is similar to the proof that $\mathsf{addr}(\mathsf{tid}, i) \neq \bot$.

($\mathbf{commit}, \mathbf{tid}, i$) The proof of $\mathsf{addr}(\mathsf{tid}, i) \neq \bot$ and $\mathsf{val}(\mathsf{tid}, i) \neq \bot$ is similar to the proof of $\mathsf{addr}(\mathsf{tid}, i) \neq \bot$ in the previous case. If $\mathsf{fetched}[i]$ is a load or a store, there must be no preceding loads and stores to unknown addresses, which holds and can be proven in a similar way. If $\mathsf{fetched}[i]$ is a load, requirement $\mathsf{loaded}[i] \neq \bot$ holds for the same reasons. If $\mathsf{fetched}[i]$ is a conditional, requirement $\mathsf{val}(\mathsf{tid}, i) \neq 0$ holds by inductive assumption (3), the fact that functions in FUN are deterministic, and the fact that $\alpha \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$.

($\mathbf{commit}, \mathbf{tid}, i, \mathbf{k}, \mathbf{a}$) Value k is unique, since it was unique in α, and α and γ consist of the same commit events. We check $\mathsf{co}(\mathsf{prop}(\mathsf{tid}, \mathsf{a})) < \mathsf{k}$. Assume it does not hold. Then, there is $\mathsf{e}' = (\mathsf{prop}, \mathsf{tid}, \mathsf{tid}', i', \mathsf{a})$, where $\mathsf{co}(\mathsf{tid}', i') > \mathsf{k}$, and e', e are located in γ in this order. If $\mathsf{e}' \in \alpha_i$, $\mathsf{e} \in \alpha_j$, $i < j$, or $\mathsf{e}' \in \{\mathsf{x}_i \mid i \in [1..n-1]\}$, or $\mathsf{e} \in \{\mathsf{x}_i \mid i \in [1..n-1]\}$, these events are located in α in this order, which contradicts $\alpha \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$. If $\mathsf{e}', \mathsf{e} \in \alpha_i$, these events are located in β in this order, which contradicts $\beta \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$.

This transition is immediately followed by a prop transition in γ, since it did so in α and β (unless $e \in \{\mathsf{x}_i \mid i \in [1..n-1]\}$, which is a simpler case), and by properties of projection.

($\mathbf{prop}, \mathbf{tid}, \mathbf{tid}', i', \mathbf{a}$) The requirement $\mathsf{co}(\mathsf{prop}(\mathsf{tid}, \mathsf{a})) < \mathsf{co}(\mathsf{tid}', i')$ is proven similarly to $\mathsf{co}(\mathsf{prop}(\mathsf{tid}, \mathsf{a})) < \mathsf{k}$ in the previous case.

As shown above, $s_{\mathsf{power}_0} \xrightarrow{\gamma} s_{\mathsf{power}}$. What is left to check, is that $s_{\mathsf{power}} \in F_{\mathsf{power}}$. The requirement that all fetched instructions are committed trivially holds: β includes the same commit events as α', therefore, by definition, γ contains the same commit events as α. The other two requirements that loads and stores agree with the coherence order hold due to Lemma 4.11, the inductive assumption (3), and the fact that α and γ consist of the same commit events (i.e., the coherence keys of matching stores are equal in these computations). □

Lemma 4.14. $T(\gamma) = T(\alpha)$

Proof. Equality of $\rightarrow_{po}$ follows from Lemma 4.11. Equality of source relation follows from Lemmas 4.5, 4.6, 4.12, 4.13. Store order is determined by a and k components of store commit events. Since computations α and γ consist of the same commit events, the $\rightarrow_{co}$ relations in the traces of α and γ are the same. □

Lemma 4.15. *$\gamma \in \mathsf{C}_{power}(\mathcal{P})$ and $T(\gamma) = T(\alpha)$.*

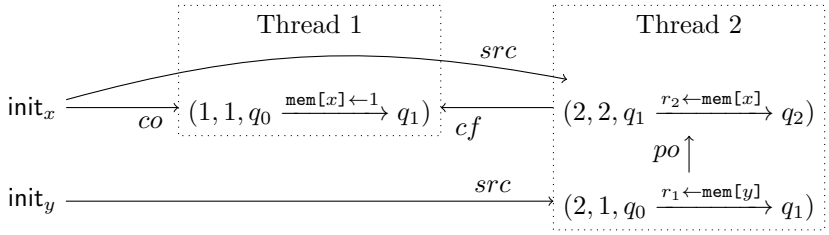

Figure 4.2: Trace of computations α' and β from Example 4.17.

Proof. Corollary of Lemmas 4.13 and 4.14. □

Without loss of generality we may assume that all fetch events of α are located within $\alpha_1 \cdot \mathsf{x}_1$: every thread can always first fetch all instructions and in the rest of the computation only execute them; such a reordering does not change the trace. Also, note that the maximal number of events an instruction can generate is $|\mathcal{P}| + 2$. This bound is achieved by a store that is fetched, committed, and propagated to all threads. Then the following lemma holds:

Lemma 4.16. *Computation γ is in normal form of degree $|\mathcal{P}| + 3$.*

Proof. By definition of γ and properties of projection. □

Together with Lemma 2.16 this proves Theorem 4.9.

Example 4.17. Consider $\alpha := \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a) \cdot \cancel{\mathsf{fetch}(b)} \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \cancel{\mathsf{commit}(b)} \cdot \cancel{\mathsf{prop}(b, 1)} \cdot \cancel{\mathsf{prop}(b, 2)} \cdot \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c)$, which is essentially σ_{MP} from Example 4.17 with fetch events moved to the front. We cancel the x_i events (crossed out) belonging to store instruction b, as b is the last instruction of thread 1 and no address depends on it (we could also cancel the events of d instead). Therefore, $\alpha_1 := \mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)$, $\alpha_2 := \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)$, $\alpha_3 := \alpha_4 := \varepsilon$, $\alpha_5 := \mathsf{load}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{commit}(c)$, and $\alpha' := \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot \alpha_4 \cdot \alpha_5$. The trace of α' is shown in Figure 4.2. The SC computation with the same trace is $\beta := \mathsf{fetch}(c) \cdot \mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d) \cdot \mathsf{fetch}(a) \cdot \mathsf{commit}(a) \cdot \mathsf{prop}(a, 1) \cdot \mathsf{prop}(a, 2)$. The normal-form computation is $\gamma := \beta{\downarrow}\alpha_1 \cdot \mathsf{x}_1 \cdots \beta{\downarrow}\alpha_5 = (\mathsf{fetch}(c) \cdot \mathsf{fetch}(d) \cdot \mathsf{fetch}(a)) \cdot \mathsf{fetch}(b) \cdot (\mathsf{commit}(a) \cdot \mathsf{prop}(a, 1)) \cdot \mathsf{commit}(b) \cdot \mathsf{prop}(b, 1) \cdot \mathsf{prop}(b, 2) \cdot (\mathsf{load}(c) \cdot \mathsf{commit}(c) \cdot \mathsf{load}(d) \cdot \mathsf{commit}(d))$. It is feasible and has the same trace as α and σ_{MP} (Figure 4.1).

4.4 From Robustness to Language Emptiness

We now reduce robustness to language emptiness. First, we define a multiheaded automaton capable of generating all normal-form computations of a program. Next, we intersect it with regular languages that check cyclicity of the happens-before relation. Altogether, the program is robust iff the intersection is empty.

4.4.1 Generating Normal-Form Computations

To generate all normal-form computations, we use multiheaded automata. We will generate all normal-form computations of program $\mathcal{P}$ with the n-headed automaton $M(\mathcal{P}) := (S_M, \mathrm{E}, \Delta_M, s_{M0}, F_M)$, where $n := |\mathcal{P}| + 3$. The automaton generates all events related to a single instruction in one shot, but, possibly, in different parts of the computation. All fetch events are generated in the first part of the computation. In order to generate them, the automaton keeps track of the destination state of the last fetched instruction in each thread (component ctrl-state of the automaton state).

Each instruction can only read the last value written to a register. Therefore, the automaton only needs to remember |REG| register values per thread (component reg-value). However, an instruction cannot be executed until the values of all registers that it reads become known. To obey this restriction, the automaton memorizes the part of the computation in which the register value gets computed (reg-comp-head). For example, while handling an assignment $r_1 \leftarrow r_1 + r_2$, the automaton learns that the new value of r_1 is the sum of the current values of r_1 and r_2. It also remembers that this value is available no earlier than the current values of r_1 and r_2 are computed. Similarly, the automaton remembers the parts of the computation in which the addresses of load and store instructions become known (addr-comp-head), and certain kinds of instructions get committed (reg-comm-head, assume-comm-head, addr-comm-head).

The automaton has to keep a separate memory state for each thread and for each part of the computation. The memory state of a thread in a part is updated when a store instruction gets propagated to this thread in this part. When a load instruction is handled, the automaton chooses a part where the load event takes place and uses the memory state of that part. Besides the memory valuation (mem-value), the memory state includes coherence keys (last-key) to guarantee that the generated computation respects the coherence order.

When starting the computation, the automaton non-deterministically guesses the memory valuations and coherence keys for all parts of the computation (except the first one). Upon termination, the automaton checks that the parts of the computation generated by each head fit together at the concatenation points. This ensures the overall computation is valid for the program. The trick is to remember the guess of the initial memory valuations and coherence keys in immutable components of the automaton state ($\mathsf{mem\text{-}value}_g$, $\mathsf{last\text{-}key}_g$). The final states require that the current memory state in part h of the computation coincides with the guessed initial state in part $\mathsf{h}+1$.

State space

A state from S_M (except the special initial state s_{M0}) includes the following information:

- ctrl-state(tid) gives the current control state of thread tid.
- reg-comp-head(tid, r) gives the part in which last value assigned to register r in thread tid gets computed.
- reg-value(tid, r) gives this computed value.

- reg-comm-head(tid, r) gives the part in which the last instruction assigning a value to register r in thread tid gets committed.
- assume-comm-head(tid) gives the part in which the latest fetched condition in thread tid is committed.
- mem-value(tid, a, h) gives the value of the last write to a propagated to thread tid in the part h or earlier.
- last-key(tid, a, h) gives the coherence key of the last write to a propagated to thread tid in the part h or earlier.
- mem-value$_g$, last-key$_g$ are immutable copies of the guessed values of the previous two components (see MH-GUESS below).
- early-mem-value(tid, a, h) gives the value written by the last fetched store to a which is still in-flight in the part h of computation, $\bot$ if there is no such store, $\top$ if the value of the store is unknown or there is a later in-flight store in this part with an unknown address.
- early-mem-key(tid, a, h) gives the coherence key of the store that produced early-mem-value(tid, a, h).
- addr-comp-head(tid) gives the leftmost part of the computation, in which the addresses of all already fetched memory accesses are computed.
- addr-comm-head(tid, a) gives the rightmost part of the computation having a commit to address a by thread tid.
- last-loaded-key(tid, a) gives the coherence key of the last store to address a loaded in thread tid.
- instr-count(tid) gives the number of instructions fetched in thread tid.

The initial state s_{M0} does not contain any information.

Transition relation

We define transitions by specifying the new (primed) values of the state components and the label λ of the transition. First, we define the transition guessing the initial memory state in each part of the computation:

MH-GUESS Assume the current state is s_{M0}. Then, there are transitions to the states satisfying the following requirements. First, all threads are in their initial control states: ctrl-state$'$:= λtid.$q_{\text{tid}0}$. All registers have initial value zero: reg-value$'$:= λtid.λr.0. Since no instructions modifying registers were executed yet, we assume that the current values have been computed and committed in the first part of the computation: reg-comp-head$'$:= λtid.λr.1, reg-comm-head$'$:= λtid.λr.1. Similarly, all conditionals are already committed in the first part: assume-comm-head$'$:= λtid.1. Since there were no stores executed, in no part of the computation a thread can read early: early-mem-value$'$:= λtid.λa.λh.$\bot$, early-mem-key$'$:= λtid.λa.λh.$\bot$. In the first part, the last

propagated stores to each address is the initial store writing zero and having zero coherence key: $\mathsf{mem\text{-}value}'(\mathsf{tid}, \mathsf{a}, 1) := 0$, $\mathsf{last\text{-}key}'(\mathsf{tid}, \mathsf{a}, 1) := 0$ for all $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$. We do not define the memory state for the other parts of the computation: they are chosen non-deterministically. However, we remember the non-deterministic guess: $\mathsf{mem\text{-}value}' = \mathsf{mem\text{-}value}'_g$, $\mathsf{last\text{-}key}' = \mathsf{last\text{-}key}'_g$. All so far fetched memory accesses (there is none) are already computed and committed in the first part: $\mathsf{addr\text{-}comp\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{addr\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.\lambda\mathsf{a}.1$. No loads were done yet: $\mathsf{last\text{-}loaded\text{-}key} := \lambda\mathsf{tid}.\lambda\mathsf{a}.0$. The number of fetched instructions in each thread is zero: $\mathsf{instr\text{-}count}' := \lambda\mathsf{tid}.0$. Finally, we require that the guessed values of coherence keys grow monotonically with the part index: $\mathsf{last\text{-}key}'(\mathsf{tid}, \mathsf{a}, \mathsf{h}) \leq \mathsf{last\text{-}key}'(\mathsf{tid}, \mathsf{a}, \mathsf{h}+1)$ for $\mathsf{h} \in [1..n-1]$, $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$ (we assume $\mathsf{last\text{-}key}'(\mathsf{tid}, \mathsf{a}, n) := \infty$). Guesses not satisfying this monotonicity requirement clearly lead to no valid computations. All transitions defined by this rule are labeled with $\lambda := \varepsilon$.

Fix a state s_M. We overload $\mathsf{eval}(\mathsf{tid}, e)$ to mean the value of expression e for the valuation of registers defined by $\lambda r.\mathsf{reg\text{-}value}(\mathsf{tid}, r)$.

Let $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{ctrl\text{-}state}(\mathsf{tid}) = q_1$, $\mathsf{instr} = q_1 \xrightarrow{\mathsf{cmd}} q_2 \in \mathcal{I}_{\mathsf{tid}}$. When the automaton executes the instruction instr, it chooses three indices from $\mathsf{HEAD} := [1..n]$. The first index $\mathsf{h}_1 := 1$ denotes the part of the computation where the fetch event is generated. The second index $\mathsf{h}_2 \in \mathsf{HEAD}$ gives the part of the computation where the result of the instruction is computed. For example, load events are generated in the part h_2. Clearly, the result of the instruction can be computed only after the instruction is fetched, $\mathsf{h}_2 \geq \mathsf{h}_1$, and all data and address dependencies are computed: $\mathsf{h}_2 \geq \mathsf{reg\text{-}comp\text{-}head}(\mathsf{tid}, r)$ for each register r read in cmd. The third index, $\mathsf{h}_3 \in \mathsf{HEAD}$, denotes the part in which the instruction is committed. The instruction must be computed before it is committed: $\mathsf{h}_3 \geq \mathsf{h}_2$. Moreover, all the data and control dependencies must be committed earlier: $\mathsf{h}_3 \geq \mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid}, r)$ for each register r read in cmd, $\mathsf{h}_3 \geq \mathsf{assume\text{-}comm\text{-}head}(\mathsf{tid})$. When executing an instruction in thread tid, we increment the counter of fetched instructions in this thread: let $i := \mathsf{instr\text{-}count}(\mathsf{tid}) + 1$, then $\mathsf{instr\text{-}count}' := \mathsf{instr\text{-}count}[\mathsf{tid} := i]$.

Depending on the type of cmd, there are the following transitions from s_M labeled by events λ:

MH-ASSIGN Let cmd be a local assignment: $\mathsf{cmd} = r \leftarrow e_{\mathsf{v}}$. The assigned value is $\mathsf{v} := \mathsf{eval}(\mathsf{tid}, e_{\mathsf{v}})$. We remember that the register r now contains the new value: $\mathsf{reg\text{-}value}' := \mathsf{reg\text{-}value}[(\mathsf{tid}, r) := \mathsf{v}]$. We note that the value is computed in part h_2: $\mathsf{reg\text{-}comp\text{-}head}' := \mathsf{reg\text{-}comp\text{-}head}[(\mathsf{tid}, r) := \mathsf{h}_2]$ and that the instruction producing this value is committed in part h_3: $\mathsf{reg\text{-}comm\text{-}head}' := \mathsf{reg\text{-}comm\text{-}head}[(\mathsf{tid}, r) := \mathsf{h}_3]$. The transition is labeled with $\lambda := (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$. For brevity we allow a single transition to be labeled with several events. An automaton with such transitions can be trivially translated to the canonical form by breaking one such transition into several consecutive ones.

MH-ASSUME Consider a conditional: $\mathsf{cmd} = \texttt{assume}(e_{\mathsf{v}})$. Assume the condition holds: $\mathsf{eval}(\mathsf{tid}, e_{\mathsf{v}}) \neq 0$. We remember the part in which the condition gets committed: $\mathsf{assume\text{-}comm\text{-}head}' := \mathsf{assume\text{-}comm\text{-}head}[\mathsf{tid} := \mathsf{h}_3]$. We label the transition with $\lambda := (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$.

MH-LOAD Let cmd be a load: $\mathsf{cmd} = r \leftarrow \mathtt{mem}[e_\mathsf{a}]$. The address being accessed is $\mathsf{a} := \mathsf{eval}(\mathsf{tid}, e_\mathsf{a})$. We require $\mathsf{h}_3 \geq \mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a})$, because memory accesses to the same address must be committed in order. If $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}) = \bot$, the load reads from the last propagated store: $\mathsf{v} := \mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$, $\mathsf{k} := \mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$. Otherwise, the load reads early from an earlier store: $\mathsf{v} := \mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$, $\mathsf{k} := \mathsf{early\text{-}mem\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$. We require $\mathsf{v} \neq \top$: the contrary would mean that there is an in-flight store to address a whose value is not yet computed. Also, we require $\mathsf{k} \geq \mathsf{last\text{-}loaded\text{-}key}(\mathsf{tid}, \mathsf{a})$, in order to guarantee POW-FIN-LD to hold in the final state.

If all the requirements hold, we update the register valuation, similar to MH-ASSIGN rule: $\mathsf{reg\text{-}value}' := \mathsf{reg\text{-}value}[(\mathsf{tid}, r) := \mathsf{v}]$, $\mathsf{reg\text{-}comp\text{-}head}' := \mathsf{reg\text{-}comp\text{-}head}[(\mathsf{tid}, r) := \mathsf{h}_2]$, $\mathsf{reg\text{-}comm\text{-}head}' := \mathsf{reg\text{-}comm\text{-}head}[(\mathsf{tid}, r) := \mathsf{h}_3]$. We update the index of the leftmost part of the computation where all addresses are computed: $\mathsf{addr\text{-}comp\text{-}head}' := \mathsf{addr\text{-}comp\text{-}head}[\mathsf{tid} := \max\{\mathsf{addr\text{-}comp\text{-}head}(\mathsf{tid}), \mathsf{h}_2\}]$. Part h_3 is now the rightmost part containing a commit of a memory access to a: $\mathsf{addr\text{-}comm\text{-}head}' := \mathsf{addr\text{-}comm\text{-}head}[(\mathsf{tid}, \mathsf{a}) := \mathsf{h}_3]$. The coherence key of the store loaded by the program-order-last load is now k: $\mathsf{last\text{-}loaded\text{-}key}' := \mathsf{last\text{-}loaded\text{-}key}[(\mathsf{tid}, \mathsf{a}) := \mathsf{k}]$. We label the transition with $\lambda := (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_2, \mathsf{load}, \mathsf{tid}, i, \mathsf{a}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$.

MH-STORE Consider a store $\mathsf{cmd} = \mathtt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}$. The address being written is $\mathsf{a} := \mathsf{eval}(\mathsf{tid}, e_\mathsf{a})$. The store cannot be committed before the addresses of all previously fetched instructions become known. Therefore, we require $\mathsf{h}_3 \geq \mathsf{addr\text{-}comp\text{-}head}(\mathsf{tid})$. Similarly, the store can be committed only after all previously fetched accesses to the same address are committed: $\mathsf{h}_3 \geq \mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a})$. The value being stored is $\mathsf{v} := \mathsf{eval}(\mathsf{tid}, e_\mathsf{v})$. The automaton non-deterministically chooses a unique coherence key $\mathsf{k} \in \mathbb{Q}$, $\mathsf{k} \neq \mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h})$ for any $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$.

The transition remembers that in the parts $[\mathsf{h}_1..\mathsf{h}_2 - 1]$ the value or the address of the last in-flight store is not known, and in the parts $[\mathsf{h}_2..\mathsf{h}_3 - 1]$ the address is known and the value being written to this address is v: $\mathsf{early\text{-}mem\text{-}value}' := \mathsf{early\text{-}mem\text{-}value}[(\mathsf{tid}, \mathsf{a}, [\mathsf{h}_1..\mathsf{h}_2 - 1]) := \top), (\mathsf{tid}, \mathsf{a}, [\mathsf{h}_2..\mathsf{h}_3 - 1]) := \mathsf{v}]$, $\mathsf{early\text{-}mem\text{-}key}' := \mathsf{early\text{-}mem\text{-}key}[(\mathsf{tid}, \mathsf{a}, [\mathsf{h}_2..\mathsf{h}_3 - 1]) := \mathsf{k}]$. Let $\mathsf{h}_2' \in [\mathsf{h}_1..\mathsf{h}_2]$ be an index satisfying $\mathsf{h}_2' \geq \mathsf{reg\text{-}comp\text{-}head}(\mathsf{tid}, r)$ for each register r used in e_a. In the parts $[\mathsf{h}_1..\mathsf{h}_2' - 1]$ the thread has an in-flight store whose address is not known, therefore, early reads are not possible in these parts: we modify $\mathsf{early\text{-}mem\text{-}value}' := \mathsf{early\text{-}mem\text{-}value}'[(\mathsf{tid}, \mathsf{a}', \mathsf{h}) := \top]$ for all $\mathsf{a}' \in \mathsf{ADDR} \setminus \{\mathsf{a}\}$, $\mathsf{h} \in [\mathsf{h}_1..\mathsf{h}_2' - 1]$ with $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}', \mathsf{h}) \in \mathsf{DOM}$. We update the index of the leftmost part of the computation where all addresses are computed: $\mathsf{addr\text{-}comp\text{-}head}' := \mathsf{addr\text{-}comp\text{-}head}[\mathsf{tid} := \max\{\mathsf{addr\text{-}comp\text{-}head}(\mathsf{tid}), \mathsf{h}_2'\}]$. Part h_3 is now the rightmost part of the computation containing a commit of a memory access to a: $\mathsf{addr\text{-}comm\text{-}head}' := \mathsf{addr\text{-}comm\text{-}head}[(\mathsf{tid}, \mathsf{a}) := \mathsf{h}_3]$.

Finally, we choose, to which other threads and in which parts of the computation the store will be propagated. Let $T \subseteq \mathsf{TID} \setminus \{\mathsf{tid}\}$ be

the set of the threads (except tid) to which the store will be propagated. Let initially $\mathsf{mem\text{-}value}' := \mathsf{mem\text{-}value}$, $\mathsf{last\text{-}key}' := \mathsf{last\text{-}key}$, and $\lambda := (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i, \mathsf{k}, \mathsf{a})$. For $\mathsf{tid}' = \mathsf{tid}$ and for each $\mathsf{tid}' \in T$ we propagate the store to tid'. For this, we choose a part for the corresponding propagate event: $\mathsf{h} \in \mathsf{HEAD}$, $\mathsf{h} \geq \mathsf{h}_3$ ($\mathsf{h} := \mathsf{h}_3$ for $\mathsf{tid}' = \mathsf{tid}$). We check that the propagation respects the coherence order: $\mathsf{last\text{-}key}(\mathsf{tid}', \mathsf{a}, \mathsf{h}) < \mathsf{k} \leq \mathsf{last\text{-}key}_g(\mathsf{tid}', \mathsf{a}, \mathsf{h}+1)$. Then, we update the memory state $\mathsf{mem\text{-}value}' := \mathsf{mem\text{-}value}'[(\mathsf{tid}', \mathsf{a}, \mathsf{h}) := \mathsf{v}]$, $\mathsf{last\text{-}key}' := \mathsf{last\text{-}key}'[(\mathsf{tid}', \mathsf{a}, \mathsf{h}) := \mathsf{k}]$, and the label $\lambda := \lambda \cdot (\mathsf{h}, \mathsf{prop}, \mathsf{tid}', \mathsf{tid}, i, \mathsf{a})$.

Final states

The set of final states F_M is a subset of $S_M \setminus \{s_{M0}\}$ consisting of all states with $\mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = \mathsf{mem\text{-}value}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h}+1)$, $\mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = \mathsf{last\text{-}key}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h}+1)$ for all $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in [1..n-1]$.

Soundness and completeness

In the following proofs we use a dot notation for referencing elements of tuples (the same notation is used in many programming languages). For example, let s_{power} be a state of the Power automaton $X_{\mathsf{power}}(\mathcal{P})$, then $s_{\mathsf{power}}.\mathsf{ts}$ is the ts component of the state, i.e., the first element of the tuple s_{power}.

Lemma 4.18. $\mathcal{L}(M) \subseteq \mathcal{C}_{power}(\mathcal{P})$.

Proof. Consider $\sigma = \lambda_1 \cdots \lambda_m$, such that $s_{M0} \xrightarrow{\lambda_1} s_{M1} \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_m} s_{Mm} \in F_M$. For $\mathsf{h} \in \mathsf{HEAD}$, let $\tau^s_{\mathsf{h}} := \mathsf{take2nd}((\lambda_1 \cdots \lambda_s)\!\downarrow\!(\{\mathsf{h}\} \times \mathrm{E}))$, $s \in [0..m]$.

Let $({s_{\mathsf{power}}}_1^0 \ldots {s_{\mathsf{power}}}_n^0) \in (S_{\mathsf{power}})^n$ be the states of $X_{\mathsf{power}}(\mathcal{P})$ defined so that SND-B holds for $s = 0$ (see below). By induction on $s \in [1..m]$ we show:

SND-A ${s_{\mathsf{power}}}_{\mathsf{h}}^0 \xrightarrow{\tau^s_{\mathsf{h}}} {s_{\mathsf{power}}}_{\mathsf{h}}^s$.

SND-B For all $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in \mathsf{HEAD}$, ${s_{\mathsf{power}}}_{\mathsf{h}}^s = (\mathsf{ts}, (\mathsf{co}, \mathsf{prop}))$, $\mathsf{ts}(\mathsf{tid}) = (\mathsf{fetched}, \mathsf{committed}, \mathsf{loaded})$ holds:

- **SND-B1** $\mathsf{fetched}$ is the list of instructions fetched by $(\tau_1^m \cdots \tau_{\mathsf{h}-1}^m \cdot \tau_{\mathsf{h}}^s)\!\downarrow\! \mathsf{fetch}\!\downarrow\!\mathsf{tid}$.
- **SND-B2** $\mathsf{committed}$ consists of the indices of instructions committed by $(\tau_1^m \cdots \tau_{\mathsf{h}-1}^m \cdot \tau_{\mathsf{h}}^s)\!\downarrow\!\mathsf{commit}\!\downarrow\!\mathsf{tid}$.
- **SND-B3** loaded contains the information about the stores being read by loads in $(\tau_1^m \cdots \tau_{\mathsf{h}-1}^m \cdot \tau_{\mathsf{h}}^s)$ determined according to Lemmas 4.5 and 4.6.
- **SND-B4** $\mathsf{co}(\mathsf{tid}, i) = \mathsf{k}$ if $(\mathsf{commit}, \mathsf{tid}, i, \mathsf{k}, \mathsf{a}) \in \tau_1^m \cdots \tau_{\mathsf{h}-1}^m \cdot \tau_{\mathsf{h}}^s$ for some $\mathsf{a} \in \mathsf{ADDR}$, otherwise, $\mathsf{co}(\mathsf{tid}, i) = \bot$.
- **SND-B5** $\mathsf{prop}(\mathsf{tid}, \mathsf{a}) = (\mathsf{tid}', i')$ if $(\mathsf{prop}, \mathsf{tid}, \mathsf{tid}', i', \mathsf{a}) = \mathsf{last}((\tau_1^m \cdots \tau_{\mathsf{h}-1}^m \cdot \tau_{\mathsf{h}}^s)\!\downarrow\!(\mathsf{prop}, \mathsf{tid}, *, *, \mathsf{a}))$, otherwise, $\mathsf{prop}(\mathsf{tid}, \mathsf{a}) = \mathsf{init}_{\mathsf{a}}$.

SND-C For each $\mathsf{tid} \in \mathsf{TID}$: $\mathsf{ctrl\text{-}state}(\mathsf{tid}) = \mathsf{dst}(\mathsf{last}({s_{\mathsf{power}}}_1^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}))$ (or q_{tid_0} if no instructions were fetched).

SND-D For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, for each $\mathsf{h} \in [\mathsf{reg\text{-}comp\text{-}head}(\mathsf{tid}, r)..n]$: $\mathsf{reg\text{-}value}(\mathsf{tid}, r) = \mathsf{eval}(\mathsf{tid}, \mathsf{instr\text{-}count}(\mathsf{tid}) + 1, r)$ computed for the state $s_{\mathsf{power}_\mathsf{h}}^s$.

SND-E For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, $\mathsf{h} \in [\mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid}, r)..n]$: let i be the index of the latest instruction in $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}$ writing to r, then $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

SND-F For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in [\mathsf{assume\text{-}comm\text{-}head}(\mathsf{tid})..n]$: $s_{\mathsf{power}_\mathsf{h}}^s$ does not contain uncommitted conditional instructions in thread tid having indices $\leq \mathsf{instr\text{-}count}(\mathsf{tid})$.

SND-G For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: let $w := s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{prop}(\mathsf{tid}, \mathsf{a})$. If $w = \mathsf{init}_\mathsf{a}$, $\mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = 0$. If $w = (\mathsf{tid}', i')$, $\mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = \mathsf{val}(\mathsf{tid}', i')$ computed in $s_{\mathsf{power}_\mathsf{h}}^s$.

SND-H For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: $\mathsf{last\text{-}key}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h}) \leq s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{co}(s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{prop}(\mathsf{tid}, \mathsf{a})) = \mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) \leq \mathsf{last\text{-}key}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h}+1)$.

SND-K For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: let $i \in \mathbb{N}$ be the maximal index, such that $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}[i]$ is a store, $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ in $s_{\mathsf{power}_n}^s$. Let i' be the maximal index, such that $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}[i']$ is a store, $\mathsf{addr}(\mathsf{tid}, i') \in \{\bot, \mathsf{a}\}$ in $s_{\mathsf{power}_\mathsf{h}}^s$. Then $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \bot$ if such i does not exist or $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$. Otherwise, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \top$ if $\mathsf{addr}(\mathsf{tid}, i') = \bot$ or $\mathsf{val}(\mathsf{tid}, i) = \bot$ in $s_{\mathsf{power}_\mathsf{h}}^s$. Otherwise, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \mathsf{val}(\mathsf{tid}, i)$ computed in $s_{\mathsf{power}_\mathsf{h}}^s$ and $\mathsf{early\text{-}mem\text{-}key}(\mathsf{tid}, i, \mathsf{h}) = s_{\mathsf{power}_\mathsf{h}}^s.s_Z.\mathsf{co}(\mathsf{tid}, i)$.

SND-L For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in [\mathsf{addr\text{-}comp\text{-}head}(\mathsf{tid})..n]$, $i \in [1..|s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}|]$: $\mathsf{addr}(\mathsf{tid}, i) \neq \bot$ in $s_{\mathsf{power}_\mathsf{h}}^s$.

SND-M For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in [\mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a})..n]$: if $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ in $s_{\mathsf{power}_n}^s$ for some i, then $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

Finally we will show that $s_{\mathsf{power}_\mathsf{h}}^m = s_{\mathsf{power}_{\mathsf{h}+1}}^0$ for all $\mathsf{h} \in [1..n-1]$ and $s_{\mathsf{power}_n}^m \in F_{\mathsf{power}}$, thus proving the claim of the lemma.

Base case: $s = 1$, we must show that s_{M1} satisfies the inductive statement. This is easy to check by definition of the destination state of MH-GUESS transition.

Step case: assume the inductive statement holds for some $s \in [0..m-1]$. Consider λ_s (for notational convenience and without loss of generality we assume below that $\mathsf{h}_j \neq \mathsf{h}_{j'}$ for $j \neq j'$):

Assignment $\lambda_s = (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$, $\mathsf{instr} = q_1 \xrightarrow{r \leftarrow e_\mathsf{v}} q_2$. Let $\mathsf{e}_1 := (\mathsf{fetch}, \mathsf{tid}, \mathsf{instr})$, $\mathsf{e}_3 := (\mathsf{commit}, \mathsf{tid}, i)$.

We need to show that $s_{\mathsf{power}_{\mathsf{h}_1}}^{s-1} \xrightarrow{\mathsf{e}_1} s_{\mathsf{power}_{\mathsf{h}_1}}^s$, i.e., that the assignment instruction can be fetched. This follows from the choice of $\mathsf{h}_1 := 1$ in MH-ASSIGN and SND-B1, SND-C.

We also need to show that $s_{\mathsf{power}_{\mathsf{h}_3}}^{s-1} \xrightarrow{\mathsf{e}_3} s_{\mathsf{power}_{\mathsf{h}_3}}^s$, i.e., that the assignment instruction can be committed. First, the e_3 transition requires the instruction being committed to be fetched, which holds due to SND-B1 and $\mathsf{h}_3 \geq \mathsf{h}_1$. Second, this instruction must be not committed yet, which holds

by SND-B2 and the fact that M commits each instruction once. Third, all control dependencies must be committed. This is by the choice of h_3 in MH-ASSIGN and SND-F. Fourth, all the preceding data dependencies must be committed. This is by the choice of h_3 in MH-ASSIGN and SND-E. Finally, the argument of the function must be computed. This is by choice of $\mathsf{h}_3 \geq \mathsf{h}_2$ in MH-ASSIGN, Lemma 4.4, and SND-D.

In the end, we must show that the invariants hold in the new state. The only non-trivial thing is SND-D, which holds due to SND-D in the source state, definition of v in MH-ASSIGN, definitions of eval, and the fact that functions in FUN are deterministic.

Assume $\lambda_s = (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, q_1 \xrightarrow{\mathsf{instr}} q_2) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$, $\mathsf{instr} = \texttt{assume}(e_\mathsf{v})$. The proof is similar to the previous case. The commit transition additionally requires $\mathsf{eval}(\mathsf{tid}, i, e_\mathsf{v}) \neq 0$, which holds due to the fact that a similar check in MH-ASSUME holds, SND-D, definitions of eval, the fact that functions in FUN are deterministic.

Load $\lambda_s = (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_2, \mathsf{load}, \mathsf{tid}, i, \mathsf{a}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i)$, $\mathsf{instr} = r \leftarrow \texttt{mem}[e_\mathsf{a}]$. Let $\mathsf{e}_1 := (\mathsf{fetch}, \mathsf{tid}, \mathsf{instr})$, $\mathsf{e}_2 := (\mathsf{load}, \mathsf{tid}, i, \mathsf{a})$, $\mathsf{e}_3 := (\mathsf{commit}, \mathsf{tid}, i)$.

$s_{\mathsf{power}_{\mathsf{h}_1}}^{s-1} \xrightarrow{\mathsf{e}_1} s_{\mathsf{power}_{\mathsf{h}_1}}^{s}$ holds for the same reasons as before.

Next, we show that $s_{\mathsf{power}_{\mathsf{h}_2}}^{s-1} \xrightarrow{\mathsf{e}_2} s_{\mathsf{power}_{\mathsf{h}_2}}^{s}$, where this transition is a POW-EARLY transition in the early read case of MH-LOAD and a POW-LOAD transition in the load from memory case. First, we must show that $s_{\mathsf{power}_\mathsf{h}}^{s}.\mathsf{ts}(\mathsf{tid}).\mathsf{loaded}[i] = \bot$. This holds by SND-B3 and the fact that M generates a load event once for a single fetched load instruction.

Assume the early read case. This means, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2) \in \mathsf{DOM}$. By SND-K, this means, the last fetched store with an unknown address or address of the load is not yet committed, has the address of the load and has the value known. By POW-EARLY, the load can take the value from this store, and SND-B3 holds in the new state.

Consider the load from memory case. This means, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2) = \bot$. By SND-K, this means, there is no earlier fetched store with the same address which is not yet committed. By POW-LOAD, the load can take the value from the last propagated store, and SND-B3 holds in the new state.

Argumentation for $s_{\mathsf{power}_{\mathsf{h}_3}}^{s-1} \xrightarrow{\mathsf{e}_3} s_{\mathsf{power}_{\mathsf{h}_3}}^{s}$ is similar to the previous cases. Additionally, first we must show that $s_{\mathsf{power}_\mathsf{h}}^{s}.\mathsf{ts}(\mathsf{tid}).\mathsf{loaded}[i] \neq \bot$. This is by $\mathsf{h}_3 \geq \mathsf{h}_2$ (MH-LOAD), SND-B3. Second, we must ensure that all preceding instructions accessing the same address a are committed, and there are no previously fetched instructions with unknown address. This holds by choice of h_3 in MH-LOAD, SND-L, and SND-M.

In the new state, SND-D holds by definition of v in POW-LOAD, definitions of eval, SND-G, and SND-K. Proofs for the other conditions are simpler.

Store $\lambda_s = (\mathsf{h}_1, \mathsf{fetch}, \mathsf{tid}, \mathsf{instr}) \cdot (\mathsf{h}_3, \mathsf{commit}, \mathsf{tid}, i, \mathsf{k}, \mathsf{a}) \cdot (\mathsf{h}_3, \mathsf{prop}, \mathsf{tid}, \mathsf{tid}, i, \mathsf{a}) \cdot (\mathsf{h}_4, \mathsf{prop}, \mathsf{tid}_1, \mathsf{tid}, i, \mathsf{a}) \cdots (\mathsf{h}_{u+3}, \mathsf{prop}, \mathsf{tid}_u, \mathsf{tid}, i, \mathsf{a})$. Let $\mathsf{e}_1 := (\mathsf{fetch}, \mathsf{tid},$

$\mathsf{instr})$, $\mathsf{e}_3 := (\mathsf{commit}, \mathsf{tid}, i, \mathsf{k}, \mathsf{a})$, $\mathsf{e}_4 := (\mathsf{prop}, \mathsf{tid}, \mathsf{tid}, i, \mathsf{a})$, $\mathsf{e}_{j+3} := (\mathsf{prop}, \mathsf{tid}_j, \mathsf{tid}, i, \mathsf{a})$ for $j \in [1..u]$.

$s_{\mathsf{power}_{\mathsf{h}_1}}^{s-1} \xrightarrow{\mathsf{e}_1} s_{\mathsf{power}_{\mathsf{h}_1}}^{s}$ holds for the same reasons as before.

$s_{\mathsf{power}_{\mathsf{h}_3}}^{s-1} \xrightarrow{\mathsf{e}_3} s_{\mathsf{power}_{\mathsf{h}_3}}^{s}$ holds for the same reasons as in the case of a load. The requirement that the coherence key is unique in POW-STORE follows from a similar requirement in MH-STORE and SND-H. By POW-STORE, the only available transition from $s_{\mathsf{power}_{\mathsf{h}_2}}^{s}$ is a propagation of the write to its thread, i.e., e_4, which indeed follows e_3 in τ. Next, we show that e_4 and further propagate transitions are feasible.

First, POW-PROP-STORE rule requires the write being propagated to have a coherence key (i.e., to be committed), which holds by choice of h_j, $j \in [3..u+3]$ in MH-STORE and SND-B2. Second, it requires the coherence key of the latest propagated store to be less than the key of the store being propagated. This is adhered due to the check $\mathsf{last\text{-}key}(\mathsf{tid}', \mathsf{a}, \mathsf{h}) < \mathsf{k}$ and SND-H.

It is easy to see that the inductive invariants hold in the new state as well.

Now we prove $s_{\mathsf{power}_{\mathsf{h}}}^{m} = s_{\mathsf{power}_{\mathsf{h}+1}}^{0}$ for all $\mathsf{h} \in [1..n-1]$. The equality of ts components immediately follows from SND-B inductive statement.

Now we prove $s_{\mathsf{power}_n}^{m} \in F_{\mathsf{power}}$. POW-FIN-COMM holds, because $X_{\mathsf{power}}(\mathcal{P})$ always emits a commit event for each fetched instruction. POW-FIN-LD holds by the requirement on the coherence key of the loaded store in MH-LOAD. POW-FIN-LD-ST is proven by case consideration: a load reading from an earlier store early, a load reading from the last store to the loaded address in the same thread, a load reading from a store propagated after the last store to the same address in the load's thread is committed. □

Lemma 4.19. *$\{\tau \in \mathsf{C}_{\mathit{power}}(\mathcal{P}) \mid \tau$ is in normal form of degree $n\} \subseteq \mathcal{L}(M)$.*

Proof. Let $\tau = \tau_1 \cdots \tau_n \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P})$ be a normal-form computation, i.e., $s_{\mathsf{power}_0} \xrightarrow{\tau} s_{\mathsf{power}} \in F_{\mathsf{power}}$. We show that there is a sequence of transitions $s_{M0} \xrightarrow{\lambda_1} s_{M1} \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_m} s_{Mm} \in F_M$, such that $\tau_{\mathsf{h}} = \mathsf{take2nd}((\lambda_1 \cdots \lambda_n) \downarrow (\{\mathsf{h}\} \times \mathrm{E}))$.

Let $\tau_{\mathsf{h}}^{s} := \mathsf{take2nd}((\lambda_1 \cdots \lambda_s) \downarrow (\{\mathsf{h}\} \times \mathrm{E}))$, $s_{\mathsf{power}_0} \xrightarrow{\tau_1 \cdots \tau_{\mathsf{h}-1} \cdot \tau_{\mathsf{h}}^{s}} s_{\mathsf{power}_{\mathsf{h}}}^{s} \rightarrow^{*} s_{\mathsf{power}}$. By induction on s starting from 1 we show the following inductive statements:

CMPL-A There is a sequence of s transitions: $s_{M0} \xrightarrow{\lambda_1} s_{M1} \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_s} s_{Ms}$.

CMPL-B For all $\mathsf{h} \in \mathsf{HEAD}$: $\tau_{\mathsf{h}} = \tau_{\mathsf{h}}^{s} . \overline{\tau_{\mathsf{h}}^{s}}$ for some $\overline{\tau_{\mathsf{h}}^{s}}$.

CMPL-C If $\mathsf{e}_1, \mathsf{e}_2 \in \tau$ are two events belonging to instruction (tid, i), then $\mathsf{e}_1 \in \tau_{\mathsf{h}}^{s}$ for some h iff $\mathsf{e}_2 \in \tau_{\mathsf{h}'}^{s}$ for some h'.

CMPL-D For each $\mathsf{tid} \in \mathsf{TID}$: $\mathsf{ctrl\text{-}state}(\mathsf{tid}) = \mathsf{dst}(\mathsf{last}(s_{\mathsf{power}_1}^{s}.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}))$ (or $\mathsf{ctrl\text{-}state}(\mathsf{tid}) = q_{\mathsf{tid}_0}$ if no instructions were fetched).

CMPL-F For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, $\mathsf{h} \in [\mathsf{reg\text{-}comp\text{-}head}(\mathsf{tid}, r)..n]$: $\mathsf{reg\text{-}value}(\mathsf{tid}, r) = \mathsf{eval}(\mathsf{tid}, \mathsf{instr\text{-}count}(\mathsf{tid}) + 1, r)$ computed in the state $s_{\mathsf{power}_{\mathsf{h}}}^{s}$.

CMPL-F' For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, $\mathsf{h} \in [1..\mathsf{reg\text{-}comp\text{-}head}(\mathsf{tid}, r) - 1]$: $\mathsf{eval}(\mathsf{tid}, \mathsf{instr\text{-}count}(\mathsf{tid}) + 1, r) = \bot$.

CMPL-G For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, $\mathsf{h} \in [\mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid})..n]$: let i be the index of the last instruction in $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}$ writing to r, then $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-G' For each $\mathsf{tid} \in \mathsf{TID}$, $r \in \mathsf{REG}$, $\mathsf{h} \in [1..\mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid}, r) - 1]$: let i be the index of the last instruction in $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}$, then $i \notin s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-K For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in [\mathsf{assume\text{-}comm\text{-}head}(\mathsf{tid})..n]$: let i be an index of an `assume()` instruction in $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}$, then $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-K' For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in [1..\mathsf{assume\text{-}comm\text{-}head}(\mathsf{tid}) - 1]$: let i be an index of the last `assume()` instruction in $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}$, then $i \notin s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-L For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: let $w := s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{prop}(\mathsf{tid}, \mathsf{a})$. If $w = \mathsf{init}_\mathsf{a}$, $\mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = 0$. If $w = (\mathsf{tid}', i')$, $\mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) = \mathsf{val}(\mathsf{tid}', i')$ computed in $s_{\mathsf{power}_\mathsf{h}}^s$.

CMPL-M For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: $\mathsf{last\text{-}key}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h}) < s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{co}(s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{prop}(\mathsf{tid}, \mathsf{a})) = \mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}) \leq \mathsf{last\text{-}key}_g(\mathsf{tid}, \mathsf{a}, \mathsf{h} + 1)$.

CMPL-N For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in \mathsf{HEAD}$: let $i \in \mathbb{N}$ be the maximal index, such that $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}[i]$ is a store, $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ in $s_{\mathsf{power}}^s{}_n$. Let i' be the maximal index, such that $s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}[i']$ is a store, $\mathsf{addr}(\mathsf{tid}, i') \in \{\bot, \mathsf{a}\}$ in $s_{\mathsf{power}_\mathsf{h}}^s$. Then $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \bot$ if such i does not exist or $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$. Otherwise, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \top$ if $\mathsf{addr}(\mathsf{tid}, i') = \bot$ or $\mathsf{val}(\mathsf{tid}, i) = \bot$ in $s_{\mathsf{power}_\mathsf{h}}^s$. Otherwise, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, i, \mathsf{h}) = \mathsf{val}(\mathsf{tid}, i)$ computed in $s_{\mathsf{power}_\mathsf{h}}^s$.

CMPL-P For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in [\mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a})..n]$: if $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ in $s_{\mathsf{power}}^s{}_n$ for some i, then $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-P' For each $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{a} \in \mathsf{ADDR}$, $\mathsf{h} \in [1..\mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a}) - 1]$: there is i with $\mathsf{addr}(\mathsf{tid}, i) = \mathsf{a}$ in $s_{\mathsf{power}}^s{}_n$, such that $i \in s_{\mathsf{power}_\mathsf{h}}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{committed}$.

CMPL-R For each $\mathsf{tid} \in \mathsf{TID}$: $\mathsf{instr\text{-}count}(\mathsf{tid}) = |s_{\mathsf{power}_1}^s.\mathsf{ts}(\mathsf{tid}).\mathsf{fetched}|$.

Base case: $s = 1$. We choose the first (MH-GUESS) transition $s_{M0} \xrightarrow{\lambda_1} s_{M1}$, so that the inductive statements hold:

Guess We define mem-value and last-key components of s_{M1} according to CMPL-L and CMPL-M requirements. The other inductive statements trivially hold.

Assume the inductive statements hold for s and $\overline{\tau_\mathsf{h}^s} \neq \varepsilon$ for some $\mathsf{h} \in \mathsf{HEAD}$. We show they hold for $s' := s + 1$. The proof is done by pointing out an appropriate transition $s_{Ms} \xrightarrow{\lambda_{s+1}} s_{Ms+1}$. We choose the first possible option out of the following:

Assignment Assume $\mathsf{e}_1 \sqsubseteq \overline{\tau^s_{\mathsf{h}_1}}$, $\mathsf{e}_3 \sqsubseteq \overline{\tau^s_{\mathsf{h}_3}}$, where $\mathsf{h}_1 < \mathsf{h}_3$ ($\mathsf{h}_1 = \mathsf{h}_3$ is possible, but here and further we write strict inequalities for notational convenience), $\mathsf{e}_1 := (\mathsf{fetch}, \mathsf{tid}, q_1 \xrightarrow{\mathsf{cmd}} q_2)$, $\mathsf{e}_3 := (\mathsf{commit}, \mathsf{tid}, i)$, $\mathsf{h}_1 = 1$, $i = \mathsf{instr\text{-}count}(\mathsf{tid})$, $\mathsf{cmd} = r \leftarrow e_{\mathsf{v}}$. Then, as we show next, an MH-ASSIGN transition is feasible.

First, $s_{\mathsf{power}^s_{\mathsf{h}_1}} \xrightarrow{\mathsf{e}_1}$, therefore, the state of the last fetched instruction in thread tid in $s_{\mathsf{power}^s_{\mathsf{h}_1}}$ is q_1. By CMPL-D, $\mathsf{ctrl\text{-}state}(\mathsf{tid}) = q_1$ too.

Second, we choose $\mathsf{h}_2 := \max\{\mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid}, r) \mid r \text{ is read in } \mathsf{cmd}\}$. It satisfies the requirements from MH-ASSIGN. Note that $\mathsf{h}_2 \leq \mathsf{h}_3$ by CMPL-F' and POW-COMMIT: an instruction cannot be committed, until its arguments are computed.

Third, we must show that for each register r read by the instruction holds $\mathsf{h}_3 \geq \mathsf{reg\text{-}comm\text{-}head}(\mathsf{tid}, r)$ and $\mathsf{h}_3 \geq \mathsf{assume\text{-}comm\text{-}head}(\mathsf{tid})$. This holds by CMPL-G', CMPL-K', and POW-COMMIT: an instruction cannot be committed until its data and control dependencies are committed.

In the destination state, CMPL-F holds by CMPL-F in the source state, definition of $\mathsf{reg\text{-}value}'$ in MH-ASSIGN and definitions of eval. The other inductive statements trivially hold.

Assume Assume $\mathsf{e}_1 \sqsubseteq \overline{\tau^s_{\mathsf{h}_1}}$, $\mathsf{e}_3 \sqsubseteq \overline{\tau^s_{\mathsf{h}_3}}$, where $\mathsf{h}_1 < \mathsf{h}_3$, $\mathsf{e}_1 = (\mathsf{fetch}, \mathsf{tid}, q_1 \xrightarrow{\mathsf{cmd}} q_2)$, $\mathsf{e}_3 = (\mathsf{commit}, \mathsf{tid}, i)$, where $i = \mathsf{instr\text{-}count}(\mathsf{tid})$, $\mathsf{h}_1 = 1$, $i = \mathsf{instr\text{-}count}(\mathsf{tid})$, $\mathsf{cmd} = \texttt{assume}(e_{\mathsf{v}})$. Then, an MH-ASSUME transition is feasible.

The proof is similar to the proof for the case of assignment. The MH-ASSUME transition additionally requires $\mathsf{eval}(\mathsf{tid}, e_{\mathsf{v}}) \neq 0$. This holds by CMPL-F, definition of $\mathsf{reg\text{-}value}'$ in MH-ASSIGN and definitions of eval.

The inductive statements trivially hold in the destination state.

Load Assume $\mathsf{e}_1 \sqsubseteq \overline{\tau^s_{\mathsf{h}_1}}$, $\mathsf{e}_2 \sqsubseteq \overline{\tau^s_{\mathsf{h}_2}}$, $\mathsf{e}_3 \sqsubseteq \overline{\tau^s_{\mathsf{h}_3}}$, where $\mathsf{h}_1 < \mathsf{h}_2 < \mathsf{h}_3$, $\mathsf{e}_1 = (\mathsf{fetch}, \mathsf{tid}, q_1 \xrightarrow{\mathsf{cmd}} q_2)$, $\mathsf{e}_2 = (\mathsf{load}, \mathsf{tid}, i, \mathsf{a})$, $\mathsf{e}_3 = (\mathsf{commit}, \mathsf{tid}, i)$, $i = \mathsf{instr\text{-}count}(\mathsf{tid})$, $\mathsf{cmd} = r \leftarrow \texttt{mem}[e_{\mathsf{v}}]$. We show that an MH-LOAD transition is feasible. We point out only the differences with respect to the proof for the assignment case.

Assume e_2 was produced by a POW-EARLY transition. This means, the last store writing to a has its address known and is not committed yet in $s_{\mathsf{power}^s_{\mathsf{h}_2}}$. Then, by CMPL-N, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2) \in \mathsf{DOM}$, and we have $\mathsf{v} := \mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$. Assume e_2 was produced by a POW-LOAD transition. Then, POW-EARLY transition was not possible (Lemma 4.5, Lemma 4.6). This means, there was no in-flight stores to a in $s_{\mathsf{power}^s_{\mathsf{h}_2}}$. Then, by CMPL-N, $\mathsf{early\text{-}mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2) = \bot$, and we have $\mathsf{v} := \mathsf{mem\text{-}value}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_2)$. In both cases, by CMPL-N, CMPL-L we have $\mathsf{reg\text{-}value}'$ and $\mathsf{reg\text{-}comp\text{-}head}'$ satisfying CMPL-F and CMPL-F'. Additionally, we must show that $\mathsf{h}_3 \geq \mathsf{addr\text{-}comm\text{-}head}(\mathsf{tid}, \mathsf{a})$. This holds by CMPL-P' and CMPL-N. The requirement on the coherence key holds due to POW-FIN-LD.

Store Assume $u \in \mathbb{N}$, $\mathsf{e}_j \sqsubseteq \overline{\tau^s_{\mathsf{h}_j}}$ for $j \in [1..u+3]$, where $\mathsf{h}_2 = \mathsf{h}_3$, $\mathsf{e}_1 = (\mathsf{fetch}, \mathsf{tid}, q_1 \xrightarrow{\mathsf{cmd}} q_2)$, $\mathsf{e}_2 = (\mathsf{commit}, \mathsf{tid}, i, \mathsf{k}, \mathsf{a})$, $\mathsf{e}_3 = (\mathsf{prop}, \mathsf{tid}, \mathsf{tid}, i, \mathsf{a})$,

$\mathsf{e}_j = (\mathsf{prop}, \mathsf{tid}_j, \mathsf{tid}, i, \mathsf{a})$ for $j \in [4..u+3]$, $i = \mathsf{instr\text{-}count}(\mathsf{tid})$, $\mathsf{cmd} = \texttt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}$. Assume that there are no other prop events for (tid, i) in τ, except for $\mathsf{e}_3 \dots \mathsf{e}_{u+3}$. We show that an MH-STORE transition is feasible.

The requirements to be checked are similar to those in the load case. The requirement that k is not already used holds by CMPL-M and the fact that the same requirement in POW-STORE is met.

Consider the requirements in MH-STORE for generating prop events. The requirement that propagation event to thread tid is generated in the same part as commit is met by assumption $\mathsf{h}_3 = \mathsf{h}_2$. The requirement $\mathsf{last\text{-}key}(\mathsf{tid}', \mathsf{a}, \mathsf{h}) < \mathsf{k} \leq \mathsf{last\text{-}key}_g(\mathsf{tid}', \mathsf{a}, \mathsf{h}+1)$ is met by CMPL-L, choice of $\mathsf{last\text{-}key}_g$ in the initial transition, and POW-PROP-STORE.

This means, inductive invariant CMPL-A holds for $s+1$. Also, CMPL-B holds by choice of $\mathsf{e}_1 \dots \mathsf{e}_{u+3}$, CMPL-D holds trivially. CMPL-C holds by assumption that there are no other prop events in τ, except for $\mathsf{e}_3 \dots \mathsf{e}_{u+3}$. CMPL-F, CMPL-F', CMPL-G, CMPL-G' hold, since store instructions do not affect register values. CMPL-K, CMPL-K' hold, because a store instruction is not `assume()`. CMPL-L holds by definition of $\mathsf{mem\text{-}value}'$ in MH-STORE. CMPL-M holds by definition of $\mathsf{last\text{-}key}'$ in MH-STORE. CMPL-N holds by definition of $\mathsf{early\text{-}mem\text{-}value}'$ in MH-STORE. CMPL-P, CMPL-P' hold by definition of $\mathsf{addr\text{-}comm\text{-}head}'$ in MH-STORE. CMPL-R hold by definition of $\mathsf{instr\text{-}count}'$ in MH-STORE.

Now we must show that one of the cases above always takes place. Consider the event $\mathsf{e} = \mathsf{first}(\overline{\tau_1^s})$. By CMPL-C and the fact that $\tau \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$, it is a fetch event $(\mathsf{fetch}, \mathsf{tid}, i, \mathsf{instr})$. Choose the case based on the kind of instr. By POW-NF-A and POW-NF-B, all events belonging to the instruction (tid, i) constitute prefixes of $\overline{\tau_\mathsf{h}^s}$, $\mathsf{h} \in \mathsf{HEAD}$. The requirement $i = \mathsf{instr\text{-}count}(\mathsf{tid})$ holds by CMPL-R. The requirements like $\mathsf{h}_1 \leq \mathsf{h}_2 \leq \mathsf{h}_3$ in the load case naturally follow from the fact that $\tau \in \mathsf{C}_{\mathsf{power}}(\mathcal{P})$.

Assume $\tau_\mathsf{h}^s = \tau_\mathsf{h}$ for all $\mathsf{h} \in \mathsf{HEAD}$. Then $\tau_\mathsf{h}^s \in F_M$ by choice of $\mathsf{mem\text{-}value}_g$ and $\mathsf{last\text{-}key}_g$ in s_{M1} and CMPL-L, CMPL-M. □

Lemma 4.20. $\{\tau \in \mathsf{C}_{power}(\mathcal{P}) \mid \tau \textit{ is in normal form of degree } n\} \subseteq \mathcal{L}(M(\mathcal{P})) \subseteq \mathsf{C}_{power}(\mathcal{P})$.

Proof. This is a corollary of Lemmas 4.18 and 4.19. □

4.4.2 Checking Cyclicity of the Happens-Before Relation

As shown in Lemma 3.7, if a computation has a happens-before cycle, it has a *beautiful* happens-before cycle, in which each thread contributes only once.

Example 4.21. The happens-before cycle shown in Figure 4.1 is beautiful.

In this section we show how to detect beautiful cycles using finite automata.

Given a cycle profile θ, we define the automaton $M'(\mathcal{P}, \theta)$ as a modification of $M(\mathcal{P})$ that marks one event in each thread $\mathsf{tid}_j \in \theta$ by enter (identifying $(\mathsf{tid}_j, i_j, *)$) and a later (or the same) event by leave (identifying $(\mathsf{tid}_j, i'_j, *)$, $i_j \leq i'_j$). Note that $M(\mathcal{P})$ generates the events in program order, which ensures $(\mathsf{tid}_j, i_j, *) \rightarrow_{po}^* (\mathsf{tid}_j, i'_j, *)$. Technically, $M'(\mathcal{P}, \theta)$ introduces the following changes:

- The alphabet is $\mathrm{E}' := \mathrm{E} \times 2^{\{\mathsf{enter},\mathsf{leave}\}}$ with index components left out from the events.
- The automaton generates only load (although, for loads from memory only) and prop events, as only they are relevant for cycle detection.
- The prop events include k component of the corresponding commit event.

To check $(\mathsf{tid}_j, i'_j, *) \rightarrow_{hop} (\mathsf{tid}_{j+1}, i_{j+1}, *)$, we use an intersection with a regular language $H^{\mathsf{tid}_j,\mathsf{tid}_{j+1}}$. The language $H^{\mathsf{tid}_1,\mathsf{tid}_2}$ includes a computation τ iff one or more of the following conditions hold:

H-ST $(\mathsf{e}_1, m_1), (\mathsf{e}_2, m_2) \in \tau$, $\mathsf{leave} \in m_1$, $\mathsf{enter} \in m_2$, $\mathsf{e}_1 = (\mathsf{prop}, \mathsf{tid}_1, \mathsf{tid}_1, \mathsf{k}_1, \mathsf{a})$, $\mathsf{e}_2 = (\mathsf{prop}, \mathsf{tid}_2, \mathsf{tid}_2, \mathsf{k}_2, \mathsf{a})$, and $\mathsf{k}_1 < \mathsf{k}_2$.

H-SRC $\tau = \tau_1 \cdot (\mathsf{e}_1, m_1) \cdot \tau_2 \cdot (\mathsf{e}_2, m_2) \cdot \tau_3$, $\mathsf{leave} \in m_1$, $\mathsf{enter} \in m_2$, $\mathsf{e}_1 = (\mathsf{prop}, \mathsf{tid}_2, \mathsf{tid}_1, \mathsf{a})$, $\mathsf{e}_2 = (\mathsf{load}, \mathsf{tid}_2, \mathsf{a})$, τ_2 does not contain events $(\mathsf{prop}, \mathsf{tid}_2, *, \mathsf{a})$.

H-CF1 $\tau = \tau_1 \cdot (\mathsf{e}_3, m_3) \cdot \tau_2 \cdot (\mathsf{e}_2, m_2) \cdot \tau_3$, $\mathsf{leave} \in m_2$, $\mathsf{e}_3 = (\mathsf{prop}, \mathsf{tid}_1, \mathsf{tid}_3, \mathsf{k}_3, \mathsf{a})$, $\mathsf{e}_2 = (\mathsf{load}, \mathsf{tid}_1, \mathsf{a})$, τ_2 does not contain events $(\mathsf{prop}, \mathsf{tid}_1, *, \mathsf{a})$, $(\mathsf{e}_3, m_3) \in \tau_1 \cdot \tau_2 \cdot \tau_3$, $m_3 \in \mathsf{enter}$, $\mathsf{e}_3 = (\mathsf{prop}, \mathsf{tid}_2, \mathsf{tid}_2, \mathsf{k}_2)$, $\mathsf{k}_3 < \mathsf{k}_2$.

H-CF2 $(\mathsf{e}_1, m_1), (\mathsf{e}_2, m_2) \in \tau$, $\mathsf{leave} \in m_1$, $\mathsf{enter} \in m_2$, $\mathsf{e}_1 = (\mathsf{load}, \mathsf{tid}_1, \mathsf{a})$, $\mathsf{e}_2 = (\mathsf{prop}, \mathsf{tid}_2, \mathsf{tid}_2, \mathsf{k}_2, \mathsf{a})$ and there is no $(\mathsf{e}_3, m_3) \in \tau$ with $\mathsf{e}_3 = (\mathsf{prop}, \mathsf{tid}_1, \mathsf{tid}_3, \mathsf{k}_3, \mathsf{a})$ in τ before (e_1, m_1).

Since finite automata are closed under intersection, we can define the *finite automaton for cycle profile* $\theta = \mathit{tid}_1 \ldots \mathit{tid}_k$ as

$$H^\theta := H^{\mathsf{tid}_1,\mathsf{tid}_2} \cap \ldots \cap H^{\mathsf{tid}_{k-1},\mathsf{tid}_k} \cap H^{\mathsf{tid}_k,\mathsf{tid}_1}.$$

Lemma 4.22. *Program $\mathcal{P}$ has a beautiful cycle with profile θ iff*

$$M'(\mathcal{P}, \theta) \cap H^\theta \neq \emptyset.$$

Note that $M'(\mathcal{P}, \theta)$ is infinite-state. To ensure $M'(\mathcal{P}, \theta)$ has finitely many states, we note that the instruction indices are irrelevant for the detection of happens-before cycles (instr-count can be dropped), and that the number of different coherence keys that must be stored in the state at any moment is polynomial in the size of $\mathcal{P}$. Indeed, the last-key, last-key$_g$, and early-mem-key components of the state each store at most $|\mathsf{ADDR}| \cdot |\mathcal{P}| \cdot n$ different coherence keys. Each modification of the last-key component of the state can be extended by a normalization step that would turn coherence keys to consecutive natural numbers starting from zero. The normalization step must preserve the less-than relation on the keys. In order for the detection of happens-before cycles to work correctly, the automaton has to remember the coherence keys of marked store events: they must be preserved during normalization. Altogether, this results into $O(|\mathsf{ADDR}| \cdot |\mathcal{P}|^2 \cdot n)$ different keys, which is polynomial in the size of $\mathcal{P}$.

Theorem 4.23. *Robustness against Power for programs over finite domains is* PSPACE*-complete.*

Proof. By Theorem 4.9, Lemma 3.7, and Lemma 4.22, a program is non-robust iff the equation from Lemma 4.22 holds for some θ. In order to check robustness, we enumerate all profiles θ and check the equation from Lemma 4.22. The enumeration can be done in PSPACE. By construction and Lemma 3.3, the size of the intersection automaton is exponential in the size of the program. By Lemma 3.4, language emptiness for it can be checked in PSPACE in the size of the program, which gives us the upper bound.

The PSPACE lower bound follows from PSPACE-hardness of SC state reachability. One can reduce reachability to robustness by inserting an artificial happens-before cycle in the target state. □

4.4.3 Handling Memory Barriers

We now come back to the Power barrier instructions and show how to support them in the reduction to language emptiness.

Clearly, barrier instructions do not produce inter-thread happens-before dependencies. Therefore, the $H^{\mathsf{tid}_1,\mathsf{tid}_2}$ automaton checking arcs between the threads remains the same. We describe how to add support for barriers in the multiheaded automaton generating normal-form computations.

First, we extend the automaton state with the following information (note that its size is polynomial in the size of the program):

- $\mathsf{store\text{-}comm\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which a store of thread tid was committed.
- $\mathsf{load\text{-}comm\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which a load of thread tid was committed.
- $\mathsf{sync\text{-}comm\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which a `sync` of thread tid was committed.
- $\mathsf{lwsync\text{-}comm\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which an `lwsync` of thread tid was committed.
- $\mathsf{isync\text{-}comm\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which an `isync` of thread tid was committed.
- $\mathsf{sync\text{-}acked\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which a `sync` of thread tid was acknowledged.
- $\mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, \mathsf{h}, \mathsf{tid}')$ gives the rightmost part of the computation in which all the `sync` and `lwsync` barriers that were propagated to thread tid in the part h or earlier are propagated to thread tid'.
- $\mathsf{barrier\text{-}prop\text{-}head}_g$ is an immutable copy of the previous component of the state.
- $\mathsf{load\text{-}before\text{-}lwsync\text{-}head}(\mathsf{tid})$ gives the rightmost part of the computation in which a load of thread tid followed by `lwsync` was committed.

Second, we update the MH-LOAD rule to require $\mathsf{h}_2 \geq \mathsf{isync\text{-}comm\text{-}head}(\mathsf{tid})$, and $\mathsf{h}_2 \geq \mathsf{sync\text{-}acked\text{-}head}(\mathsf{tid})$, to make sure that a load happens only after all previous `isync`s and `lwsync`s are committed and `sync`s are acknowledged. We

require $\mathsf{h}_2 \geq \mathsf{load\text{-}before\text{-}lwsync\text{-}head}(\mathsf{tid})$ to make sure that the load happens after all preceding loads followed by an `lwsync` are committed. Also, committing a load requires all previous `sync`, `lwsync`, `isync` instructions to be committed and `sync`s to be acknowledged: $\mathsf{h}_3 \geq \mathsf{sync\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{lwsync\text{-}comm\text{-}head}$, $\mathsf{h}_3 \geq \mathsf{isync\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{sync\text{-}acked\text{-}head}(\mathsf{tid})$. Finally, we update the rightmost part where a load was committed: $\mathsf{load\text{-}comm\text{-}head}' := \mathsf{load\text{-}comm\text{-}head}[\mathsf{tid} := \max\{\mathsf{load\text{-}comm\text{-}head}(\mathsf{tid}), \mathsf{h}_3\}]$.

Third, we update the MH-STORE rule. Committing a store requires all previous `sync`, `lwsync`, `isync` instructions to be committed and `sync`s to be acknowledged: $\mathsf{h}_3 \geq \mathsf{sync\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{lwsync\text{-}comm\text{-}head}$, $\mathsf{h}_3 \geq \mathsf{isync\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{sync\text{-}acked\text{-}head}(\mathsf{tid})$. Also, a store can be propagated to a thread tid' in part h only if the group-A `sync`s and `lwsync`s are propagated to that thread: $\mathsf{h} \geq \mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, \mathsf{h}_3, \mathsf{tid}')$. Finally, we update the rightmost part where a store was committed: $\mathsf{store\text{-}comm\text{-}head}' := \mathsf{store\text{-}comm\text{-}head}[\mathsf{tid} := \max\{\mathsf{store\text{-}comm\text{-}head}(\mathsf{tid}), \mathsf{h}_3\}]$.

Fourth, we add a rule for `isync`. The rule is defined as expected, with the only special requirement that `isync` must be committed only after the addresses of all program-order-earlier loads and stores are computed: $\mathsf{h}_3 \geq \mathsf{addr\text{-}comp\text{-}head}(\mathsf{tid})$. The rule updates the information about the rightmost part where an `isync` was committed: $\mathsf{isync\text{-}comm\text{-}head}' := \mathsf{isync\text{-}comm\text{-}head}[\mathsf{tid} := \max\{\mathsf{isync\text{-}comm\text{-}head}(\mathsf{tid}), \mathsf{h}_3\}]$.

Fifth, we add rules for `sync` and `lwsync`. Committing a `sync` or `lwsync` requires all previous loads, stores, `sync`, `lwsync`, and `isync` instructions to be committed: we require $\mathsf{h}_3 \geq \mathsf{load\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{store\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{sync\text{-}comm\text{-}head}(\mathsf{tid})$, $\mathsf{h}_3 \geq \mathsf{lwsync\text{-}comm\text{-}head}(\mathsf{tid})$. We update the rightmost part where a `sync` (`lwsync`) was committed: $\mathsf{sync\text{-}comm\text{-}head}' := \mathsf{sync\text{-}comm\text{-}head}[\mathsf{tid} := \max\{\mathsf{sync\text{-}comm\text{-}head}(\mathsf{tid}), \mathsf{h}_3\}]$ ($\mathsf{lwsync\text{-}comm\text{-}head}' := \mathsf{lwsync\text{-}comm\text{-}head}[\mathsf{tid} := \max\{\mathsf{lwsync\text{-}comm\text{-}head}(\mathsf{tid}), \mathsf{h}_3\}]$). The rules allow to propagate a barrier to a thread tid' in part h if $\mathsf{last\text{-}key}(\mathsf{tid}', \mathsf{a}, \mathsf{h}) \geq \mathsf{last\text{-}key}(\mathsf{tid}, \mathsf{a}, \mathsf{h}_3)$ for all $\mathsf{a} \in \mathsf{ADDR}$. Once the parts of the computation where the barrier is propagated to each thread is chosen, we update $\mathsf{barrier\text{-}prop\text{-}head}$ accordingly. Technically, the memory model does not require propagation of the barriers to *all* the threads. We can use a special value ∞ in $\mathsf{barrier\text{-}prop\text{-}head}$ to encode the fact that a barrier is never propagated to a thread. If the instruction is a `sync`, we update $\mathsf{sync\text{-}acked\text{-}head}$: assume the rightmost propagation is done in part h_4 ($\mathsf{h}_4 := \infty$ if the `sync` was not propagated to all the threads), then $\mathsf{sync\text{-}acked\text{-}head}' := \mathsf{sync\text{-}acked\text{-}head}[\mathsf{tid} := \max\{\mathsf{sync\text{-}acked\text{-}head}(\mathsf{tid}), \mathsf{h}_4\}]$. If the instruction is `lwsync`, we update the part in which a load fetched before `lwsync` was committed: $\mathsf{load\text{-}before\text{-}lwsync\text{-}head}' := \mathsf{load\text{-}before\text{-}lwsync\text{-}head}[\mathsf{tid} := \mathsf{load\text{-}comm\text{-}head}(\mathsf{tid})]$.

Sixth, we update the MH-GUESS rule. Namely, $\mathsf{store\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{load\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{sync\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{lwsync\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{isync\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{sync\text{-}acked\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, 1, \mathsf{tid}') := 1$, $\mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, \mathsf{h}, \mathsf{tid}') \leq \mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, \mathsf{h}+1, \mathsf{tid}')$ for all $\mathsf{tid} \in \mathsf{TID}$, $\mathsf{h} \in \mathsf{HEAD}$, $\mathsf{tid}' \in \mathsf{TID}$, $\mathsf{barrier\text{-}prop\text{-}head}_g := \mathsf{barrier\text{-}prop\text{-}head}$, $\mathsf{load\text{-}before\text{-}lwsync\text{-}head}' := \lambda\mathsf{tid}.1$, $\mathsf{load\text{-}comm\text{-}head}' := \lambda\mathsf{tid}.1$.

Finally, we add a new condition on the final states: $\mathsf{barrier\text{-}prop\text{-}head}(\mathsf{tid}, \mathsf{h}, \mathsf{tid}') = \mathsf{barrier\text{-}prop\text{-}head}_g(\mathsf{tid}, \mathsf{h}+1, \mathsf{tid}')$.

4.5 Reachability under Power

In this chapter we study the state reachability problem for Power, as defined in Section 2.5. Under the current control state of a thread we understand the destination state of the last instruction fetched in the thread.

Theorem 4.24. *State reachability under Power for single-threaded finite-state programs is* PSPACE*-complete.*

Proof. Follows from the fact that state reachability under SC is PSPACE-complete (Lemma 2.7), and the fact that Power creates an illusion of sequential consistency for single-threaded programs. First, by definition of eval, an instruction always reads the value of a register that is assigned by the latest preceding instruction writing to this register. Second, by POW-FIN-LD-ST, a load instruction always reads last value written by the latest preceding store to the same address. □

Theorem 4.25 ([16])**.** *State reachability under Power for finite-state programs consisting of two or more threads is undecidable.*

Proof. We show how to implement a perfect channel by exploiting the fact that the number of instructions that can be concurrently executed in a thread is unbounded. Since a finite automaton equipped with a perfect FIFO channel is as powerful as a Turing machine, we immediately get undecidability. The implementation $\mathcal{P}$ of a perfect channel machine consists of two threads: $\mathcal{T}_{\mathsf{main}}$ is the thread implementing a FIFO channel machine using send and receive operations, $\mathcal{T}_{\mathsf{aux}}$ is the thread effectively implementing the perfect infinite FIFO channel.

We implement the channel using two variables x and y initially having special value $\perp$ which is never transmitted over the channel. Let the value to be sent through the channel be stored in register r_{data}. We implement the send operation in $\mathcal{T}_{\mathsf{main}}$ as follows:

$$q_1 \xrightarrow{r \leftarrow \mathtt{mem}[x]} q_2 \xrightarrow{\mathtt{assume}(r=\perp)} q_3 \xrightarrow{\mathtt{mem}[x] \leftarrow r_{\mathsf{data}}} q_4.$$

This implementation blocks if sending fails.

Assume the value to be received from the channel must be written to register r_{data}. Then we implement the receive operation in $\mathcal{T}_{\mathsf{main}}$ as follows:

$$q_1 \xrightarrow{r_{\mathsf{data}} \leftarrow \mathtt{mem}[y]} q_2 \xrightarrow{\mathtt{assume}(r_{\mathsf{data}} \neq \perp)} q_3 \xrightarrow{\mathtt{mem}[y] \leftarrow \perp} q_4.$$

Similarly, the implementation blocks when the operation fails.

Finally, we define the auxiliary thread $\mathcal{T}_{\mathsf{aux}} := (Q_{\mathsf{aux}}, \mathsf{CMD}, \mathcal{I}_{\mathsf{aux}}, q_{\mathsf{aux}0}, Q_{\mathsf{aux}f})$ copying x into y. The set of all control states is $Q_{\mathsf{aux}} := \{q_k \mid k \in [0..6]\}$. The initial state is $q_{\mathsf{aux}0} := q_0$. The set of final states is $Q_{\mathsf{aux}f} := Q_{\mathsf{aux}}$. The transition relation $\mathcal{I}_{\mathsf{aux}}$ consists of the following instructions:

$$q_0 \xrightarrow{r_{\mathsf{mask}} \leftarrow \top} q_1 \xrightarrow{r \leftarrow \mathtt{mem}[x]} q_2 \xrightarrow{r_{\mathsf{mask}} \leftarrow r_{\mathsf{mask}} \wedge (r \neq \perp)} q_3 \xrightarrow{\mathtt{mem}[x] \leftarrow \perp} q_4,$$

$$q_4 \xrightarrow{r' \leftarrow \mathtt{mem}[y]} q_5 \xrightarrow{r_{\mathsf{mask}} \leftarrow r_{\mathsf{mask}} \wedge (r' = \perp)} q_6 \xrightarrow{\mathtt{mem}[y] \leftarrow r \wedge r_{\mathsf{mask}}} q_1.$$

Here, one assumes $a \wedge \top \equiv a$, $a \wedge \perp \equiv \perp$, and comparisons returning $\perp$ (false) and $\top$ (true).

The idea of the construction is as follows. The send operation checks if $\mathcal{T}_{\mathsf{aux}}$ has already processed the previously sent value (variable x contains $\bot$). Only in this case the new value is written into x. The receive operation does the reverse: it reads the value from y, checks that this value is not $\bot$ (i.e., was written by $\mathcal{T}_{\mathsf{aux}}$), and writes $\bot$ to y to signal $\mathcal{T}_{\mathsf{aux}}$ that a new value can be put there.

The thread $\mathcal{T}_{\mathsf{aux}}$ executes an infinite loop reading values from x and writing them to y. The thread uses register r_{mask} for remembering whether reading or writing a value did previously fail. After reading a value from x it checks that this value is not $\bot$, i.e., some value was actually sent. If this is not the case, r_{mask} becomes $\bot$. Next, the thread writes $\bot$ to x, thus signalling that a new value can be sent. After that, the thread checks that y contains $\bot$ (i.e., the previously written value was received). If not, again r_{mask} is set to $\bot$. Finally, the thread writes either the value that was read (if $r_{\mathsf{mask}} = \top$) or $\bot$ (if $r_{\mathsf{mask}} = \bot$) to y. Accordingly, all subsequent receive operations will fail if $\mathcal{T}_{\mathsf{aux}}$ at least once detected that x does not contain a value to be sent or y contains the previously copied value.

Note that the sequence of values loaded from x will be the same as the sequence of values written to y, as Power forbids reordering of load operations from the same address and store operations to the same address. This guarantees that the channel is a FIFO channel. Moreover, the loads from y and stores to y can be delayed arbitrary long by the thread, which makes the delay between reading a new value from x and writing it back to y arbitrary large. This is the source of infiniteness of the channel. Finally, a value cannot be sent, until it was read by $\mathcal{T}_{\mathsf{aux}}$; also, a value cannot be written back by $\mathcal{T}_{\mathsf{aux}}$ until the previous value was received by another thread. This makes the channel lossless, i.e., perfect.

Depending on the scheduling, this channel implementation may spuriously fail (which is detected, after which the subsequent receive operations on the channel block). However, there is always a schedule in which no operation fails (except when one attempts to receive from an empty channel): each send operation in $\mathcal{T}_{\mathsf{main}}$ is immediately followed by $q_1 \dots q_4$ instructions of $\mathcal{T}_{\mathsf{aux}}$ being executed and each receive operation in $\mathcal{T}_{\mathsf{main}}$ is always preceded by $q_4 \dots q_1$ instructions of $\mathcal{T}_{\mathsf{aux}}$ being executed and the store to y from $\mathcal{T}_{\mathsf{aux}}$ propagated to all the threads. □

Chapter 5

Robustness against SPARC Memory Models

The SPARC Architecture Manual [84] defines three memory models: Relaxed Memory Order (RMO), Partial Store Order (PSO), and Total Store Order (TSO). The first, RMO, is the most relaxed of the three. PSO is defined as a restriction of RMO, and TSO is defined as a restriction of PSO.

The current memory model used by a processor is determined by the values of two bits `PSTATE.MM` in the processor's state register `PSTATE`. Clearly, a valid CPU implementation may implement any memory model, as long as it is no more relaxed than the requested one, and recent SPARC CPUs seem to implement solely TSO [67].

In this chapter we give an overview of existing formalizations of the models from the SPARC hierarchy and discuss the application of our robustness framework to these models.

5.1 Relaxed Memory Order

The SPARC Architecture Manual [84] provides an axiomatic definition of RMO. Alglave [5] formalizes this definition in terms of acyclicity of a happens-before relation.

According to the definitions, RMO is similar, however, incomparable to Power. First, RMO is store-atomic: a store, once executed on memory, becomes immediately visible to all the threads. In this part, it is stronger than Power. Second, RMO allows reordering of loads from the same address: a program-order-earlier load instruction can read a value that was written after the value read by a program-order-later load instruction accessing the same address. In this part, it is weaker than Power.

Ordering of independent memory operations on SPARC can be enforced using the `membar` instruction. The instruction takes as an argument a bit mask, saying which types of operations must be ordered. The available bits include `#LoadLoad`, `#StoreLoad`, `#LoadStore`, `#StoreStore`. For example, `membar #LoadStore | #StoreStore` makes sure that all the loads and stores issued before `membar` are executed on memory before the stores issued after `membar`.

Following the above discussion, one can construct a formal operational semantics for RMO by modifying the Power semantics from Section 4.1. In POW-STORE rule one must require that a commit of a store is immediately followed by POW-PROP-STORE transitions propagating the committed store to *all* the threads. This ensures store atomicity. Excluding POW-FIN-LD from the set of requirements on final states allows out-of-order loads from the same address. Next, one relaxes requirements in POW-COMMIT to allow loads to be committed even before program-order-earlier loads to the same (or unknown) address are committed. Finally, POW-COMMIT rule must be extended to check that the dependencies introduced by `membar` instructions are honored.

Being quite close to Power, RMO inherits its complexity and (un)decidability results.

Theorem 5.1. *Robustness against RMO for programs with finite data domain is* PSPACE*-complete.*

Proof. The proofs in Section 4.3 do not rely on POW-FIN-LD or the fact that stores may be propagated not immediately after commit. Therefore, Theorem 4.9 holds for RMO as well. Actually, due to store atomicity, the normal form for RMO is even more strict: the number of parts τ_i with $\tau_i \neq \varepsilon$ does not exceed three.

Next, we apply the reduction of robustness to the emptiness problem for multiheaded automata, similar to Section 4.4. The multiheaded automaton construction requires appropriate relaxations in MH-LOAD rule: the check that the coherence key of the store loaded by the current load is not less than the coherence key of the store read by the program-order-previous load from the same address is not needed. Moreover, loads from the same address can be committed out of program order on RMO, so, instead of tracking addr-comm-head, the automaton must remember the part in which the last store to the given address is committed in a given thread and the part in which the last load in the given thread is committed, and in MH-LOAD check the ordering only with respect to stores, whereas in MH-STORE check the ordering with respect to both loads and stores. MH-STORE rule is enhanced to generate all prop events immediately after commit event.

The dependencies introduced by `membar` instruction can be handled using finite amount of information as well. Indeed, one only needs to track, in which part of the computation the last load (store) in each thread was done (committed), and in which part of the computation the next load (store) can be done (committed). Once, e.g., a `membar #LoadStore | #StoreStore` is handled, the multiheaded automaton must remember that the next store instruction in the current thread can be committed no earlier than in the part where the last load of the same thread was done, and no earlier than the part in which the last store of the same thread is committed.

□

Theorem 5.2. *State reachability under RMO for single-threaded finite-state programs is* PSPACE*-complete.*

Proof. Similar to the proof of Theorem 4.24. □

Theorem 5.3. *State reachability under RMO for finite-state programs consisting of two or more threads is undecidable.*

Proof. By a reduction of state reachability for perfect channel machine to state reachability under RMO, similar to the proof of Theorem 4.25. The ordering of load operations from the same address, which is guaranteed under Power, but not under RMO, has to be enforced, e.g., by introducing artificial address dependencies through registers between loads. □

5.2 Partial Store Order

The SPARC Architecture Manual [84] defines PSO as RMO, where a load is implicitly followed by a `membar #LoadLoad | #LoadStore` instruction. Axiomatic definitions [5, 64, 59] of PSO allow reordering of stores to different addresses and delaying of stores past loads. Operational definitions [14, 26, 15] emulate these reorderings by means of per-thread and per-address store buffers.

The SPARC definition of PSO allows to reduce robustness against PSO to robustness against RMO for a program instrumented with `membar` instructions. This immediately leads to the following complexity result.

Theorem 5.4. *Robustness against PSO for programs with finite data domain is* PSPACE*-complete.*

5.3 Total Store Order

SPARC [84] defines TSO as PSO, where each store is implicitly followed by a `membar #StoreStore` instruction. Therefore, one can reduce robustness against TSO to robustness against PSO, and get the following complexity result.

Theorem 5.5. *Robustness against TSO for programs with finite data domain is* PSPACE*-complete.*

In the next chapter we consider TSO in more detail and show that the normal form for TSO can be restricted even further: it is sufficient to look only for computations where a single thread does reorderings. This fact will allow us to obtain practical algorithms for checking and enforcing robustness against this memory model.

Chapter 6

Robustness against Total Store Order

In this chapter we study the problem of checking and enforcing robustness against TSO, a memory model used in Intel x86 and Sun SPARC architectures [72, 84]. Intuitively, the model reflects the use of store buffers: stores performed by a thread are enqueued into this thread's store buffer and executed on memory later in FIFO order. A formal definition of the model is given in Section 2.4.2.

Robustness against TSO was first addressed by Burckhardt and Musuvathi in [25]. They proposed a monitoring algorithm that could detect non-robustness by monitoring SC computations of a program. Burnim et al. [26] pointed out a mistake in the axiomatic definition of TSO used in [25] and proposed monitoring algorithms for TSO and PSO memory models. Alglave and Maranget [10] presented a tool that statically overapproximates the set of happens-before cycles in programs written in x86 and Power assembly and inserts memory barriers forbidding these cycles and ensuring robustness (called *stability* in their work). Alglave et al. [7] carry over the static overapproximation approach from [10] to C programs and use integer linear programming to compute a fence set of optimal cost eliminating all potential cycles.

Although the above procedures can detect non-robustness or modify a program to make it robust, they cannot be used to check robustness. The first decidability result for robustness against TSO was presented by Bouajjani et al. [21]. They showed that robustness is PSPACE-complete for finite-state programs with unbounded store buffers. Their algorithm for checking robustness is based on enumeration of SC computations of bounded length and is not very useful for practical purposes.

In this chapter we present practical algorithms for checking and enforcing robustness against TSO. The algorithms are based on a generic, source-to-source reduction of robustness to SC state reachability of an instrumented program. The instrumentation is linear in the size of the original program, assumes no bound on the size of the store buffers, and is applicable to programs with arbitrary data domain and unlimited number of threads. To obtain the reduction, we first characterize robustness in terms of the absence of *TSO witnesses* — TSO computations of a special form, where only one thread does reorderings.

Then, we show how to modify the original program to capture TSO witnesses by SC computations.

The reduction immediately leads to new decidability and complexity results: for finite-state programs robustness against TSO is PSPACE-complete for finite number of threads and decidable in the parameterized setting. Additionally, we describe an algorithm using the reduction and SC reachability queries as subprocedures to compute an optimal fence set ensuring robustness.

The results presented in this chapter are also published in [20].

Related work. Triangular-race freedom (TRF) is a correctness criterion proposed by Owens [71] which is stronger than robustness and weaker than data-race freedom. Therefore, TRF may require more fences than actually necessary for robustness.

State reachability under TSO was shown to be decidable, although non-primitive recursive-complete, by Atig et al. [14]. Several tools are capable of solving state reachability under relaxed memory models.

Memorax [2] implements a sound and complete decision procedure for TSO reachability combining automata-based abstraction of the set of feasible program computations with backward reachability analysis. The tool also implements a counterexample-guided fence insertion algorithm capable of computing minimal fence sets forcing the program to satisfy a given safety specification. A later version of the tool [1] introduced predicate abstraction to allow the analysis of infinite-state programs.

Remmex [59] performs forward reachability analysis using finite automata to represent the exact contents of store buffers. The algorithm is sound, however, does not guarantee termination. The tool supports TSO and PSO memory models.

CBMC [9] uses SMT-based bounded model checking and encodes memory model constraints into SMT formulae. Because of the underapproximate nature of bounded analysis, CBMC is sound, but not complete. The tool supports a wide range of memory models, including TSO and a fragment of Power.

CheckFence [24], similar to CBMC, uses bounded model checking and SAT queries to construct the set of all program computations under a relaxed memory model.

Other approaches to solving reachability under TSO include bounding the number of context switches [17], overapproximating buffer contents [54], bounding the size of the store buffers [8]. Automatic inference of memory fences for enforcing safety, robustness, and linearizability, using explicit state space exploration under the unmodified relaxed semantics, followed by a SAT query, is covered in [53, 62]. Finally, there is work on compiler optimizations that either add memory fences to ensure sequential consistency [81] or remove redundant memory fences to improve performance [83].

6.1 Locality and TSO Witnesses

We instantiate the robustness problem (Section 2.8) for TSO.

Problem 6.1 (Robustness against TSO)**.** Given a program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}) = T_{\mathsf{tso}}(\mathcal{P})$.

According to Lemma 2.16, checking robustness of a program amounts to checking whether the program has computations with cyclic happens-before relation. In this section we show that if a program has TSO computations with cyclic happens-before relation, among them there is a computation where only a single thread does reorderings. The result was originally proven by Bouajjani et al. [21] for programs with blocking load semantics: a load expects to read a particular value and blocks if the value in the memory is different. We show that this *locality* result holds for programs with the traditional semantics, where loads just copy the value from the memory to a register, without performing any additional checks.

Consider a computation $\tau = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ with two events a and b of the same thread $\mathsf{tid}(a) = \mathsf{tid}(b) = \mathsf{tid}$. We define the *distance* $d_\tau(a, b)$ *between* a *and* b *in* τ as the number of events in β that also belong to this thread: $d_\tau(a, b) := |\beta{\downarrow}\mathsf{tid}|$, where $\alpha{\downarrow}\mathsf{tid}$ is the longest subsequence of events e with $\mathsf{tid}(\mathsf{e}) = \mathsf{tid}$ of α. The *number of delays* $\#(\tau)$ in computation τ is the sum of distances between matching store and flush events:

$$\#(\tau) := \sum_{\text{matching } \mathsf{e}_1, \mathsf{e}_2 \in \tau} d_\tau(\mathsf{e}_1, \mathsf{e}_2).$$

We call a computation τ with cyclic happens-before relation a *minimal violation* if it is has a minimal number of delays among all computations with cyclic happens-before relation. Clearly, a program $\mathcal{P}$ has computations with cyclic happens-before relation iff it has a minimal violation.

The following lemma says that if in a minimal violation a store was delayed, then it was delayed past a load event of the same thread. Moreover, the load did not read the value from this store early.

Lemma 6.2. *Consider a minimal violation* $\tau = \alpha \cdot \mathsf{e}_1 \cdot \beta \cdot \mathsf{e}_2 \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$*, where* e_1 *and* e_2 *are the matching store and* flush *events with* $\mathsf{tid}(\mathsf{e}_1) = \mathsf{tid}(\mathsf{e}_2) = \mathsf{tid}$*. Then* $\beta{\downarrow}\,\mathsf{tid}$ *is either empty, or* $\beta{\downarrow}\,\mathsf{tid} = \beta' \cdot \mathsf{e}_3 \cdot \beta''$ *where* e_3 *is a load from memory (not an early read) event and* β'' *contains only* flush *events.*

Proof. Suppose β contains one or more events of thread tid. If all events of thread tid in β are flush events, then also $\tau' = \alpha \cdot \beta \cdot \mathsf{e}_1 \cdot \mathsf{e}_2 \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$. It has the same trace as τ but $\#(\tau') < \#(\tau)$, which contradicts the minimality of τ.

Otherwise let e_3 be the last non-flush event in $\beta{\downarrow}\mathsf{tid}$, i.e., $\beta = \beta_1 \cdot \mathsf{e}_3 \cdot \beta_2$ and all events in β_2 are flush events or belong to threads different from tid. Since flush events cannot be delayed past a memory fence of the same thread, e_3 is a store event, a local assignment, condition, or a load. In the former three cases, as well as if e_3 is an early read, delaying e_2 past e_3 can be avoided in the computation $\tau' = \alpha \cdot \mathsf{e}_1 \cdot \beta_1 \cdot \beta_2 \cdot \mathsf{e}_2 \cdot \mathsf{e}_3 \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$. The computation has the same trace as τ and $\#(\tau') < \#(\tau)$, which contradicts the minimality of τ. □

To avoid case distinction, in the remainder of the section we assume that a store event $\mathsf{e}_1 \in \tau$ and a matching flush event $\mathsf{e}_2 \in \tau$ event are related by happens-before: $\mathsf{e}_1 \rightarrow_{hb} \mathsf{e}_2$.

Consider $\tau = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$. We say that a *is happens-before* b *through* β if there is a (potentially empty) subsequence $\mathsf{e}_1 \dots \mathsf{e}_n$ of β, such that for all $i \in [0..n]$ holds $\mathsf{e}_i \rightarrow_{hb} \mathsf{e}_{i+1}$ or $\mathsf{e}_i \rightarrow^+_{po} \mathsf{e}_{i+1}$, assuming $\mathsf{e}_0 := a$ and $\mathsf{e}_{n+1} := b$.

The following lemma says that if two events in a minimal violation are not related via $\rightarrow_{hb}^{+}$, they can be reordered without changing the trace and the order of events within each thread.

Lemma 6.3 (Duality)**.** *Consider a minimal violation $\tau = \alpha \cdot \mathsf{e}_1 \cdot \beta \cdot \mathsf{e}_2 \cdot \gamma \in \mathsf{C}_{tso}(\mathcal{P})$. Then (1) $\mathsf{e}_1 \rightarrow_{hb}^{+} \mathsf{e}_2$ through β or (2) there is $\tau' = \alpha \cdot \beta_1 \cdot \mathsf{e}_2 \cdot \mathsf{e}_1 \cdot \beta_2 \cdot \gamma \in \mathsf{C}_{tso}(\mathcal{P})$, such that $T(\tau) = T(\tau')$ and $\tau{\downarrow}\mathsf{tid} = \tau'{\downarrow}\mathsf{tid}$ for all $\mathsf{tid} \in \mathsf{TID}$.*

Proof. We establish $\neg(1) \Rightarrow (2)$. Note that this proves the disjunction since $\neg(2) \Rightarrow (1)$ is the contrapositive. We proceed by induction on $|\beta|$ and slightly strengthen the hypothesis: we also show that β_2 is a subsequence of β.

Base case: $|\beta| = 0$**.** Then $\tau = \alpha \cdot \mathsf{e}_1 \cdot \mathsf{e}_2 \cdot \gamma$ and $\mathsf{e}_1 \not\rightarrow_{hb} \mathsf{e}_2$. If $\mathsf{tid}(\mathsf{e}_1) = \mathsf{tid}(\mathsf{e}_2)$, then $\mathsf{e}_2 \rightarrow_{po}^{+} \mathsf{e}_1$. Therefore, e_2 is a flush event which has been delayed past e_1. Swapping e_1 and e_2 will save the delay without changing the trace, in contradiction to the minimality of τ.

If $\mathsf{tid}(a) \neq \mathsf{tid}(b)$, then either at least one of the two events is not a memory access, the events access different addresses, or both are loads. In all the cases swapping them produces τ' as required in the statement of the lemma.

Step case. Assume the statement of the lemma holds for $|\beta| \leq n$. Consider $\tau' = \alpha \cdot \mathsf{e}_1 \cdot \beta \cdot \mathsf{e}_2 \cdot \gamma$ with $|\beta| = n + 1$. Let e_3 be the last event in $\beta = \beta' \cdot \mathsf{e}_3$. Since $\mathsf{e}_1 \not\rightarrow_{hb}^{+} \mathsf{e}_2$ through β, then $\mathsf{e}_1 \not\rightarrow_{hb}^{+} \mathsf{e}_3$ through β' or $\mathsf{e}_3 \not\rightarrow_{hb} \mathsf{e}_2$.

Consider the case when $\mathsf{e}_1 \not\rightarrow_{hb}^{+} \mathsf{e}_3$ through β'. We apply the induction hypothesis to τ with respect to e_1 and e_3. This gives $\tau' = \alpha \cdot \beta_1' \cdot \mathsf{e}_3 \cdot \mathsf{e}_1 \cdot \beta_2' \cdot \mathsf{e}_2 \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ with the same trace and thread computations as τ. Note that $\mathsf{e}_1 \not\rightarrow_{hb}^{+} \mathsf{e}_2$ through β_2' in τ', because $T(\tau) = T(\tau')$ and β_2' is a subsequence of β. Consequently, we can apply the induction hypothesis to τ' with respect to e_1 and e_2. This yields $\tau'' = \alpha \cdot \beta_1' \cdot \mathsf{e}_3 \cdot \beta_{21}' \cdot \mathsf{e}_2 \cdot \mathsf{e}_1 \cdot \beta_{22}' \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ having the same trace and thread computations as τ' and τ. Note that β_{22}' is a subsequence of β_2', which in turn is a subsequence of β' and hence of β.

The case when $\mathsf{e}_3 \not\rightarrow_{hb} \mathsf{e}_2$ is symmetric. We apply the induction hypothesis to τ with respect to e_2 and e_3, getting $\tau' = \alpha \cdot \mathsf{e}_1 \cdot \beta' \cdot \mathsf{e}_2 \cdot \mathsf{e}_3 \cdot \gamma \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ with the same trace and thread computations as τ. Applying it again to τ' with respect to e_1 and e_2 gives $\tau'' = \alpha \cdot \beta_1' \cdot \mathsf{e}_2 \cdot \mathsf{e}_1 \cdot \beta_2' \cdot \mathsf{e}_3 \cdot \gamma$. The computation has the same trace and thread computations as τ' and τ. Since β_2' is a subsequence of β', $\beta_2' \cdot \mathsf{e}_3$ is a subsequence of β. □

Lemma 6.4 (Locality)**.** *In a minimal violation only a single thread delays stores.*

Proof. Consider a minimal violation $\tau \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$. By definition, τ has cyclic happens-before relation, therefore, at least one thread delayed stores. Suppose that at least two threads delayed stores. By Lemma 6.2, each flush was delayed past a load of the same thread. Let e_2' of thread tid_2 be the overall last delayed flush event in τ, and let e_2 be the last load of tid_2 overstepped by e_2'. Similarly, let e_1' be the overall last delayed flush event in a thread $\mathsf{tid}_1 \neq \mathsf{tid}_2$. Let e_1 be the last load overstepped by e_1'.

The following fundamental mutual dispositions of these four events are possible:

1. $\tau = \gamma_1 \cdot \mathsf{e}_1 \cdot \gamma_2 \cdot \mathsf{e}'_1 \cdot \gamma_3 \cdot \mathsf{e}_2 \cdot \gamma_4 \cdot \mathsf{e}'_2 \cdot \gamma_5$,
2. $\tau = \gamma_1 \cdot \mathsf{e}_2 \cdot \gamma_2 \cdot \mathsf{e}_1 \cdot \gamma_3 \cdot \mathsf{e}'_1 \cdot \gamma_4 \cdot \mathsf{e}'_2 \cdot \gamma_5$,
3. $\tau = \gamma_1 \cdot \mathsf{e}_1 \cdot \gamma_2 \cdot \mathsf{e}_2 \cdot \gamma_3 \cdot \mathsf{e}'_1 \cdot \gamma_4 \cdot \mathsf{e}'_2 \cdot \gamma_5$.

In these three computations each pair $(\mathsf{e}_i, \mathsf{e}'_i)$ provides a happens-before cycle: $\mathsf{e}'_i \to^+_{po} \mathsf{e}_i$ and, by Lemma 6.3 and minimality, $\mathsf{e}_i \to^+_{hb} \mathsf{e}'_i$ through the appropriate subword of τ.

In the first disposition τ is not minimal, since it can be shortened to the violating computation $\tau' := \gamma_1 \cdot \mathsf{e}_1 \cdot \gamma_2 \cdot \mathsf{e}'_1 \cdot \beta$ with $\#(\tau') < \#(\tau)$, where the β part contains only flush events that complete the buffered stores.

In the second disposition τ is not minimal either. Starting from e_2 and until e'_2, thread tid_2 does not have any events, except delayed stores (Lemma 6.2). Therefore, e_2 and all program-order-later events of tid_2 can be safely removed from τ without affecting the happens-before cycle produced by tid_1. The resulting computation has a smaller number of delays (due to the removed e_2), but its trace still includes the cycle by tid_1. A contradiction to minimality of τ.

Lastly, in the third case τ is also not minimal. First, we delete γ_5, as it does not contain delayed flush events, by choice of e'_2. Then, we erase all events from γ_4 that do not belong to tid_2: $\gamma'_4 := \gamma_4 \downarrow \mathsf{tid}_2$. By construction, the resulting computation τ' is a feasible TSO computation:

$$\tau' := \gamma_1 \cdot \mathsf{e}_1 \cdot \gamma_2 \cdot \mathsf{e}_2 \cdot \gamma_3 \cdot \mathsf{e}'_1 \cdot \gamma'_4 \cdot \mathsf{e}'_2 \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P}).$$

Computation τ' still contains the happens-before cycle $\mathsf{e}'_1 \to^+_{po} \mathsf{e}_1 \to^+_{hb} \mathsf{e}'_1$ inherited from τ. Since deleting events cannot increase the number of delays and τ is a minimal violation, $\#(\tau') = \#(\tau)$. Therefore, τ' is a minimal violation too.

By Lemma 6.3, $\mathsf{e}_2 \to^+_{hb} \mathsf{e}'_2$ through $\gamma_3 \cdot \mathsf{e}'_1 \cdot \gamma'_4$. By the choice of e_1 and e'_1 and in accordance with Lemma 6.2, $(\gamma_2 \cdot \mathsf{e}_2 \cdot \gamma_3) \downarrow \mathsf{tid}_1$ only contains delayed stores that were issued before e_1. By definition, γ'_4 does not contain events of tid_1 at all. Therefore, e_1 is the program-order-last (load) event of tid_1 in τ'. It can be safely removed from τ' without affecting the cycle of tid_2. The resulting computation is

$$\tau'' := \gamma_1 \cdot \gamma_2 \cdot \mathsf{e}_2 \cdot \gamma_3 \cdot \mathsf{e}'_1 \cdot \gamma'_4 \cdot \mathsf{e}'_2 \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P}).$$

Note that $\#(\tau'') < \#(\tau') = \#(\tau)$, but computation τ'' still contains the cycle $\mathsf{e}'_2 \to^+_{po} \mathsf{e}_2 \to^+_{hb} \mathsf{e}'_2$. A contradiction to minimality of τ. □

Clearly, checking robustness amounts to checking whether a program has no minimal violations. Actually, we can simplify the task and restrict the form of computations for which we must look even further.

We call a computation $\tau \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ a *TSO witness* if $\tau = \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3 \cdot \tau_4$, where:

WIT-A Only a single thread tid_A delays stores. We call this thread an *attacker*.

WIT-B Event e_3 is the flush event of the first delayed store in tid_A, e_1 is the matching store event, e_2 is the last, load event of tid_A before e_3. This means, all events in τ_3 belong to threads other than the attacker. We call these threads *helpers*.

WIT-C The load e_2 is not an early read.

WIT-D $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}$ for each $\mathsf{e} \in \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3$.

WIT-E τ_4 consists solely of flush events of tid_A.

We call the triplet $\mathsf{A} := (\mathsf{tid}_\mathsf{A}, \mathsf{instr}(\mathsf{e}_1), \mathsf{instr}(\mathsf{e}_2))$ the *attack* of witness τ.

The following lemma characterizes robustness in terms of absence of TSO witnesses.

Theorem 6.5. *A program $\mathcal{P}$ is robust iff it has no TSO witnesses.*

Proof. Assume that the program is not robust. Then, it has a minimal violation τ. For τ, WIT-A is implied by Lemma 6.4. Let tid_A be the single thread that delayed stores. Let $\mathsf{e}_3 \in \tau$ be the leftmost flush event of a delayed store, $\mathsf{e}_1 \in \tau$ be the matching store event, and e_2 is the rightmost event of thread tid_A before e_3. By Lemma 6.2, e_2 is a load event and is not an early read. Therefore, WIT-B and WIT-C hold. WIT-D holds by Lemma 6.3. Assume WIT-E does not hold. Without loss of generality assume that in τ flush events of helpers immediately follow matching store events. We can filter τ_4 and keep only the flush events with matching store events in $\mathsf{e}_1 \cdot \tau_2$. Let the resulting subsequence of τ_4 be τ_4'. Then, the computation $\tau' := \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3 \cdot \tau_4' \in \mathsf{C}_{\mathsf{tso}}(\mathcal{P})$ is a minimal violation, for which WIT-E holds. (Effectively, τ' reproduces the behavior of τ up to e_3, after which the attacker immediately flushes all the buffered stores, and the other threads simply stop.)

Assume that the program is robust. Then, it has no TSO witnesses, as each TSO witness has a happens-before cycle $\mathsf{e}_1 \rightarrow_{po}^{+} \mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}_3$, where e_1 and e_3 are matching store and flush events. □

Example 6.6. Computation $\tau := abc \cdot \mathsf{flush}(c) \cdot d \cdot \mathsf{flush}(a)$ from Example 2.4 is a TSO witness of the SB program (Figure 1.1). Here, $\tau_1 := \varepsilon$, $\mathsf{e}_1 := a$, $\tau_2 := \varepsilon$, $\mathsf{e}_2 := b$, $\tau_3 := c \cdot \mathsf{flush}(c) \cdot d$, $\tau_4 := \mathsf{flush}(a)$. According to Theorem 6.5, the SB program must be not robust against TSO, which is indeed the case. Note that the program has a similar TSO witness with the second store delaying the store.

6.2 From Robustness to SC Reachability

Fix a program $\mathcal{P}$ and an attack $\mathsf{A} := (\mathsf{tid}_\mathsf{A}, \mathsf{instr}_{\mathsf{store}}, \mathsf{instr}_{\mathsf{load}})$. In this section we show how to check whether the program $\mathcal{P}$ has a TSO witness with the attack A.

A TSO witness with attack A makes very limited use of the store buffers. This allows us to model it with SC computations of an *instrumented* program $\mathcal{P}_\mathsf{A}$. By instrumentation we mean an extension of each thread of the program with additional instructions. The attacker thread is instrumented with instructions that soundly emulate delaying of stores. The helper threads are instrumented to check the existence of a happens-before path $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}_3$.

The idea of the attacker instrumentation comes from the observation that the stores delayed by the attacker are never read by any helper thread. The stores can only be observed by the attacker's loads which can read early from them. Fortunately, early reads can access only the last buffered value for each address. Rather than storing the whole buffer contents, we will store the last

written values, one per each address. For this, with each address a we associate a *shadow* address (a, d) containing the last buffered value. The delayed stores will update the value at the shadow address, loads will take the value from the shadow address, if it was written there, and perform the usual load from memory otherwise. The helpers will not know anything about these shadow addresses and, therefore, will not observe the delayed stores.

The helper instrumentation is responsible for checking $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}_3$. Assume that a helper produces a new event e. When does $\mathsf{e}_2 \rightarrow_{hb}^{*} \mathsf{e}$ hold? First, it holds if some previous event e' of this helper thread has contributed to the happens-before path: $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}' \rightarrow_{po}^{+} \mathsf{e}$. We can remember whether a thread has contributed to the happens-before path in the control state of the thread. Second, $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}$ if there is some previous event e' (possibly, in a different thread), $\mathsf{e}_2 \rightarrow_{hb}^{*} \mathsf{e}'$, with $\mathsf{addr}(\mathsf{e}') = \mathsf{addr}(\mathsf{e})$ and at least one of the two events is a flush event. For each address we keep track whether there was such an access to this address that is happens-before dependent on e_2. We also remember whether among these accesses there was a store, or all of them were loads. If there was a store access, the happens-before path can be extended by a store or a load event. If there was a load access, the path can be only extended by a store event. We again use shadow addresses $(\mathsf{a}, \mathsf{hb})$ to store this information.

6.2.1 Instrumentation of an Attacker

Consider the attacker thread $\mathcal{T}_{\mathsf{tid_A}}$. In the τ_1 part of a TSO witness the attacker executes under SC semantics. Just before executing the store instruction $\mathsf{instr}_{\mathsf{store}} = q_1 \xrightarrow{\texttt{mem[}e_{\mathsf{a}}\texttt{]}\leftarrow e_{\mathsf{v}}} q_2$ it can decide to start delaying the stores and thereby enter the τ_2 part of the computation. Therefore, we add the instructions that remember the address of the first delayed store, store the written value at the corresponding shadow address, and enter the copy of the attacker's code that simulates the τ_2 part of the TSO witness:

$$q_1 \xrightarrow{r_{\mathsf{a}}\leftarrow e_{\mathsf{a}}} q_x \xrightarrow{\texttt{mem[}(r_{\mathsf{a}},\mathsf{d})\texttt{]}\leftarrow(e_{\mathsf{v}},\mathsf{d})} \widehat{q_2}. \tag{6.1}$$

Here and further, the control states and the registers not mentioned in the original instruction are assumed not to be used in the original thread. Moreover, the states q_x denote fresh states in each rule application.

In the τ_2 part, a store instruction $q_1 \xrightarrow{\texttt{mem[}e_{\mathsf{a}}\texttt{]}\leftarrow e_{\mathsf{v}}} q_2$ from $\mathcal{I}_{\mathsf{tid_A}}$ translates to a store instruction performing the write to the shadow address:

$$\widehat{q_1} \xrightarrow{\texttt{mem[}(e_{\mathsf{a}},\mathsf{d})\texttt{]}\leftarrow(e_{\mathsf{v}},\mathsf{d})} \widehat{q_2}. \tag{6.2}$$

A load instruction $q_1 \xrightarrow{r\leftarrow\texttt{mem[}e\mathsf{a}\texttt{]}} q_2$ reads the value from memory only if there was no delayed store to the loaded address; otherwise, it reads from the shadow address:

$$\begin{aligned} \widehat{q_1} &\xrightarrow{\texttt{assume(mem[}(e_{\mathsf{a}},\mathsf{d})\texttt{]=0)}} \widehat{q_{x1}} \xrightarrow{r\leftarrow\texttt{mem[}e_{\mathsf{a}}\texttt{]}} \widehat{q_2}, \\ \widehat{q_1} &\xrightarrow{\texttt{assume(mem[}(e_{\mathsf{a}},\mathsf{d})\texttt{])}} \widehat{q_{x2}} \xrightarrow{(r,\mathsf{d})\leftarrow\texttt{mem[}(e_{\mathsf{a}},\mathsf{d})\texttt{]}} \widehat{q_2}. \end{aligned} \tag{6.3}$$

A local assignment or a condition cmd is not changed in the attacker's copy: $q_1 \xrightarrow{\mathsf{cmd}} q_2$ simply becomes

$$\widehat{q_1} \xrightarrow{\mathsf{cmd}} \widehat{q_2}. \tag{6.4}$$

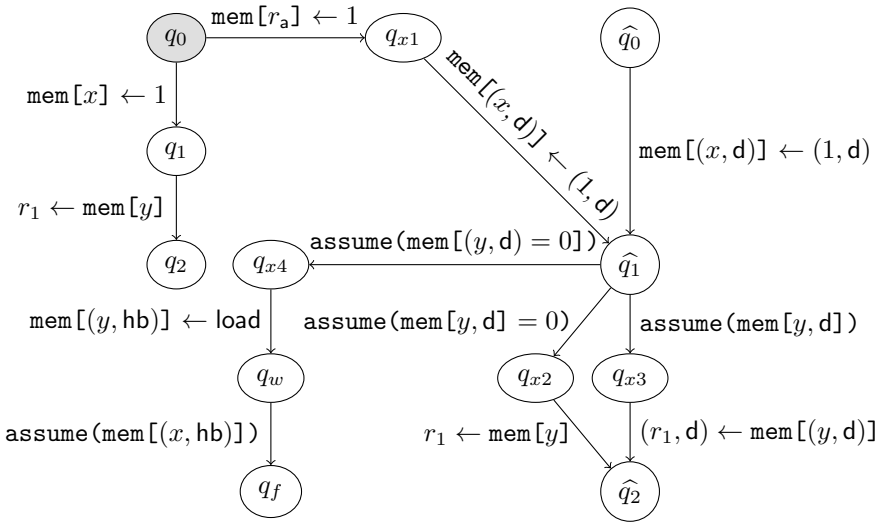

Figure 6.1: Thread 1 of the SB program (Figure 1.1) instrumented as an attacker for the attack $\mathsf{A} := (1, q_1 \xrightarrow{\texttt{mem}[x] \leftarrow 1} q_2, q_2 \xrightarrow{r_1 \leftarrow \texttt{mem}[y]} q_3)$.

Memory fences are forbidden in the attacker's copy, as they would prevent delaying the stores past it, therefore, we do not add any instructions for `mfence` instructions.

Finally, the attacker can quit the τ_2 part by executing the load instruction from the attack. Let $\mathsf{instr}_{\mathsf{load}} = q_1 \xrightarrow{r \leftarrow \texttt{mem}[e]} q_2$. The attacker checks that the load does not read early and signals the helpers that they can start building the happens-before path:

$$\widehat{q_1} \xrightarrow{\texttt{assume}(\texttt{mem}[(e,\mathsf{d})]=0)} q_x \xrightarrow{\texttt{mem}[(e,\mathsf{hb})] \leftarrow \mathsf{load}} q_w. \tag{6.5}$$

Note that the attacker does not actually perform the load, as it is not going to use the read value anyway.

After simulating $\mathsf{instr}_{\mathsf{load}}$ the attacker waits until the helpers build the happens-before path and enters state q_f:

$$q_w \xrightarrow{\texttt{assume}(\texttt{mem}[(r_\mathsf{a},\mathsf{hb})])} q_f. \tag{6.6}$$

For simplicity, in the above rules we allowed memory accesses (`mem[]`) within expressions. One can transform instructions using these expressions to the canonical form by prepending them with appropriate load transitions.

Example 6.7. Figure 6.1 shows the result of the attacker instrumentation of the first thread of the SB program (Figure 1.1).

6.2.2 Instrumentation of a Helper

An instrumented helper thread has two modes of execution. Initially, it runs as usual. When it produces the first event that is dependent on the load event

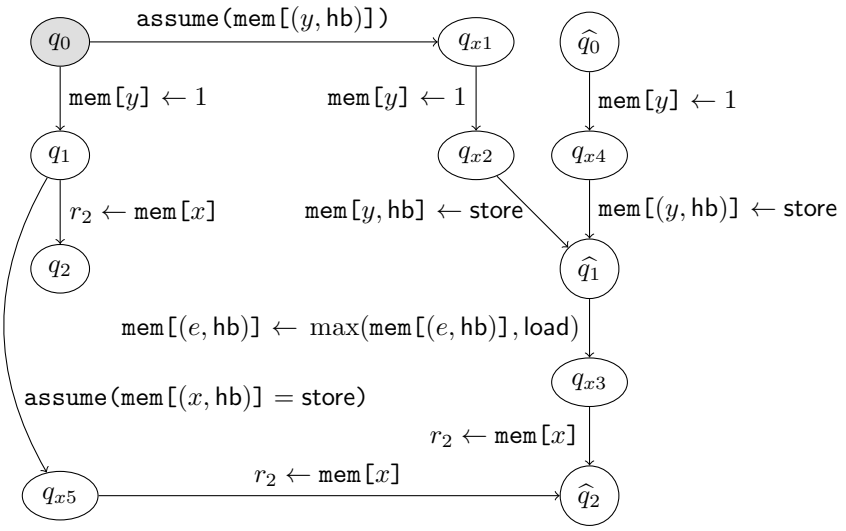

Figure 6.2: Thread 2 of the SB program (Figure 1.1) instrumented as a helper.

e_2, it remembers this in the control state (enters the code copy). So, a load instruction $q_1 \xrightarrow{r \leftarrow \texttt{mem}[e]} q_2$ produces

$$q_1 \xrightarrow{\texttt{assume(mem}[(e,\mathsf{hb})]=\mathsf{store})} q_x \xrightarrow{r \leftarrow \texttt{mem}[e]} \widehat{q_2}. \tag{6.7}$$

A store instruction $q_1 \xrightarrow{\texttt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}} q_2$ produces

$$q_1 \xrightarrow{\texttt{assume(mem}[(e,\mathsf{hb})])} q_{x1} \xrightarrow{\texttt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}} q_{x2} \xrightarrow{\texttt{mem}[(e_\mathsf{a},\mathsf{hb})] \leftarrow \mathsf{store}} \widehat{q_2}. \tag{6.8}$$

All transitions from the hat states will generate events that are happens-before dependent on the attacker's e_2. So, for a load $q_1 \xrightarrow{r \leftarrow \texttt{mem}[e]} q_2$ the instrumentation adds

$$\widehat{q_1} \xrightarrow{\texttt{mem}[(e,\mathsf{hb})] \leftarrow \max(\texttt{mem}[(e,\mathsf{hb})],\mathsf{load})} q_x \xrightarrow{r \leftarrow \texttt{mem}[e]} \widehat{q_2}, \tag{6.9}$$

where max returns the maximum of its arguments, assuming $0 < \mathsf{load} < \mathsf{store}$. A store $q_1 \xrightarrow{\texttt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}} q_2$ gives the instructions

$$\widehat{q_1} \xrightarrow{\texttt{mem}[e_\mathsf{a}] \leftarrow e_\mathsf{v}} q_x \xrightarrow{\texttt{mem}[e_\mathsf{a},\mathsf{hb}] \leftarrow \mathsf{store}} \widehat{q_2}. \tag{6.10}$$

Finally, commands cmd that are local assignments, conditionals, `mfence` are copied as is: a transition $q_1 \xrightarrow{\mathsf{cmd}} q_2$ simply gives

$$\widehat{q_1} \xrightarrow{\mathsf{cmd}} \widehat{q_2}. \tag{6.11}$$

Example 6.8. Figure 6.2 shows the result of the helper instrumentation of the second thread of the SB program.

6.2.3 Soundness and Completeness

Given a program $\mathcal{P}_\mathsf{A}$ instrumented for attack $\mathsf{A} := (\mathsf{tid}_\mathsf{A}, \mathsf{instr}_\mathsf{store}, \mathsf{instr}_\mathsf{load})$, we call an SC state $(\mathsf{sn}, \mathsf{pc}, \mathsf{mem})$ of this program a *goal* state if $\mathsf{pc}(\mathsf{tid}_\mathsf{A}) = q_f$. The following theorem says that a program $\mathcal{P}$ has a TSO witness with attack A iff the instrumented program $\mathcal{P}_\mathsf{A}$ can reach a goal state under SC.

Theorem 6.9 (Soundness and Completeness)**.** *A program $\mathcal{P}$ has a TSO witness with attack $A := (tid_A, instr_{store}, instr_{load})$ iff $\mathcal{P}_A$ can reach the goal state under SC.*

Proof.

Soundness Suppose that the program $\mathcal{P}_\mathsf{A}$ can reach the goal state. Then, it has to reach it via a computation of the following form:

$$\sigma := \sigma_1 \cdot \mathsf{e}_\mathsf{store} \cdot \sigma_2 \cdot \mathsf{e}_\mathsf{hb} \cdot \sigma_3 \cdot \mathsf{e}_f.$$

The last event, e_f, is the event produced by the instruction from (6.6). Therefore, the condition from (6.6) must have been satisfied, i.e., the address $(r_\mathsf{a}, \mathsf{hb})$ must contain a non-zero value. Stores to the addresses of the form $(\mathsf{a}, \mathsf{hb})$ exist in the attacker, (6.5), and in the helpers. The value at $(r_\mathsf{a}, \mathsf{hb})$ could not be set by the attacker in (6.5), due to the condition just before the store in (6.5), the definition of r_a and the following store in (6.1). Therefore, the non-zero value was written to $(r_\mathsf{a}, \mathsf{hb})$ by a helper in the σ_3 part of the computation.

If the helper wrote to $(r_\mathsf{a}, \mathsf{hb})$, it has entered its code copy. It can only enter its code copy if it has read a non-zero value from $(\mathsf{a}, \mathsf{hb})$, see (6.7), (6.8). This value must have been stored by another helper or by the attacker. The first helper that has enter its code copy must have read the value written by the attacker, by executing the store instruction from (6.5), event e_hb. This means, the attacker has entered its code copy by executing the store from (6.1), event $\mathsf{e}_\mathsf{store}$, where it delayed stores, and reached the instrumented $\mathsf{instr}_\mathsf{load}$. Moreover, the check in (6.5) succeeded, which means that no store to the address being loaded was delayed, (6.1), (6.2).

Altogether, in σ_1 the attacker and the helper executed the instructions of the original program. Event $\mathsf{e}_\mathsf{store}$ is produced by the instrumented $\mathsf{instr}_\mathsf{store}$. In σ_2 the attacker executed the instrumented code copy, the helpers executed the original instructions. Event e_hb is produced by the instrumented $\mathsf{instr}_\mathsf{load}$. In σ_3 the helpers executed original instructions and code copies, the attacker only executed the instructions added by (6.6).

We now turn σ into the following computation τ of program $\mathcal{P}$:

$$\tau := \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3 \cdot \tau_4.$$

In τ_1 the program executes the instructions executed by $\mathcal{P}_\mathsf{A}$ in σ_1, in the same order, with flush events immediately following the matching stores. After that, by definition of SC and TSO semantics, the program $\mathcal{P}$ will reach the TSO state which has empty buffers and the same control and memory configuration as the SC state reached by $\mathcal{P}_\mathsf{A}$ just before $\mathsf{e}_\mathsf{store}$.

Then, the attacker of $\mathcal{P}_\mathsf{A}$ can execute $\mathsf{instr}_\mathsf{store}$, buffer the store, but not flush it. The attacker can next execute the instructions, from which the instrumented instructions executed by the attacker in τ_2 were produced. Store instructions

are executed in place of corresponding store events, the stores are buffered. An instrumented load produces two events: the conditional and the load, (6.3), the program $\mathcal{P}$ executes the corresponding load instruction only in place of the load event. One can show that τ_2 is executable by induction, with the following invariants:

INV-1-A The control state of the attacker thread in $\mathcal{P}_\mathsf{A}$ is the hat-version of the control state of the attacker thread in $\mathcal{P}$ after corresponding transition.

INV-1-B The control states of the helper threads are the same in both programs.

INV-1-C The memory configurations of programs $\mathcal{P}$ and $\mathcal{P}_\mathsf{A}$ are the same, modulo the addresses and registers added by the instrumentation.

INV-1-D Each shadow address (a, d) in $\mathcal{P}_\mathsf{A}$ contains the value written by the last delayed attacker's store to address a of $\mathcal{P}$, or 0 if there was no such delayed store.

Finally, the attacker executes $\mathsf{instr}_{\mathsf{load}}\ (\mathsf{e}_2)$.

In τ_3 all events belong to helpers. Similarly, they execute the instructions, from which the instrumented instructions executed by the helpers in σ_3 were produced, in the place of the events e with $\mathsf{lab}(\mathsf{instr}(\mathsf{e}))$ being the command of the original instruction. One can show that τ_3 is executable by induction, with the following invariants:

INV-2-A If in the instrumented program, after a prefix of τ_3 is executed, the address $(\mathsf{a}, \mathsf{hb})$ contains store (load), then in the corresponding prefix of σ_3 there is a store (load) event with address a which is happens-before dependent on e_2.

INV-2-B The memory configurations of programs $\mathcal{P}$ and $\mathcal{P}_\mathsf{A}$ are the same, modulo the addresses and registers added by the instrumentation.

INV-2-C The control state of a helper in $\mathcal{P}_\mathsf{A}$ is the same as in $\mathcal{P}$ if it did not contribute to the executed part of τ_3. Otherwise, it is the hat-version of this state.

The $\mathsf{e}_3 \cdot \tau_4$ part of the computation consists of flush events for the delayed stores.

Now we show that the computation τ is *almost* a TSO witness. Indeed, WIT-A, WIT-B, WIT-E hold by construction. WIT-C holds due to INV-1-D and the check in (6.5). WIT-D does not immediately hold: although, by INV-2-A, there is a happens-before cycle $\mathsf{e}_2 \rightarrow_{hb}^{+} \mathsf{e}_3 \rightarrow_{hb} \mathsf{e}_2$, not all events in τ_3 are happens-before dependent on e_2. One can transform τ to a TSO witness τ' by moving the events in τ_3 that are not happens-before dependent on e_2 into τ_2, similar to how it was done in the proof of Lemma 6.3.

Completeness Suppose there is a TSO witness τ for attack A, as defined in Section 6.1:

$$\tau = \tau_1 \cdot \mathsf{e}_1 \cdot \tau_2 \cdot \mathsf{e}_2 \cdot \tau_3 \cdot \mathsf{e}_3 \cdot \tau_4.$$

We show that the instrumented program $\mathcal{P}_\mathsf{A}$ has an SC computation that leads to a goal state. Without loss of generality we assume that flush events in τ

immediately follow the matching store events (except for those delayed by the attacker).

In the beginning, the instrumented attacker and helper threads can execute the same instructions that were executed by the original program $\mathcal{P}$ in the τ_1 part of the TSO witness. After that, by definition of SC and TSO semantics, the program $\mathcal{P}_\mathsf{A}$ will reach the SC state which has the same control and memory configuration as the TSO state reached by $\mathcal{P}$ just before e_1. The TSO state of $\mathcal{P}$ has empty buffer configuration.

Then, the attacker of $\mathcal{P}_\mathsf{A}$ can execute the instruction $\mathsf{instr}_{\mathsf{store}} = \mathsf{instr}(\mathsf{e}_1)$ instrumented by (6.1) and enter the code copy. In the code copy it executes the instrumented versions (6.2), (6.3), (6.4) of the instructions executed by the attacker thread of $\mathcal{P}$ in τ_2. In the part τ_2 the helpers of $\mathcal{P}_\mathsf{A}$ still execute the instructions of the original program. The invariants INV-1-A to INV-1-D are maintained. Note that τ_2 does not contain memory fences, otherwise the store e_3 could not have been delayed past the load e_2. Therefore, the absence of copies of the `mfence` instructions cannot provoke a block of the attacker.

Finally, the attacker executes $\mathsf{instr}_{\mathsf{load}} = \mathsf{instr}(\mathsf{e}_2)$ instrumented according to (6.5) and reaches control state q_w. The instrumentation has a condition that the load e_2 was not an early read, which is guaranteed by WIT-C.

All events in τ_3 belong to helpers. By WIT-D, they are in happens-before relation with e_2. The transitions of helpers of $\mathcal{P}$ in τ_3 can be simulated by the helper thread of $\mathcal{P}_\mathsf{A}$ so that invariants INV-2-A to INV-2-C hold. For example, consider a store transition e performed by the helper thread. If this is the first transition of the thread in τ_3, by WIT-D, it is happens-before dependent on an earlier event $\mathsf{e}' \in \tau_3$ of some other thread, $\mathsf{addr}(\mathsf{e}) = \mathsf{addr}(\mathsf{e}') = \mathsf{a}$. Consequently, $(\mathsf{a}, \mathsf{hb})$ contains either load or store, and the helper thread in $\mathcal{P}_\mathsf{A}$ can execute the instructions defined by (6.8) and enter the code copy. If this is not the first transition of the thread, the thread is already in the code copy, and can execute the instructions defined by (6.10). It is easy to see that the invariants continue to hold. Handling of the other kinds of instructions is similar.

At least one of the helper's events e in τ_3 is a load or a store to the address $\mathsf{addr}(\mathsf{e}) = \mathsf{addr}(\mathsf{e}_1) = \mathsf{a}$. Otherwise, WIT-D would not hold. When executing the instrumented version of $\mathsf{instr}(\mathsf{e})$ in the $\mathcal{P}_\mathsf{A}$, the helper will set $(\mathsf{a}, \mathsf{hb})$ to a non-zero value, see (6.9) and (6.10). Therefore, at the next step the attacker (situated in the control state q_w) will be able to reach q_f in accordance with (6.6) and make the instrumented program reach the goal state. □

The following statement is a corollary of Theorem 6.5 and Theorem 6.9.

Theorem 6.10. *A program $\mathcal{P}$ is robust against TSO if there is no attack A such that $\mathcal{P}_A$ reaches a goal state under SC.*

The theorem gives us a procedure for checking robustness against TSO for a program $\mathcal{P}$. One can enumerate all attacks (their number is only quadratic in the size of the program) and for each attack check, whether $\mathcal{P}_\mathsf{A}$ reaches a goal state. Altogether, robustness reduces to a quadratic number of reachability queries. Notably, these reachability queries are independent and can be performed in parallel.

Actually, one can reduce robustness checking to a *single* reachability query. For this, each thread must be instrumented both as an attacker and as a helper.

Further, the instrumentations (6.1), (6.5) must be applied to each store, respectively, load instruction. In order to forbid multiple threads to become attackers, one has to introduce a global flag. When a thread is going to enter the code copy, (6.1), it tries to raise (modify) this flag using atomic compare-and-swap. On success, i.e., if the flag contained the initial value, it enters the code copy and starts delaying stores. On failure, it continues to run the original code. The correctness of the just described instrumentation is proven similar to Theorem 6.9.

6.3 Parameterized Robustness

In this section we consider the problem of checking robustness against TSO for parameterized programs, as defined in Section 2.8.

Problem 6.11 (Parameterized robustness against TSO)**.** Given a parameterized program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}(I)) = T_{\mathsf{tso}}(\mathcal{P}(I))$ for all $I \in \mathbb{N}^{\mathsf{TID}}$.

By Theorem 6.10, a parameterized program is robust iff for all $I \in \mathbb{N}^{\mathsf{TID}}$ and for any attack the program $\mathcal{P}(I)_{\mathsf{A}}$ does not reach the goal state. However, we cannot instrument unboundedly many program instances. Instead, we instrument the parameterized program itself and reduce parameterized robustness against TSO to parameterized SC reachability. The idea here is to swap instantiation and instrumentation, i.e., instrument the parameterized program itself.

Actually, we can apply the instrumentation from Section 6.2 to a parameterized program almost without changes. Only the attacker instrumentation, namely (6.1), requires extra care. In an instance program, only one copy of the thread should act as attacker, the remaining copies must behave like helpers. Therefore, the thread must be instrumented not only as an attacker, but also as a helper. To ensure that only one copy of the attacker delays stores, we introduce a global flag. When a thread is going to enter the code copy, (6.1), it tries to raise (modify) this flag using atomic compare-and-swap. On success, i.e., if the flag contained the initial value, it enters the code copy and starts delaying stores. On failure, it continues to run the original code.

Theorem 6.12. *A parameterized program $\mathcal{P}$ is robust against TSO if there is no attack A, such that parameterized program $\mathcal{P}_{\mathsf{A}}$ reaches a goal state under SC.*

6.4 Decidability and Complexity

Until now, we did not impose any restrictions on the programs for which we check robustness. The reductions presented in Sections 6.2 and 6.3 are independent of the data domain, size of the address space, thread creation time (static number of threads vs. dynamic thread creation). Although the programming model used in the thesis omits recursion, the reductions are applicable to recursive programs as well. It is the back-end model checker that has to deal with the complexity of a particular programming model. In this section we limit ourselves to programs with finite data domain (and address space) and derive decidability and complexity results for this class of programs.

The reduction of robustness to SC reachability presented in Section 6.2 gives an alternative proof of the complexity result shown earlier in Theorem 5.5.

Theorem 6.13. *Robustness against TSO for programs over finite domains is* PSPACE*-complete.*

Proof. By Theorem 6.10, in order to check robustness of a program $\mathcal{P}$, one can enumerate all the attacks of this program and for each attack check whether $\mathcal{P}_{\mathsf{A}}$ reaches a goal state under SC. Since the number of attacks is quadratic in the size of the program, the enumeration can be done in PSPACE. The size of the instrumented program $\mathcal{P}_{\mathsf{A}}$ is linear in the size of the original program. By Lemma 2.7, the reachability of a goal state can be decided in PSPACE. Altogether, this gives us the PSPACE upper bound.

For the lower bound we reduce SC state reachability to robustness. The former problem is PSPACE-hard already for single-threaded programs (Lemma 2.7), which are trivially robust (under TSO a load always reads the value written by the last store to the address). In order to check whether a single-threaded program reaches a state with address a containing a value different from zero, we extend it with two threads that, first, check whether $\texttt{mem[a]} \neq 0$ and, if yes, violate robustness, e.g., by executing the SB program, Figure 1.1. The extended program is robust iff the state with address a containing non-zero is reachable in the original program. □

The reduction of parameterized robustness to parameterized SC reachability presented in Section 6.3 gives us the following complexity result.

Theorem 6.14. *Parameterized robustness against TSO for programs over finite domains is decidable and* EXPSPACE*-hard — already for programs with* $|\mathsf{DOM}| \geq 3$*.*

Proof. By Theorem 6.12, in order to check robustness, we can enumerate all the attacks and for each attack check if the parameterized instrumented program reaches a goal state. The enumeration of quadratic number of attacks and their instrumentation can be done in PSPACE. State reachability of a goal state is decidable, Lemma 2.8. Altogether, this means that parameterized TSO-Robustness is decidable.

The EXPSPACE-hardness follows from the reduction of parameterized SC reachability to parameterized robustness (similar to the one from the proof of the previous theorem) and EXPSPACE-hardness of parameterized state reachability for programs with $|\mathsf{DOM}| \geq 3$, Lemma 2.9. □

6.5 Enforcing Robustness

In this section, our goal is to insert memory fences into a program to make it robust. By *inserting a fence* into state q of thread tid of program $\mathcal{P}$ we mean the following modification of the program. First, we add a fresh state q_x to Q_{tid}. Then, we replace each instruction $q \xrightarrow{\mathsf{cmd}} q' \in \mathcal{I}_{\mathsf{tid}}$ with the instruction $q_x \xrightarrow{\mathsf{cmd}} q'$. Finally, we add a memory fence $q \xrightarrow{\texttt{mfence}} q'$.

Let $\mathsf{FENCES} := \bigcup_{\mathsf{tid}} \{\mathsf{tid}\} \times Q_{\mathsf{tid}}$ be the set of all possible fence locations. We call a set $\mathcal{F} \subseteq \mathsf{FENCES}$ a *valid fence set for program* $\mathcal{P}$ if inserting memory fences into the given states yields a robust program. We say that $\mathcal{F}$ is *irreducible*

if it does not have any strict subset which is a valid fence set. Clearly, a fence set that includes all the destination control states of store instructions of the program is a valid fence set. In general, however, we would like to compute a valid fence set which is *optimal* in some sense. Therefore, we pose the *optimal TSO-fencing* problem:

Problem 6.15 (Optimal TSO-fencing)**.** Given a program $\mathcal{P}$ and a *cost function* $\mathcal{C}\colon \mathsf{FENCES} \to \mathbb{R}$, compute a valid fence set with $\sum_{f \in \mathcal{F}} \mathcal{C}(f)$ minimal.

The parameterized version of this problem is defined as expected.

We consider two criteria of optimality: minimization of program size and maximization of program performance. By solving the problem for $\mathcal{C} \equiv 1$ we compute a fence set of minimal size, thus minimizing the code size of the fenced program. Maximization of program performance requires minimizing the number of times memory fence instructions are executed: practical measurements, Appendix A, show that it is impossible to save CPU cycles by executing more fences, but with less stores in the TSO buffer. For this, $\mathcal{C}(f)$ is defined as the frequency of reaching the fence location f in program executions. Concrete values of $\mathcal{C}$ can be either estimated by profiling or computed by mathematical reasoning about the program.

From the complexity point of view, fence computation is at least as hard as robustness. Indeed, robustness holds if and only if the optimal valid fence set is $\mathcal{F} = \emptyset$. Actually, since fence sets can be enumerated, computing an optimal valid fence set does not require more space than checking robustness. This gives us the following theorem.

Theorem 6.16. *For programs over finite domains, optimal TSO-fencing is* PSPACE*-complete. In the parameterized case, it is decidable and* EXPSPACE*-hard for programs with* $|\mathit{DOM}| \geq 3$.

In the remainder of the section we give a practical algorithm for optimal TSO-fencing.

6.5.1 Fence Sets for Attacks

Given an attack $\mathsf{A} := (\mathsf{tid}, \mathsf{instr}_{\mathsf{store}}, \mathsf{instr}_{\mathsf{load}})$, we define the set D_{A} of locations *involved* into the attack as $D_{\mathsf{A}} := \{(\mathsf{tid}, q) \mid \mathsf{dst}(\mathsf{instr}_{\mathsf{store}}) \to^* q \to^* \mathsf{src}(\mathsf{instr}_{\mathsf{load}})\}$. We call a set of locations $\mathcal{F}_{\mathsf{A}}$ *an eliminating fence set for attack* A if inserting fences at all locations in $\mathcal{F}_{\mathsf{A}}$ eliminates the attack (i.e., forbids all the TSO witnesses for this attack). We call the set $\mathcal{F}_{\mathsf{A}}$ *irreducible* if it does not have any strict subset which is an eliminating fence set for attack A. Note that any irreducible eliminating set $\mathcal{F}_{\mathsf{A}}$ satisfies $\mathcal{F}_{\mathsf{A}} \subseteq D_{\mathsf{A}}$.

Example 6.17. The SB program (Figure 1.1) has two irreducible eliminating fence sets: $\mathcal{F}_{\mathsf{A}} = \{(1, q_1)\}$ eliminates the only attack by the first thread and $\mathcal{F}_{\mathsf{A}'} = \{(2, q_1)\}$ eliminates the only attack by the second thread.

Lemma 6.18. *Every irreducible valid fence set $\mathcal{F}$ can be represented as a union of irreducible eliminating fence sets for all attacks having TSO witnesses.*

Proof. By Theorem 6.5, fence set $\mathcal{F}$ must forbid all the TSO witnesses. Therefore, it includes some irreducible eliminating fence set $\mathcal{F}_{\mathsf{A}}$ for every feasible attack A. By irreducibility, $\mathcal{F}$ cannot contain locations outside the union of these $\mathcal{F}_{\mathsf{A}}$ sets. □

Example 6.19. In compliance with the above Lemma 6.18, in the SB program (Figure 1.1) the only irreducible valid fence set is $\mathcal{F} := \mathcal{F}_{\mathsf{A}} \cup \mathcal{F}_{\mathsf{A}'} = \{(1, q_1), (2, q_2)\}$.

Lemma 6.18 is useful for fence computation since optimal fence sets are always irreducible. All irreducible eliminating fence sets for attacks can be constructed by an exhaustive search through all selections of locations involved in the attack. For each candidate fence set, to judge whether it eliminates the attack, we check SC reachability in the instrumented program as described in Sections 6.2 and 6.3.

Note that this search may raise an exponential number of reachability queries. In practice this rarely constitutes a problem. First, attacks seldom have large sets of involved locations, so the number of candidates is small. Second, the reachability checks can be avoided if a candidate fence set covers all the paths from $\mathsf{dst}(\mathsf{instr}_{\mathsf{store}})$ to $\mathsf{src}(\mathsf{instr}_{\mathsf{load}})$.

6.5.2 Computing an Optimal Valid Fence Set

In order to decide which sets $\mathcal{F}_{\mathsf{A}}$ must be included into the optimal valid fence set, we set up and solve a 0/1-integer linear programming (ILP) problem (6.12)–(6.14). Here, 0/1 means the variables are restricted to have only values 0 or 1. The optimal solutions correspond to optimal valid fence sets.

Consider an attack A. Let $\mathcal{F}_1 \dots \mathcal{F}_n$ be the irreducible eliminating fence sets for this attack. For each fence set we introduce a variable $x_{\mathcal{F}_i}$; if the value of this variable is 1 in the solution, then $\mathcal{F}_i$ must be included into the optimal valid fence set. For each attack we include an inequality requiring that at least one eliminating fence set for this attack is chosen:

$$\sum_{1 \le i \le n} x_{\mathcal{F}_i} \ge 1 \tag{6.12}$$

Moreover, for each fence location $f \in \mathsf{FENCES}$ we create a variable x_f; if the value of this variable is 1 in the solution, then f belongs to the optimal valid fence set. The following equation requires that if a fence set is included into a solution, all its member fence locations are included as well.

$$\sum_{f \in \mathcal{F}_i} x_f \ge |\mathcal{F}_i| x_{\mathcal{F}_i} \tag{6.13}$$

Finally, the minimized function is the total cost function:

$$\sum_{f \in \mathsf{FENCES}} \mathcal{C}(f) x_f \to \min \tag{6.14}$$

An optimal solution x^* of the resulting 0/1-ILP denotes the fence set $\mathcal{F}(x^*) := \{f \in \mathsf{FENCES} \mid x^*_f = 1\}$. By construction of the ILP, the following theorem holds.

Theorem 6.20. *$\mathcal{F}(x^*)$ is an optimal valid fence set.*

Chapter 7

The Trencher Tool

We have implemented the algorithms from Chapter 6 in a tool called Trencher. Trencher is able to analyze multithreaded programs written in a simple, however, Turing-complete assembler-like programming language. The tool implements the reduction of robustness to SC reachability described in Section 6.2 and the fence insertion algorithm from Section 6.5. Trencher's source code and user documentation are available online at `https://github.com/yegord/trencher`.

In Section 7.1 we discuss certain design decisions and optimizations that made the tool fast. In Section 7.2 we presents the results of experiments that we conducted using Trencher.

7.1 Making It Fast

The fundamental subtask in checking and enforcing robustness is to decide whether a given attack has a TSO witness. Trencher solves this subtask by instrumenting the input program as described in Section 6.2 and performing an SC reachability query in the instrumented program. The instrumentation step takes linear time in the size of the input program. Consequently, the dominant in the running time is the PSPACE-hard SC reachability analysis, which makes it the primary target for performance optimizations.

The first version of Trencher [20] used SPIN [44] as a back-end SC reachability checker. Trencher translated a reachability query into a program in Promela language. SPIN took this program as an input and produced a C source code of the verifier (pan). Next, this source code was compiled by a C compiler. Finally, running the model checker produced the answer to the reachability query.

Using an off-the-shelf model checker allowed Trencher to benefit from the state space reductions and further optimizations already implemented in SPIN. However, this design decision came with a disadvantage: experiments [20] showed that most of the time spent on checking robustness was consumed by the C compiler. The costs of parsing system headers, verifier's source code, compilation, optimization, code generation, linking outweighed the time spent by the verifier in order to answer the reachability query.

As an attempt to reduce the running times, we implemented a custom SC reachability checker as a part of Trencher. The reachability checker is simply a depth-first search algorithm, however, applied to a reduced state space. Before

going into details on how state space reduction works, we need to define more exactly, what kind of a reachability query the checker must answer.

7.1.1 SC Semantics with Locks

While describing TSO semantics in Section 2.4.2 we intentionally omitted `lock`ed instructions to keep the constructions simple. The instructions can be treated similar to `mfence`, with all the results continuing to hold. The SC model checker bears all the burden of dealing with these instructions. In this subsection we complete the definition of SC, in the presence of `lock`ed instructions.

Following [72], we extend the TSO set of commands with `lock` and `unlock` instructions:

$$\begin{aligned}\langle cmd\rangle \;::=\; & \langle reg\rangle \leftarrow \texttt{mem[}\langle expr\rangle\texttt{]} \mid \texttt{mem[}\langle expr\rangle\texttt{]} \leftarrow \langle expr\rangle \\ \mid\; & \langle reg\rangle \leftarrow \langle expr\rangle \mid \texttt{assume(}\langle expr\rangle\texttt{)} \\ \mid\; & \texttt{mfence} \mid \texttt{lock} \mid \texttt{unlock}\end{aligned}$$

The `lock` instruction acquires an exclusive lock on memory, `unlock` releases it. If a thread owns a lock on memory, the other threads cannot access memory, i.e., their loads and stores block. Consequently, an atomic increment of the value at address r_{a} can be implemented with the help of `lock` and `unlock` as follows:

$$q_1 \xrightarrow{\texttt{lock}} q_2 \xrightarrow{r\leftarrow\texttt{mem}[r_{\mathsf{a}}]} q_3 \xrightarrow{\texttt{mem}[r_{\mathsf{a}}]\leftarrow r+1} q_4 \xrightarrow{\texttt{unlock}} q_5.$$

Formally, we define the semantics of a program $\mathcal{P}$ under SC with locks as $X_{\mathsf{scl}}(\mathcal{P}) := (S_{\mathsf{scl}}, \mathrm{E}_{\mathsf{scl}}, \Delta_{\mathsf{scl}}, s_{\mathsf{scl}0}, F_{\mathsf{scl}})$. A state $s \in S_{\mathsf{scl}}$ is a tuple $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{lock})$, where counter configuration $\mathsf{sn}\colon \mathsf{TID} \to \mathbb{N}$ gives, for each thread, the id that will be assigned to the next instruction executed in this thread, $\mathsf{pc}(\mathsf{tid}) \in Q_{\mathsf{tid}}$ gives the control state of the thread tid, $\mathsf{mem}\colon \mathsf{TID}\times\mathsf{REG}\cup \mathsf{ADDR} \to \mathsf{DOM}$ gives, for each address and each register, the value stored at this address or in this register, and $\mathsf{lock} \in \mathsf{TID}\cup\{\bot\}$ gives the thread currently owning the memory lock. The initial state is $s_{\mathsf{scl}0} := (\mathsf{sn}_0, \mathsf{pc}_0, \mathsf{mem}_0, \mathsf{lock}_0)$, where $\mathsf{sn}_0 := \lambda\mathsf{tid}.1$, all the control states are initial: $\mathsf{pc}_0(\mathsf{tid}) := q_{\mathsf{tid}\,0}$, the memory is filled with zeroes: $\mathsf{mem}_0(\mathsf{a}) := 0$ for all $\mathsf{a} \in \mathsf{TID} \times \mathsf{REG} \cup \mathsf{ADDR}$, and nobody owns the memory lock: $\mathsf{lock}_0 := \bot$. All states are final: $F_{\mathsf{scl}} := S$.

The SC transition relation Δ_{scl} consists of all the transitions defined by the rules in Table 7.1. The first two rules describes loads and stores which can be executed only if the memory lock is held by the current thread or not held at all. The third and the forth rules describe the local assignment and conditional, they are similar to those from Table 2.1. A memory fence, the fifth rule, is a no-op under SC. The sixth rule says that a lock can be taken only if nobody holds it already. The seventh rule defines that a lock can be released by the thread that holds it.

7.1.2 Restricted SC Semantics with Locks

The idea of the state space reduction implemented in Trencher is as follows. Assume the running thread has executed a local assignment, a condition, or a memory fence. The other threads cannot observe the effects of this transition, because the memory state was not changed. Therefore, it makes sense to

$$\frac{\mathsf{cmd} = r \leftarrow \texttt{mem}[e], \quad \mathsf{a} := \widehat{e}, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[r := \mathsf{mem}(\mathsf{a})], \mathsf{lock})}$$

$$\frac{\mathsf{cmd} = \texttt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}}, \quad \mathsf{a} := \widehat{e_{\mathsf{a}}}, \quad \mathsf{v} := \widehat{e_{\mathsf{v}}}, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[\mathsf{a} := \mathsf{v}], \mathsf{lock})}$$

$$\frac{\mathsf{cmd} = r \leftarrow e}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \widehat{e}], \mathsf{lock})}$$

$$\frac{\mathsf{cmd} = \texttt{assume}(e), \quad \widehat{e} \neq 0}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{lock})}$$

$$\frac{\mathsf{cmd} = \texttt{mfence}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{lock})}$$

$$\frac{\mathsf{cmd} = \texttt{lock}, \quad \mathsf{lock} = \bot}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{tid})}$$

$$\frac{\mathsf{cmd} = \texttt{unlock}, \quad \mathsf{lock} = \mathsf{tid}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \bot)}$$

Table 7.1: Transition rules for SC with locks, assuming $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{lock})$, $\mathsf{pc}(\mathsf{tid}) = q$, an instruction $\mathsf{instr} = q \xrightarrow{\mathsf{cmd}} q'$, $\mathsf{pc}' := \mathsf{pc}[\mathsf{tid} := q']$, and $\mathsf{sn}' := \mathsf{sn}[\mathsf{tid} := \mathsf{sn}(\mathsf{tid}) + 1]$.

$$\frac{\mathsf{cmd} = r \leftarrow \texttt{mem}[e], \quad \mathsf{a} := \widehat{e}, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[r := \mathsf{mem}(\mathsf{a})], \mathsf{lock}, \bot)}$$

$$\frac{\mathsf{cmd} = \texttt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}}, \ \mathsf{a} := \widehat{e_{\mathsf{a}}}, \ \mathsf{v} := \widehat{e_{\mathsf{v}}}, \ \mathsf{lock} \in \{\mathsf{tid}, \bot\}, \ \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr},\mathsf{a})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[\mathsf{a} := \mathsf{v}], \mathsf{lock}, \bot)}$$

$$\frac{\mathsf{cmd} = r \leftarrow e, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{tid}, r) := \widehat{e}], \mathsf{lock}, \mathsf{tid})}$$

$$\frac{\mathsf{cmd} = \texttt{assume}(e), \quad \widehat{e} \neq 0, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{lock}, \mathsf{tid})}$$

$$\frac{\mathsf{cmd} = \texttt{mfence}, \quad \mathsf{lock} \in \{\mathsf{tid}, \bot\}, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{lock}, \mathsf{tid})}$$

$$\frac{\mathsf{cmd} = \texttt{lock}, \quad \mathsf{lock} = \bot, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{tid}, \mathsf{tid})}$$

$$\frac{\mathsf{cmd} = \texttt{unlock}, \quad \mathsf{lock} = \mathsf{tid}, \quad \mathsf{fav} \in \{\mathsf{tid}, \bot\}}{s \xrightarrow{(\mathsf{tid},\mathsf{sn}(\mathsf{tid}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \bot, \bot)}$$

Table 7.2: Transition rules for restricted SC with locks, assuming $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{lock}, \mathsf{fav})$, $\mathsf{pc}(\mathsf{tid}) = q$, an instruction $\mathsf{instr} = q \xrightarrow{\mathsf{cmd}} q'$, $\mathsf{pc}' := \mathsf{pc}[\mathsf{tid} := q']$, and $\mathsf{sn}' := \mathsf{sn}[\mathsf{tid} := \mathsf{sn}(\mathsf{tid}) + 1]$.

continue executing the current thread, until it performs a memory access. Similarly, if a thread has acquired a memory lock, the other threads cannot access memory at all. Therefore, it does not make sense to do a context switch, until the thread releases the memory lock. These observations lead to the *restricted SC semantics with locks*.

Formally, the semantics is $X_{\mathsf{sclr}}(\mathcal{P}) := (S_{\mathsf{sclr}}, \mathrm{E}_{\mathsf{sclr}}, \Delta_{\mathsf{sclr}}, s_{\mathsf{sclr}0}, F_{\mathsf{sclr}})$. A state $s \in S_{\mathsf{sclr}}$ is a tuple $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{lock}, \mathsf{fav})$, where $\mathsf{fav} \in \mathsf{TID} \cup \{\bot\}$ determines which thread can execute the next instruction, $\bot$ standing for any. We call the thread fav the *favourite* thread. In the initial state $s_{\mathsf{sclr}0} := (\mathsf{sn}_0, \mathsf{pc}_0, \mathsf{mem}_0, \mathsf{lock}_0, \mathsf{fav}_0)$ any thread can execute the next instruction: $\mathsf{fav}_0 := \bot$. The other components of the tuples s and $s_{\mathsf{sclr}0}$ are defined as in the previous subsection.

The SC transition relation Δ_{sclr} consists of all the transitions defined by the rules in Table 7.2. The rules extend those from Table 7.1 by the additional requirement: in order for a thread to be able to execute an instruction, no other thread must be a favourite thread or hold the lock. Loads, stores, and `unlock` reset the favourite thread to $\bot$ and make a context switch possible. All other instructions make the current thread the favourite, i.e., forbid context switches.

The following theorem states that state reachability under SC with locks is equivalent to state reachability under restricted SC with locks.

Theorem 7.1. *Fix a program $\mathcal{P}$, a thread tid, and a control state q. The thread can reach the control state under SC with locks iff it can reach the control state under restricted SC with locks.*

Proof. The implication from right to left is trivial. Consider the implication from left to right. Assume thread tid can reach control state q via computation σ:

$$s_{\mathsf{scl}0} \xrightarrow{\sigma} (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{lock}) \text{ with } \mathsf{pc}(\mathsf{tid}) = q.$$

Without loss of generality we can assume that there is no shorter computation, via which thread tid can reach control state q.

In the proof we call an event e *local* if $\mathsf{instr}(\mathsf{e})$ is a local assignment, condition, memory fence, or lock. We call the event e *non-local* if $\mathsf{instr}(\mathsf{e})$ is a load, a store, or unlock. It is easy to check by case consideration that if $\sigma = \sigma_1 \cdot \mathsf{e}_1 \cdot \mathsf{e}_2 \cdot \sigma_2 \in \mathsf{C}_{\mathsf{scl}}(\mathcal{P})$, e_1 is a local event, and $\mathsf{tid}(\mathsf{e}_1) \neq \mathsf{tid}(\mathsf{e}_2)$, we can move e_1 to the right of e_2 without changing the trace: $\sigma' := \sigma_1 \cdot \mathsf{e}_2 \cdot \mathsf{e}_1 \cdot \sigma_2 \in \mathsf{C}_{\mathsf{scl}}(\mathcal{P})$ and $T(\sigma) = T(\sigma')$. In other words, local events are right movers in the sense of [61].

Since σ is the shortest computation, if we consider the last event in σ belonging to a particular thread, this event is a non-local event (except when the thread is thread tid). Indeed, if the event would be a local event, it would represent a computation whose result is never used in other threads. Consequently, we could remove this event and obtain a shorter computation, via which thread tid could reach control state q. Similarly, one can show that no thread (except for possibly tid) executes `lock` without executing `unlock` later, i.e., $\mathsf{lock} \in \{\mathsf{tid}, \bot\}$ when q is reached by thread tid.

If we move all local events to the right as much as possible (i.e., to the next non-local event of the same thread or to the end of the computation), we obtain a computation $\sigma'' := \sigma_1 \cdot \mathsf{e}_1 \cdot \sigma_2 \cdot \mathsf{e}_2 \cdots \sigma_n$. Events $\mathsf{e}_1 \ldots \mathsf{e}_{n-1}$ are non-local events. Events in σ_j, $j \in [1..n-1]$ are local events. Events in $\sigma_j \cdot \mathsf{e}_j$, $j \in [1..n-1]$ belong to the same thread. Events in σ_n also belong to the same thread, namely, thread tid. As noted earlier, each reordering preserves the trace. Consequently, $T(\sigma'') = T(\sigma)$ and thread tid reaches control state q via σ''.

It is easy to see that $\sigma'' \in \mathsf{C}_{\mathsf{sclr}}(\mathcal{P})$. Indeed, context switches happen only after non-local events. Moreover, context switches happen only at states with $\mathsf{lock} = \bot$, otherwise, the thread to which the switch happens would not be able to produce the non-local event e_j. □

7.1.3 Live Variable Optimization

A variable is commonly called *live* at a control state q if its value at this control state can be read on paths starting in q. For example, in

$$q_1 \xrightarrow{r \leftarrow \texttt{mem}[x]} q_2 \xrightarrow{\texttt{assume}(r \neq 0)} q_1$$

the register r is live at control state q_2, but not at q_1. Clearly, one does not need to keep the exact value of a register if it is not live at the current control state. This is exactly the idea of the live variable optimization.

Trencher implements classic live variable analysis, as described, e.g., in [50]. While exploring the state space and computing the destination state of a transition, Trencher resets values of dead (not live) registers in the destination state to zero.

Similar optimizations are implemented in multiple model checkers, including SPIN [45], XMC [36], Bandera [31], IF [23], Bebop [18].

7.1.4 Atomic Instructions

Last but not least, Trencher's model checker supports atomic instructions, i.e., instructions labeled by multiple commands that must be executed in one step. For example, an atomic instruction

$$q_1 \xrightarrow{r_1 \leftarrow \texttt{mem}[x] \cdot \texttt{assume}(r_1 = r_2) \cdot \texttt{mem}[x] \leftarrow r_3} q_2$$

implements compare-and-set which either succeeds or blocks.

Trencher translates sequences of instructions generated during instrumentation of a single instruction to atomic instructions, thus even more reducing the number of possible interleavings.

The Promela language used by SPIN [44] provides a similar construct — `atomic` sequence of statements. Such a sequence is guaranteed to be executed atomically, however, only if no statement in the sequence blocks. If it does, a context switch happens. Trencher's atomic statements, on the contrary, guarantee atomicity in all cases: an instruction is either executed fully in one step, or it is not executed at all.

7.2 Experiments

We evaluated Trencher on a set of examples modelling various concurrent algorithms and data structures. In this section we present results of the evaluation and discuss the results. We start with the description of examples.

7.2.1 Examples

The first class of examples on which we tested Trencher consists of mutual exclusion algorithms using shared variables. Naive implementations of the algorithms turn out to be non-robust against TSO and do not guarantee mutual exclusion under this memory model. Correct versions of the algorithms require memory fences and are robust against TSO. We study robust and non-robust versions of classic Dekker's [35], Peterson's [74], Burns' [27] protocols for two threads entering and leaving a critical section in a cycle, and the Lamport's fast mutex [57] for three threads. We also checked CLH and MCS list-based queue locks [42] which rely on atomic compare-and-set and are robust.

The second class of examples constitute concurrent data structures. The first data structure is the work stealing queue (WSQ) from an implementation of Cilk 5 programming language [38]. A WSQ provides three operations: push, pop, and steal. Push and pop operations have the usual meaning for queues, however, can only be performed by one thread. The steal operation has the semantics of a pop, but can be performed by any number of threads concurrently. We consider two programs using the work stealing queue: one that does push and pop in different threads, and one that uses the queue properly. The second data structure that we consider is a concurrent lock-free stack implementation [42].

Examples from the third class model algorithms found in real systems code. We consider the buggy C++ Parker class implementing

N	Example	Thr	St	Tr	RQ1	RQ2	RQ3	F	CPU	Real
1	Dekker (not fenced)	2	24	30	24	38	43	4	109	43
2	Dekker (fenced)	2	28	34	30	0	0	0	0	0
3	Peterson (not fenced)	2	14	18	1	12	8	2	19	8
4	Peterson (fenced)	2	16	20	12	0	0	0	0	0
5	Burns (not fenced)	2	11	14	1	4	12	3	7	3
6	Burns (fenced)	2	17	19	8	0	0	0	0	0
7	Lamport (not fenced)	3	33	36	9	15	12	6	1009	280
8	Lamport (fenced)	3	39	42	27	0	0	0	0	0
9	CLH Lock	3	42	41	60	10	0	0	28	9
10	MCS Lock	2	54	58	64	8	0	0	18	5
11	Cilk WSQ (incorrect use)	5	80	79	137	12	3	3	12399	3229
12	Cilk WSQ (correct use)	3	73	72	133	16	0	0	18	4
13	Lock-free stack	4	46	50	14	0	0	0	0	0
14	Parker (not fenced)	2	9	8	0	1	1	1	0	0
15	Parker (fenced)	2	10	9	2	0	0	0	0	0
16	NBW+Spinlock	4	45	45	26	4	0	0	18	7

Table 7.3: Examples and testing results.

`java.util.concurrent.LockSupport` in Sun JVM [34] and the non-blocking write protocol + spinlock example studied in Section 8 of [71].

Table 7.3, the left half, provides qualitative characteristics of the examples. Columns Thr, St, and Tr give respectively the number of threads, states, and transitions in the examples.

7.2.2 Results

We ran the fence insertion algorithm implemented in Trencher on examples from Table 7.3 using a 4-core machine equipped with Intel(R) Core(TM) i5 CPU M 560 @ 2.67GHz. The results of the testing are presented in Table 7.3, the right part. The RQ1 column gives the number of reachability queries that were solved without actually performing a full-fledged reachability check. Trencher is capable to infer that an attack does not have a TSO witness if the attack's store and the attack's load are separated by memory fences, and avoids a real reachability query in this case. The RQ2 column gives the number of full-fledged reachability checks that returned a negative answer. The RQ3 column gives the number of checks that showed that the goal state is reachable. So, the total number of queries is RQ1+RQ2+RQ3. The F column shows the number of fences inserted by Trencher. The CPU and Real columns show the CPU and wall-clock time, in milliseconds, spent by Trencher on each example (the minimum values out of four runs).

To estimate the effects of the optimizations described in Section 7.1, we ran Trencher on the same set of examples with some of the optimizations disabled. We performed the same tests using SPIN as a model checker. The results of these runs are shown in Figure 7.1 (CPU times) and Figure 7.2 (number of states visited during reachability checks). In Figure 7.1 we excluded the time taken by SPIN to generate the verifier's source code, as well as the time taken by the C compiler to build the verifier. The time taken by the C compiler constitutes

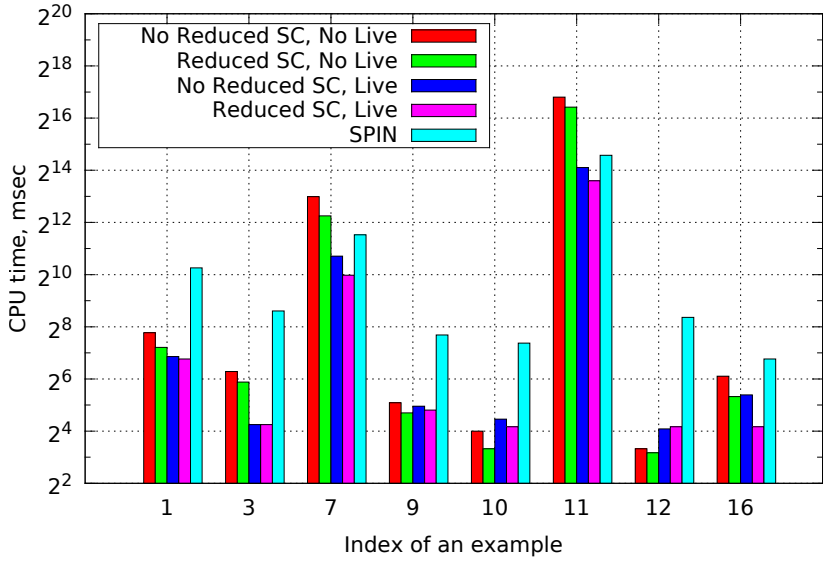

Figure 7.1: Time spent by Trencher on computing minimal fence set for the examples from Table 7.3 with various combinations of optimizations described in Section 7.1. Reduced SC stands for reduced SC semantics with locks. Live stands for live variable optimization. SPIN bars show the time spent by Trencher using SPIN as a model checker (only time spent by Trencher and the verifier is taken into account; time used for generation and compilation of the verifier is left out for fairness; the verifier was compiled by Clang 3.5 with `-O2`). Only examples whose analysis with all optimizations on took more that 10ms are shown.

the major part of the total time, see Appendix B for details.

7.2.3 Discussion

Trencher could show robustness of the program that used Cilk WSQ correctly (performed push and pop in the same thread). Notably, Trencher detected non-robustness of the example that performed push and pop in different threads. The tool could detect non-robustness in the Parker class that leads to wrong behaviors and verify the spinlock example.

Interestingly, Dekker's and Burns' protocols required respectively 2 and 1 more fences for robustness than it is actually necessary for ensuring mutual exclusion. We explain the reasons on the example of Dekker's algorithm, Burn's algorithm has a similar issue. The SB program (Figure 1.1) emulates the first stage of the Dekker's protocol: a thread signals that is wants to enter a critical section; if the other thread did not signal that it wants to enter the critical section too, the first thread is going to enter it. Fences at states q_1 are required under TSO to prevent both threads from reading zero. In a real implementation, if a thread has read 1, it signals that it does not want to enter the critical section (writes 0 to its variable) and starts polling on the other thread's variable, waiting until it becomes zero. In other words, it essentially executes the SB code, with

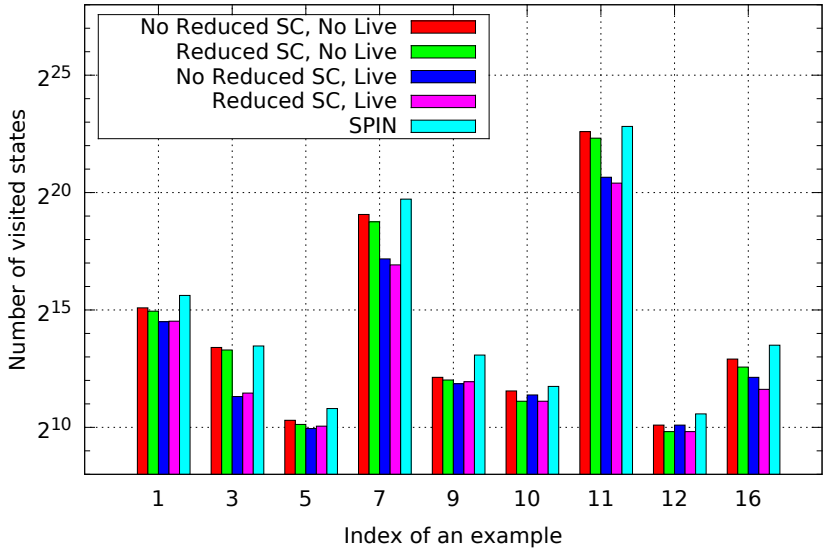

Figure 7.2: Total number of states visited by the reachability checkers while computing minimal fence sets for the examples from Table 7.3. For the description of the legend, refer to Figure 7.1. Only examples where the number of visited states is greater than 1024 are shown.

the only difference that it stores 0 instead of 1. As in SB, this leads to happens-before cycles, which are eliminated by two additional fences, one per each thread. These fences are not needed for mutual exclusion: without them the other thread may just wait a bit longer until it learns that the first thread no longer wants to enter the critical section.

We would like to note that the analysis of robust examples is particularly fast. Most of the reachability queries are answered without performing a full reachability check. All considered mutual exclusion algorithms, except for CLH and MCS locks that use atomic compare and swap, did not require such checks at all. In CLH and MCS locks their number is less than 20% of the total. Also, Trencher is capable of executing independent reachability queries in parallel. This ability shows its effect on relatively complex examples, where the real (wall-clock) time constitutes 1/4 to 1/3 of the CPU time (on a 4-core machine).

Comparison of reachability checkers in Figure 7.1 and Figure 7.2 shows that the optimizations described in Section 7.1 do matter. Use of reduced SC semantics with locks and live variable optimization can decrease the state space by a factor of up to four (see examples 3, 7, 11), saving CPU time accordingly. Interestingly, the number of states visited by the SPIN verifier is slightly larger than the number of states visited by Trencher's reachability checker without any optimizations. This might be caused by the fact that atomic sequences of instructions in SPIN are actually not atomic, i.e., they are atomic only if no statement in the sequence blocks, which is often not the case in the instrumented programs. The necessity of running an external program might be the cause of the additional overhead seen in the CPU times chart (Figure 7.1) for Trencher using SPIN.

Chapter 8

Robustness against Partitioned Global Address Space

Partitioned Global Address Space (PGAS) is a parallel programming model for the development of high-performance software for clusters. It provides a global address space partitioned among the cluster nodes (Figure 8.1). Programs written in languages like C, C++, and Fortran can access the global memory by means of PGAS APIs, such as SHMEM [29], ARMCI [68], GASNet [19], GPI [63], and GASPI [39]. HPC languages like UPC [30], Titanium [43], and Co-Array Fortran [70] support the PGAS programming model directly.

PGAS programs are typically written in the *single instruction, multiple data* (SPMD) paradigm. At run time, an SPMD program consists of multiple processes executing the same code on different nodes. Each process is identified by its *rank*, which is essentially the index of the node it runs on. PGAS APIs and languages typically provide functions that allow a process to learn its rank and the total number of processes (nodes).

A key feature of PGAS is the emphasis on one-sided communication: a process running on one node may directly read and write the memory of the process running on a remote node, without any synchronization with the remote process. This is contrary to the message passing paradigm, which requires explicit synchronization of the sender and the receiver. One-sided communication can

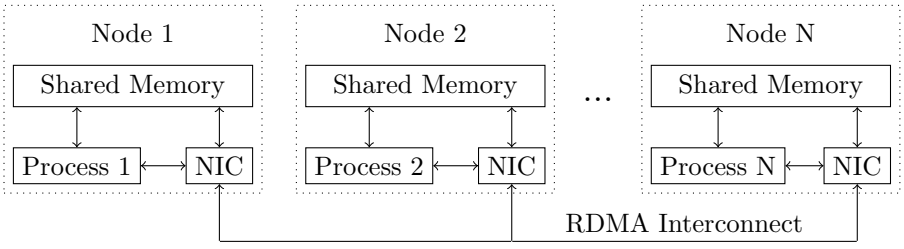

Figure 8.1: PGAS model.

```
1  int x = 1, y = 0;
2  int main() {
3      int myRank = getMyRank();
4      int nodeCount = getNodeCount();
5      int rightNeighborRank = myRank() % nodeCount + 1;
6
7      write(&x, rightNeighborRank, &y, sizeof(x), Queue0);
8      barrier();
9      assert(y == 1);
10 }
```

Figure 8.2: OneToOne program.

be efficiently implemented on top of interconnect hardware featuring remote direct memory access (RDMA). A process can request an RDMA-enabled network interface controller to copy a block of memory from/to a given node, and the controller will perform the transfer on its own, without involving the CPU or the operating system on neither of the nodes.

Absence of mandatory (and often unnecessary) synchronization leads to higher performance of PGAS applications. However, this high performance comes at the cost of putting the burden of correct synchronization onto the programmer, which must account for the possible reorderings and delays of memory accesses introduced by the hardware. Insufficient synchronization leads to subtle bugs that are hard to reproduce, debug, and fix.

Example 8.1. Consider the OneToOne program shown in Figure 8.2. The program transfers the value 1 contained in variable x on the local node to the variable y on a remote node. It works as follows. When started, each process of this program learns about its rank and the total number of nodes (lines 3–4). Next, it computes the rank of the remote node where the value must be transferred, line 5. Finally, it requests the PGAS API to copy a block of memory occupied by variable x to the memory block occupied by variable y on the remote node, line 7. The API implementation, in turn, asks the hardware to perform an RDMA transfer between the nodes. After calling `write()`, the processes synchronize: the `barrier()` function returns only after all processes have entered it. This means, if a process has reached the line 9, all other processes have called `barrier()`, and therefore executed `write()`. Nonetheless, the assertion at line 9 can fail, because the process did not synchronize with the hardware: the remote write, although issued, might have been not yet completed.

What is a correctly synchronized PGAS program? In this chapter we choose and study robustness (Section 2.8), previously considered for CPU memory models, as a notion of correct synchronization for programs using PGAS APIs. First, we define a formal model for describing the semantics of PGAS programs, presented in Section 8.1. The model reflects the main features of popular real-world APIs such as SHMEM, ARMCI, GASNet, GPI, and GASPI. In Section 8.2 we define robustness for programs in this model.

Second, we devise an algorithm for checking robustness of PGAS programs.

Similar to the previous chapters, we use the following two ideas. First, in Section 8.3 we show that if a program is not robust, there is a normal-form computation that demonstrates non-robustness. Next, in Section 8.4 we show how to detect these computations by a multiheaded automaton. Essentially, we reduce robustness against PGAS to an emptiness for a multiheaded automaton. This leads to the main result of this chapter: robustness against PGAS is PSPACE-complete.

Related work An alternative notion of synchronization correctness is data race freedom [4]. Park et al. [73] proposed a testing framework for data race detection and implemented it for the UPC language. However, the analysis of the NAS Parallel Benchmarks [69] carried out by these authors showed that a significant portion of the detected data races are actually not harmful. Several examples from [73] show that harmful data races (like in the `knapsack` example) lead to non-robustness, while benign data races (like in the examples `NPB 3.3 BT` and `SP`) do not.

8.1 PGAS Semantics

In this section we consider the most popular PGAS APIs, highlight their differences and similarities, and devise a unified programming model that allows to model the reorderings of remote memory accesses allowed by these APIs.

8.1.1 PGAS APIs

All popular PGAS APIs favour the SPMD programming model: the running program consists of multiple processes executing the same code on different nodes. All APIs provide functions allowing a process to know its rank and the total number of processes. The processes can access the global partitioned address space. Local partition of the global space can be accessed directly. The remote partitions, located on the remote nodes, can be accessed using API calls. The APIs differ in the guarantees about the ordering and synchroneity of the remote memory accesses.

In SHMEM [29] data transfers are performed via `shmem_get` and `shmem_put` families of routines. The `get` routines copy the data from a remote node to the local one and are blocking, i.e., return only when the data is actually copied and is locally available. The `put` routines copy the data from a local node to the remote one. They are non-blocking, but return only after the data is copied from the local buffer and the buffer can be reused. Ordering of put operations to the same node can be enforced using `shmem_fence` routine, ordering of puts across all the nodes — by `shmem_quiet`. However, there is no simple way to ensure that the data of a given put operation is fully written to the remote node.

ARMCI [68] features blocking (`ARMCI_Get`, `ARMCI_Put`) as well non-blocking (`ARMCI_NbGet`, `ARMCI_NbPut`) read and write operations. The non-blocking variants return a handle (an opaque value used to identify the issued operation). This handle can be passed to the `ARMCI_Wait` routine that will block until the respective operation is completed. A get operation is considered completed when the requested data is locally available. A put operation is considered completed when the data was sent (but not necessarily arrived). To ensure that

the data has arrived, one has to call `ARMCI_Fence` or `ARMCI_FenceAll`. Operations to the same remote node are executed in the ordered in which they were issued. Operations to different nodes can complete in any order.

GASNet [19] is the library used for implementing PGAS Languages: UPC [30], Titanium [43], Co-Array Fortran [70]. Like ARMCI, it provides blocking (`gasnet_get`, `gasnet_put`) and non-blocking (`gasnet_get_nb`, `gasnet_put_nb`) versions of read and write operations. The non-blocking operations return a handle that can be passed, e.g., to `gasnet_wait_syncnb` to wait until the operation with the given handle is completed. The order in which non-blocking operations complete is intentionally left unspecified.

GPI [63] and GASPI [39] provide functions only for non-blocking data transfers: `readDmaGPI`, `writeDmaGPI` and `gaspi_read`, `gaspi_write`. These routines copy the data between the specified local and remote memory blocks. Together with the addresses and sizes of the memory blocks, the user also specifies a queue id. GPI and GASPI provide several queues, and the copy operation is added to the queue with a given id to be executed in background. In GPI it is guaranteed that the operations from the same queue to the same remote node are executed in the order in which they were added, there are no ordering constrains for operations from different queues or to different nodes. GASPI does not give any guarantees about the ordering of operations. In order to block until all the operations in a certain queue are (locally and remotely) completed, `waitDmaGPI` and `gaspi_wait` functions are used.

Summing up, in a uniform PGAS programming model it should be possible to

- perform blocking and non-blocking data transfers,
- assign a non-blocking operation a handle or a queue id,
- wait for completion of an individual operation or of all operations in a given queue,
- enforce ordering between operations.

We define a core model for PGAS that supports all these features. Our model provides only non-blocking remote reads and writes with explicit queues, but is flexible enough to accommodate all the above idioms.

8.1.2 PGAS Model

We again define PGAS programs and their semantics in terms of automata, similar to Section 2.3. A *program* is an automaton $\mathcal{P} := (Q, \mathsf{CMD}, \mathcal{I}, q_0, Q)$ with a finite set of control states Q, all of them being final, initial state q_0, and a set of transitions $\mathcal{I}$ called *instructions* and labeled with *commands* CMD defined below.

We extend the set of commands CMD with the remote read and write API calls `read` and `write`, and the barrier command `barrier`. Altogether,

$$
\begin{aligned}
\langle cmd \rangle ::= {} & \langle reg \rangle \leftarrow \texttt{mem[}\langle expr \rangle\texttt{]} \mid \texttt{mem[}\langle expr \rangle\texttt{]} \leftarrow \langle expr \rangle \\
\mid {} & \langle reg \rangle \leftarrow \langle expr \rangle \mid \texttt{assume(}\langle expr \rangle\texttt{)} \\
\mid {} & \texttt{read(}\langle \textit{local-addr} \rangle\texttt{,}\langle rank \rangle\texttt{,}\langle \textit{remote-addr} \rangle\texttt{,}\langle \textit{queue-id} \rangle\texttt{)} \\
\mid {} & \texttt{write(}\langle \textit{local-addr} \rangle\texttt{,}\langle rank \rangle\texttt{,}\langle \textit{remote-addr} \rangle\texttt{,}\langle \textit{queue-id} \rangle\texttt{)} \\
\mid {} & \texttt{barrier}
\end{aligned}
$$

where all the undefined non-terminals are expressions $\langle expr \rangle$.

We assume that a program comes with the address domain ADDR, data domain DOM, and queue domain QUE. To avoid special cases, we assume that $\mathsf{ADDR} = \mathsf{DOM} = \mathsf{QUE}$. In this chapter we mainly analyze programs which run on a fixed number of nodes N. We denote a program together with the number of nodes it runs on as $(\mathcal{P}, N)$. As the *size of program* $(\mathcal{P}, N)$ we take the sum of N and the size of $\mathcal{P}$ (as defined in Section 2.3).

At run time, there is a process on each node that executes the code of program $\mathcal{P}$. We will identify each process with its rank from $\mathsf{RANK} := [1..N]$. For modeling purposes, one may assume there are special expressions that let a process learn its rank and the total number of processes N.

The semantics of a PGAS program $(\mathcal{P}, N)$ is an automaton $X_{\mathsf{pgas}}(\mathcal{P}, N) := (S_{\mathsf{pgas}}, \Delta_{\mathsf{pgas}}, \mathrm{E}, s_{\mathsf{pgas}_0}, F_{\mathsf{pgas}})$. A state $s \in S_{\mathsf{pgas}}$ is a tuple $s = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$, where counter configuration $\mathsf{sn}\colon \mathsf{RANK} \to \mathbb{N}$ gives, for each thread, the id that will be assigned to the next instruction executed in it, control configuration $\mathsf{pc}\colon \mathsf{RANK} \to Q$ maps each process to its current control state, memory configuration $\mathsf{mem}\colon \mathsf{RANK} \times (\mathsf{REG} \cup \mathsf{ADDR}) \to \mathsf{DOM}$ maps each process to the values stored in each register and at each address, queue configuration $\mathsf{fa}\colon \mathsf{RANK} \times \mathsf{QUE} \to (\mathbb{N} \times \mathsf{RANK} \times \mathsf{ADDR} \times \mathsf{RANK} \times \mathsf{ADDR})^*$ maps each process to the remote read and write requests that were issued, and $\mathsf{fb}\colon \mathsf{RANK} \times \mathsf{QUE} \to (\mathbb{N} \times \mathsf{RANK} \times \mathsf{ADDR} \times \mathsf{DOM})^*$ contains the values to be transferred.

The initial state is $s_{\mathsf{pgas}_0} := (\mathsf{sn}_0, \mathsf{pc}_0, \mathsf{mem}_0, \mathsf{fa}_0, \mathsf{fb}_0)$, where for all ranks $\mathsf{r} \in \mathsf{RANK}$, registers and addresses $\mathsf{a} \in \mathsf{REG} \cup \mathsf{ADDR}$, and queue identifiers $\mathsf{q} \in \mathsf{QUE}$ we have $\mathsf{pc}_0(\mathsf{r}) := q_0$, $\mathsf{mem}_0(\mathsf{r}, \mathsf{a}) := 0$, and $\mathsf{fa}_0(\mathsf{r}, \mathsf{q}) := \varepsilon =: \mathsf{fb}_0(\mathsf{r}, \mathsf{q})$. The set of final states is $F_{\mathsf{pgas}} := \{(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb}) \in S_{\mathsf{pgas}} \mid \mathsf{fa}(\mathsf{r}, \mathsf{q}) = \varepsilon = \mathsf{fb}(\mathsf{r}, \mathsf{q}) \text{ for all } \mathsf{r} \in \mathsf{RANK},\ \mathsf{q} \in \mathsf{QUE}\}$. The semantics of commands ensures that queues can always be emptied, so acceptance with empty queues is not really a restriction.

The transitions from Δ_{pgas} are labeled with events E that we define together with the transitions. The transition relation Δ_{pgas} is the minimal relation defined by the rules from Table 8.1. When a process executes a remote write command, rule (write), a new item is added to a queue in fa. This item contains the source rank and source address from which the data will be copied, together with the destination rank and destination address to which the data will be copied. Eventually, the item is popped from the queue in fa, rule (popa), the value is read from the source address, and a new item is pushed into the corresponding queue in fb. The new item contains the destination rank and destination address, and the value that was read from the source address. Eventually, this item is popped from the queue, rule (popb), and the value is written to the destination address in the destination rank. Modeling two queue configurations yields a symmetry between remote writes and reads: a read can be interpreted as a write that comes upon request. Moreover, two queue configurations capture well the delays between request creation, reading of the data, and writing of the data.

The set of all computations of a PGAS program is $\mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N) := \mathcal{L}(X_{\mathsf{pgas}}(\mathcal{P}, N)) \subseteq \mathrm{E}^*$.

Example 8.2. Consider the PGAS program $(\mathsf{1to1}, 2)$ with the program code

$$\frac{\mathsf{cmd} = r \leftarrow e}{s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{r}, r) := \widehat{e}], \mathsf{fa}, \mathsf{fb})} \quad \text{(assign)}$$

$$\frac{\mathsf{cmd} = r \leftarrow \texttt{mem}[e_{\mathsf{a}}]}{s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr},(\mathsf{r},\widehat{e_{\mathsf{a}}}))} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{r}, r) := \mathsf{mem}(\mathsf{r}, \widehat{e_{\mathsf{a}}})], \mathsf{fa}, \mathsf{fb})} \quad \text{(load)}$$

$$\frac{\mathsf{cmd} = \texttt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}}}{s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr},(\mathsf{r},\widehat{e_{\mathsf{a}}}))} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}[(\mathsf{r}, \widehat{e_{\mathsf{a}}}) := \widehat{e_{\mathsf{v}}}], \mathsf{fa}, \mathsf{fb})} \quad \text{(store)}$$

$$\frac{\mathsf{cmd} = \texttt{assume}(e), \qquad \widehat{e} \neq 0}{s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}, \mathsf{fb})} \quad \text{(assume)}$$

$$\frac{\mathsf{cmd} = \texttt{read}(e_{\mathsf{a}}^{\mathsf{loc}}, e_{\mathsf{r}}^{\mathsf{rem}}, e_{\mathsf{a}}^{\mathsf{rem}}, e_{\mathsf{q}})}{\begin{array}{c} s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr},\widehat{e_{\mathsf{q}}})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}[(\mathsf{r}, \widehat{e_{\mathsf{q}}}) := \alpha'], \mathsf{fb}), \\ \alpha' := \mathsf{fa}(\mathsf{r}, \widehat{e_{\mathsf{q}}}) \cdot (\mathsf{sn}(\mathsf{r}), \widehat{e_{\mathsf{r}}^{\mathsf{rem}}}, \widehat{e_{\mathsf{a}}^{\mathsf{rem}}}, \mathsf{r}, \widehat{e_{\mathsf{a}}^{\mathsf{loc}}}) \end{array}} \quad \text{(read)}$$

$$\frac{\mathsf{cmd} = \texttt{write}(e_{\mathsf{a}}^{\mathsf{loc}}, e_{\mathsf{r}}^{\mathsf{rem}}, e_{\mathsf{a}}^{\mathsf{rem}}, e_{\mathsf{q}})}{\begin{array}{c} s \xrightarrow{(\mathsf{r},\mathsf{sn}(\mathsf{r}),\mathsf{instr},\widehat{e_{\mathsf{q}}})} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}[(\mathsf{r}, \widehat{e_{\mathsf{q}}}) := \alpha'], \mathsf{fb}), \\ \alpha' := \mathsf{fa}(\mathsf{r}, \widehat{e_{\mathsf{q}}}) \cdot (\mathsf{sn}(\mathsf{r}), \mathsf{r}, \widehat{e_{\mathsf{a}}^{\mathsf{loc}}}, \widehat{e_{\mathsf{r}}^{\mathsf{rem}}}, \widehat{e_{\mathsf{a}}^{\mathsf{rem}}}) \end{array}} \quad \text{(write)}$$

$$\frac{\mathsf{fa}(\mathsf{r}, \mathsf{q}) = (\mathsf{id}, \mathsf{r_s}, \mathsf{a_s}, \mathsf{r_d}, \mathsf{a_d}) \cdot \alpha}{\begin{array}{c} s \xrightarrow{(\mathsf{r},\mathsf{id},\mathsf{popa},(\mathsf{r_s},\mathsf{a_s}))} (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}[(\mathsf{r}, \mathsf{q}) := \alpha], \mathsf{fb}[(\mathsf{r}, \mathsf{q}) := \beta']), \\ \beta' := \mathsf{fb}(\mathsf{r}, \mathsf{q}) \cdot (\mathsf{id}, \mathsf{r_d}, \mathsf{a_d}, \mathsf{mem}(\mathsf{r_s}, \mathsf{a_s})) \end{array}} \quad \text{(popa)}$$

$$\frac{\mathsf{fb}(\mathsf{r}, \mathsf{q}) = (\mathsf{id}, \mathsf{r_d}, \mathsf{a_d}, \mathsf{v}) \cdot \beta}{s \xrightarrow{(\mathsf{r},\mathsf{id},\mathsf{popb},(\mathsf{r_d},\mathsf{a_d}))} (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}[(\mathsf{r_d}, \mathsf{a_d}) := \mathsf{v}], \mathsf{fa}, \mathsf{fb}[(\mathsf{r}, \mathsf{q}) := \beta])} \quad \text{(popb)}$$

$$\frac{\mathsf{instr_r} = \mathsf{pc}(\mathsf{r}) \xrightarrow{\texttt{barrier}} \mathsf{pc}'(\mathsf{r}) \text{ for each } \mathsf{r} \in \mathsf{RANK}}{\begin{array}{c} s \xrightarrow{(1,\mathsf{sn}(1),\mathsf{instr}_1)\cdots(N,\mathsf{sn}(N),\mathsf{instr}_N)} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}, \mathsf{fb}), \\ \mathsf{sn}' := \mathsf{sn}[1 := \mathsf{sn}(1) + 1] \ldots [N := \mathsf{sn}(N) + 1] \end{array}} \quad \text{(bar)}$$

Table 8.1: Transition rules for $X_{\mathsf{pgas}}(\mathcal{P}, N)$, given instruction $\mathsf{instr} := q_1 \xrightarrow{\mathsf{cmd}} q_2$ and current state $s = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$ with $\mathsf{pc}(\mathsf{r}) = q_1$. We define (unless stated otherwise in the rule) $\mathsf{pc}' := \mathsf{pc}[\mathsf{r} := q_2]$ and $\mathsf{sn}' := \mathsf{sn}[\mathsf{r} := \mathsf{sn}(\mathsf{r}) + 1]$. As $\widehat{e}$ we denote the value of expression e in process r and memory configuration mem.

from Figure 8.2 being run on two nodes. It has the following computation:

$$\tau_{\mathsf{1to1}} = \mathsf{write} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \cdot \mathsf{load} \cdot \mathbf{popb} \cdot \mathsf{popb}.$$

Bold events belong to the process with rank 2, the other events to the process with rank 1. We have $\mathsf{addr}(\mathsf{popa}) = (1, x)$, $\mathsf{addr}(\mathsf{popb}) = (2, y)$. Symmetrically, $\mathsf{addr}(\mathbf{popa}) = (2, x)$ and $\mathsf{addr}(\mathbf{popb}) = (1, y)$. The `assert` in Figure 8.2 is a shortcut for a combination of load and assume, and in this computation $\mathsf{addr}(\mathsf{load}) = (1, y)$.

Given an event e, we write $\mathsf{rank}(\mathsf{e})$ for its first component, $\mathsf{id}(\mathsf{e})$ for its second component. As $\mathsf{instr}(\mathsf{e})$ we denote the instruction that created the event. For non-popa and non-popb events, $\mathsf{instr}(\mathsf{e})$ is the third component of the tuple e. For popa and popb events, $\mathsf{instr}(\mathsf{e})$ is equal to those of the leftmost matching event (the event with the same rank and id). For convenience we define *event kinds* $\mathsf{K} := \{\mathsf{load}, \mathsf{store}, \mathsf{assign}, \mathsf{assume}, \mathsf{read}, \mathsf{write}, \mathsf{popa}, \mathsf{popb}, \mathsf{bar}\}$. We use $\mathsf{kind}(\mathsf{e})$ for the third component of the event if it is popa or popb. For the other events we define $\mathsf{kind}(\mathsf{e})$ as expected, depending on $\mathsf{instr}(\mathsf{e})$. If $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{load}, \mathsf{store}, \mathsf{popa}, \mathsf{popb}\}$, we define $\mathsf{addr}(\mathsf{e})$ to be the last component of the event tuple. Otherwise, we define $\mathsf{addr}(\mathsf{e}) := \bot$. We say that e is a *read of* $\mathsf{addr}(\mathsf{e})$ if $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{load}, \mathsf{popa}\}$; we say that it is a *write of* $\mathsf{addr}(\mathsf{e})$ if $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{store}, \mathsf{popb}\}$. Finally, we define $\mathsf{que}(\mathsf{e})$ to be the last component of the event e if $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{read}, \mathsf{write}\}$. If $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{popa}, \mathsf{popb}\}$, then $\mathsf{que}(\mathsf{e})$ is equal to that of the matching read or write event. Otherwise, $\mathsf{que}(\mathsf{e}) := \bot$.

8.1.3 Simulating PGAS APIs

Our formalism natively supports asynchronous data transfers and queues. Operations in the same queue are completed in the order in which they were issued. Using this, we can model the ordering guarantees given by ARMCI and GPI — by putting ordered operations into the same queue.

To model waiting on individual operations (waiting on a handle), we associate a shadow memory address with each operation. Before issuing the operation, the value at this address is set to 0. When the operation has been issued, the process sends to the same queue a read request which overwrites the shadow memory to 1. Now waiting on the individual operation can be implemented by polling on the shadow address associated with the operation. Waiting on all operations in a given queue is done similarly. Synchronous data transfers are modeled by asynchronous transfers, immediately followed by a wait.

8.2 Traces and Robustness

A *trace* of a PGAS program's computation $\sigma \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$ is a graph $T(\sigma) := (V, \rightarrow_{po}, \rightarrow_{cf}, \leftrightarrow_{bar})$ with the set of nodes $V := V \subseteq \bigcup_{\mathsf{r} \in \mathsf{RANK}} \{\mathsf{r}\} \times \mathbb{N} \times \mathcal{I}_{\mathsf{r}}$ and three kinds of (directed and undirected) edges. For notational convenience we will sometimes write e actually meaning $(\mathsf{rank}(\mathsf{e}), \mathsf{id}(\mathsf{e}), \mathsf{instr}(\mathsf{e}))$.

The *program order* relation $\rightarrow_{po}$ gives the ordering in which instructions were executed by the program. Let $\mathsf{e}_{i_1} \dots \mathsf{e}_{i_m}$ be the longest subsequence of non-popa and non-popb events in σ with $\mathsf{tid}(\mathsf{e}_{i_k}) = \mathsf{tid}$ for all $k \in [1..m]$ and $\mathsf{kind}(\mathsf{e}_{i_k}) \notin \{\mathsf{popa}, \mathsf{popb}\}$. Then $\mathsf{e}_{i_1} \rightarrow_{po} \dots \rightarrow_{po} \mathsf{e}_{i_m}$.

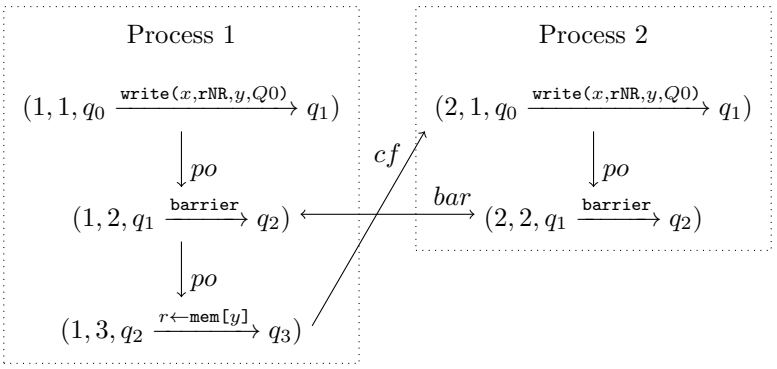

Figure 8.3: Trace of computation of τ_{1to1} from Example 8.2.

The conflict order $\rightarrow_{cf}$ is intuitively a union of $\rightarrow_{src}$, $\rightarrow_{co}$, $\rightarrow_{cf}$ relations as they were defined for Power (Section 4.2) and TSO (Section 2.7.2). Let $\tau = \alpha \cdot \mathsf{e}_1 \cdot \beta \cdot \mathsf{e}_2 \cdot \gamma$, where e_1 and e_2 access the same address, and at least one of them is a write: $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2) = (\mathsf{r}, \mathsf{a})$, $\mathsf{kind}(\mathsf{e}_1) \in \{\mathsf{store}, \mathsf{popb}\}$ or $\mathsf{kind}(\mathsf{e}_2) \in \{\mathsf{store}, \mathsf{popb}\}$. If there is no $\mathsf{e} \in \beta$ such that $\mathsf{addr}(\mathsf{e}) = (\mathsf{r}, \mathsf{a})$ and $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{store}, \mathsf{popb}\}$, then $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$.

The barrier relation $\leftrightarrow_{bar}$ is a symmetric relation that connects matching barrier calls: if $\tau = \alpha \cdot \mathsf{e}_1 \cdots \mathsf{e}_N \cdot \beta$, where $\mathsf{kind}(\mathsf{e}_i) = \mathsf{bar}$ and $\mathsf{rank}(\mathsf{e}_i) = i$, $i \in \mathsf{RANK}$, then $\mathsf{e}_i \leftrightarrow_{bar} \mathsf{e}_j$ for $i \neq j$.

Example 8.3. The trace of computation τ_{1to1} from Example 8.2 is shown in Figure 8.3. We omitted the instructions (local assignments) computing the `rightNeighborRank` (`rNR` in the figure) for simplicity.

We instantiate the robustness problem (Section 2.8) for PGAS.

Problem 8.4 (Robustness against PGAS)**.** Given a program $(\mathcal{P}, N)$, to check whether $T_{\mathsf{sc}}(\mathcal{P}, N) = T_{\mathsf{pgas}}(\mathcal{P}, N)$.

In the above problem definition we assume that the set of SC computations $\mathsf{C}_{\mathsf{sc}}(\mathcal{P}, N)$ consists of all computations from $\mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$, where matching events are located next to each other, i.e., remote reads and writes are executed immediately.

By Lemma 2.16, robustness amounts to the absence of (non-trivial) cycles in the happens-before relation $\rightarrow_{hb} := \rightarrow_{po} \cup \rightarrow_{cf} \cup \leftrightarrow_{bar}$.

Example 8.5. The happens-before relation of computation τ_{1to1} is cyclic (Figure 8.3). Therefore, the program $(\mathcal{P}, N)$ is not robust. Indeed, if remote memory accesses are performed immediately, as they do in sequentially consistent computation, the load done in the `assert()` is guaranteed to read the value of y written by the left neighbor process.

8.3 Normal-Form Computations

We say that computation $\tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P})$ is *in normal form of degree n* if there is a partitioning $\tau = \tau_1 \cdots \tau_n$, such that the following holds:

PGAS-NF-A The parts $\tau_2 \dots \tau_n$ consist solely of popa and popb events.

PGAS-NF-B Let $i \in \{1, 2\}$ let e_i, e'_i be events with $\mathsf{rank}(\mathsf{e}_i) = \mathsf{rank}(\mathsf{e}'_i)$ and $\mathsf{id}(\mathsf{e}_i) = \mathsf{id}(\mathsf{e}'_i)$. If $\mathsf{e}_1, \mathsf{e}_2 \in \tau_j$ and $\mathsf{e}'_1, \mathsf{e}'_2 \in \tau_{j'}$, then $\mathsf{e}_1 <_{\tau_j} \mathsf{e}_2$ iff $\mathsf{e}'_1 <_{\tau_{j'}} \mathsf{e}'_2$.

We elaborate on the second requirement PGAS-NF-B. Consider two accesses a and b to remote processes that can be found in the first part of the computation τ_1. Assume corresponding pop events a' and b' are delayed and can both be found in a later part of the computation, say τ_2. Then the ordering of a' and b' in τ_2 coincides with the order of a and b in τ_1.

Example 8.6. Computation τ_{1to1} from Example 8.2 is not in normal-form of degree n, for any $n \in \mathbb{N}$. Indeed, the definition of normal form (PGAS-NF-A) requires the second and further part to consist solely of popa and popb events. Therefore, the τ_1 part must include at least the following prefix of τ_{1to1}: $\mathsf{write} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \cdot \mathsf{load}$. However, any partitioning of τ_{1to1} with such τ_1 would violate PGAS-NF-B: the write event of the first process comes *before* the **write** event of the second process, but the matching popa comes *after* **popb**.

In the rest of the section we prove the following theorem:

Theorem 8.7. *A PGAS program $(\mathcal{P}, N)$ is robust iff it has no normal-form computation of degree 4 with cyclic happens-before relation.*

Example 8.8. The computation τ''_{1to1} is a normal-form computation of degree 2 (and, consequently, any other higher degree):

$$\tau''_{\mathsf{1to1}} := (\mathsf{write} \cdot \mathsf{popa} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \cdot \mathsf{load}) \cdot (\mathbf{popb} \cdot \mathsf{popb}).$$

Notably, it has the same trace as τ_{1to1} (Figure 8.3) with cyclic happens-before relation.

Consider a computation $\sigma \in \mathsf{C}_{\mathsf{mm}}(\mathcal{P}, N)$. By $\sigma \setminus (\mathsf{r}, \mathsf{id})$ we denote the computation obtained from σ by deleting all events e with $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$ and $\mathsf{id}(\mathsf{e}) = \mathsf{id}$. We shorten $\sigma \setminus (\mathsf{rank}(\mathsf{e}), \mathsf{id}(\mathsf{e}))$ to $\sigma \setminus \mathsf{e}$ in the future.

Lemma 8.9 (Cancellation)**.** *Consider a computation $\varepsilon \neq \tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$ and let e be the last event in τ with $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{popa}, \mathsf{popb}\}$. Then $\tau \setminus \mathsf{e} \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$.*

Proof. All transitions that produced events to the right of e are unconditionally executable. Moreover, τ does not have $\rightarrow_{po}$-successors following e. Therefore, the resulting computation $\tau \setminus \mathsf{e}$ is in $\mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$. □

Assume that a PGAS program $(\mathcal{P}, N)$ is not robust, i.e., by Lemma 2.16, has computations with cyclic happens-before relation. Let τ be the shortest among these computations. Let $\mathsf{e} \in \tau$ be the event determined by Lemma 8.9. If $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{read}, \mathsf{write}\}$, then $\tau = \tau_1 \cdot \mathsf{e} \cdot \tau_2$. Otherwise, $\tau = \tau_1 \cdot \mathsf{e} \cdot \tau_2 \cdot \mathsf{e}' \cdot \tau_3 \cdot \mathsf{e}'' \cdot \tau_4$ with $\mathsf{rank}(\mathsf{e}) = \mathsf{rank}(\mathsf{e}') = \mathsf{rank}(\mathsf{e}'') = \mathsf{r}$ and $\mathsf{id}(\mathsf{e}) = \mathsf{id}(\mathsf{e}') = \mathsf{rank}(\mathsf{e}'') = \mathsf{id}$. Consider the latter case (the former is simpler). Then, $\tau' := \tau \setminus (\mathsf{r}, \mathsf{id}) = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4$. Since $|\tau'| < |\tau|$, the new computation has acyclic happens-before relation. Therefore, by Lemma 2.16, there is a sequentially consistent computation $\sigma \in \mathsf{C}_{\mathsf{sc}}(\mathcal{P})$ (where popa and popb events immediately follow matching read and write events) with the same trace $T(\sigma) = T(\tau')$.

We now use σ to rearrange the events in $\tau \setminus \mathsf{e}$ and transform the original computation τ into a normal-form τ''. The idea is to project σ to the events in τ_1 to τ_4. Reinserting e yields a normal-form computation:

$$\tau'' := (\sigma{\downarrow}\tau_1) \cdot \mathsf{e} \cdot (\sigma{\downarrow}\tau_2) \cdot \mathsf{e}' \cdot (\sigma{\downarrow}\tau_3) \cdot \mathsf{e}'' \cdot (\sigma{\downarrow}\tau_4).$$

The following lemma concludes the proof of Theorem 8.7.

Lemma 8.10 (Reinsertion)**.** *$\tau'' \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$, $T(\tau'') = T(\tau)$, and τ'' is in normal form of degree* 4*.*

Proof. To relieve the reader from the burden of syntax, we consider the case when $\tau' := \tau \setminus \mathsf{e} = \tau_1 \cdot \tau_2$. We start with the program order. Let $\mathsf{e}_1, \mathsf{e}_2 \in \tau_1$ with $\mathsf{e}_1 \rightarrow_{po} \mathsf{e}_2$ in τ and, consequently, in τ'. By definition of σ, we have $\mathsf{e}_1 \rightarrow_{po} \mathsf{e}_2$ in σ. Since $\sigma{\downarrow}\tau_1$ contains e_1 and e_2 and does not add events between them, $\mathsf{e}_1 \rightarrow_{po} \mathsf{e}_2$ holds for $\sigma{\downarrow}\tau_1$ and, consequently, τ''. Assume $\mathsf{e}_1 \in \tau_1$ and $\mathsf{e}_2 \in \tau_2$ with $\mathsf{e}_1 \rightarrow_{po} \mathsf{e}_2$ in τ and in τ'. Then e_1 is the rightmost element in τ_1 with its rank that is different from a pop. Similarly, e_2 is the leftmost element in τ_2 with its rank and different from a pop. The same is valid for their positions in $\sigma{\downarrow}\tau_1$ and $\sigma{\downarrow}\tau_2$, which leads to $\mathsf{e}_1 \rightarrow_{po} \mathsf{e}_2$ in τ''. The case when $\mathsf{e}_1 \in \tau_1$ and $\mathsf{e}_2 = \mathsf{e}$ is similar. Since τ and τ'' consist of the same events, the cardinalities of the respective $\rightarrow_{po}$ relations are equal, and the above inclusion already means the program orders in both computations are equal.

Now we consider the conflict relation. Let $\mathsf{e}_1, \mathsf{e}_2 \in \tau_1$ with $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in τ and hence in τ'. By definition of σ, we have $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in σ. Since $\sigma{\downarrow}\tau_1$ contains e_1 and e_2 and does not add new actions between them, $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ holds for $\sigma{\downarrow}\tau_1$ and, consequently, for τ''.

Assume $\mathsf{e}_1, \mathsf{e}_2 \in \tau_1$ and $\mathsf{e}_1 \not\rightarrow_{cf} \mathsf{e}_2$ in τ. One option is that e_1 and e_2 do not access the same address or both are reads. Then they still will not conflict in τ''. The other option is that $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_3$ in τ, where e_3 is a write to $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2)$ that is located between e_1 and e_2 in τ_1. Then, as already proven, $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_3$ will hold in τ''. Consequently, $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ will not hold in τ''. The case when $\mathsf{e}_1, \mathsf{e}_2 \in \tau_2$ is similar.

Assume $\mathsf{e}_1 \in \tau_1$, $\mathsf{e}_2 \in \tau_2$, and $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in τ. Then, e is not a write to $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2)$, and $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in τ'. Note that $\sigma \downarrow \tau_1$ does not contain a write to $\mathsf{addr}(\mathsf{e}_1)$ to the right of e_1. Otherwise, τ_1 would contain a write e_3 to $\mathsf{addr}(\mathsf{e}_1)$, and $\mathsf{e}_1 \rightarrow^{+}_{cf} \mathsf{e}_3$, which contradicts $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in τ. With a similar argument, $\sigma{\downarrow}\tau_2$ does not contain a write to $\mathsf{addr}(\mathsf{e}_1)$ to the left of e_2. Therefore, $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$ in $\sigma{\downarrow}\tau_1 \cdot \mathsf{e} \cdot \sigma{\downarrow}\tau_2$.

Assume $\mathsf{e}_1 \in \tau_1$, $\mathsf{e}_2 \in \tau_2$, and $\mathsf{e}_1 \not\rightarrow_{cf} \mathsf{e}_2$ in τ. The proof of $\mathsf{e}_1 \not\rightarrow_{cf} \mathsf{e}_2$ in τ'' is as in the case when $\mathsf{e}_1, \mathsf{e}_2 \in \tau_1$.

The case when $\mathsf{e}_1 = \mathsf{e}$ or $\mathsf{e}_2 = \mathsf{e}$ is no harder.

Equality of $\leftrightarrow_{bar}$ in τ and τ'' follows from the fact that consecutive barrier events in τ remain consecutive in σ. By choice of the deleted events in Lemma 8.9, these events belong to part τ_1 and will remain consecutive in $\sigma{\downarrow}\tau_1$.

To prove that $\tau'' \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$, we proceed by contradiction. Let $\alpha \neq \tau''$ be the longest prefix of τ'' so that $s_{\mathsf{pgas}_0} \xrightarrow{\alpha} s$ for some state s. Then $\tau'' = \alpha \cdot \tilde{\mathsf{e}} \cdot \beta$ with $s_{\mathsf{pgas}_0} \xrightarrow{\alpha} s$ and $s \not\xrightarrow{\tilde{\mathsf{e}}}$. Let $s = (\mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$. If $\mathsf{kind}(\tilde{\mathsf{e}}) \in \{\mathsf{popa}, \mathsf{popb}\}$, then $s \not\xrightarrow{\tilde{\mathsf{e}}}$ means that the respective queue fa or fb contains an incorrect topmost element or is empty in s. This is impossible, since τ and τ'' have the same

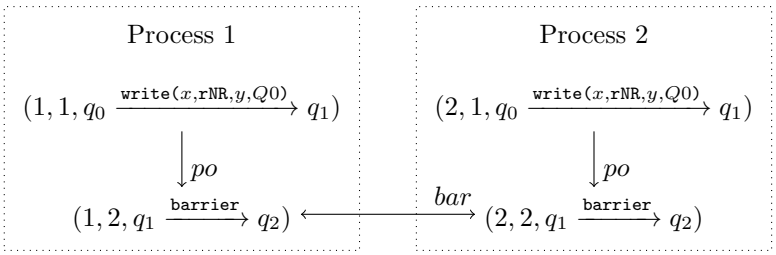

Figure 8.4: Trace of computation of τ'_{1to1} from Example 8.11.

ordering of read and write events with the same rank and queue id (due to the equalities of $\rightarrow_{po}$ and $\rightarrow_{cf}$ in both computations) and the same ordering of popa (popb) events with the same rank and queue id (proven similar to the equality of $\rightarrow_{po}$ in these computations). If $\mathsf{kind}(\tilde{\mathsf{e}}) \notin \{\mathsf{popa}, \mathsf{popb}\}$, then $s \overset{\tilde{\mathsf{e}}}{\not\rightarrow}$ may hold because the transition $q_1 \xrightarrow{\mathsf{cmd}} q_2$ of $\tilde{\mathsf{e}}$ requires a different source state, $q_1 \neq \mathsf{pc}(\mathsf{rank}(\tilde{\mathsf{e}}))$. But since $\mathsf{pc}(\mathsf{rank}(\tilde{\mathsf{e}}))$ is unambiguously determined by the instruction of the $\rightarrow_{po}$-predecessor of $\tilde{\mathsf{e}}$, which is the same in τ'' and in τ due to the matching program-order relations, this is not the case. The last opportunity why $s \overset{\tilde{\mathsf{e}}}{\not\rightarrow}$ may hold is because the transition producing $\tilde{\mathsf{e}}$ reads different values from registers or memory, e.g., $\tilde{\mathsf{e}}$ is a conditional $\texttt{assume}(e)$ and $\widehat{e} = 0$ in s. But since τ'' consists of the same events as τ, has the same program and conflict relations (i.e., reads receive values from the same writes in both computations), and $\tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$, this cannot be the case.

Finally, τ'' is in normal-form of degree 4: $\tau''_1 := \sigma \downarrow \tau_1 \cdot \mathsf{e}$, $\tau''_2 := \sigma \downarrow \tau_2 \cdot \mathsf{e}'$, $\tau''_3 := \sigma \downarrow \tau_3 \cdot \mathsf{e}''$, $\tau''_4 := \sigma \downarrow \tau_4$. PGAS-NF-A holds by the choice of the deleted event, PGAS-NF-B holds by definitions of τ'' and σ. □

Example 8.11. Computation τ_{1to1} from Example 8.2 is a shortest computation with cyclic happens-before relation of program $(\mathsf{1to1}, 2)$. The event deleted according to Lemma 8.9 is $\mathsf{e} = \mathsf{load}$. Therefore, $\tau'_{\mathsf{1to1}} := \tau_1 \cdot \tau_2$, where

$$\tau_1 := \mathsf{write} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \qquad \text{and} \qquad \tau_2 := \mathbf{popb} \cdot \mathsf{popb}.$$

The computation τ'_{1to1} has the trace shown in Figure 8.4. It has acyclic happens-before relation (modulo the trivial cycle due to $\leftrightarrow_{bar}$). The same trace has the sequentially consistent computation σ_{1to1}:

$$\sigma_{\mathsf{1to1}} := \mathsf{write} \cdot \mathsf{popa} \cdot \mathsf{popb} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathbf{popb} \cdot \mathsf{bar} \cdot \mathbf{bar}.$$

The resulting normal-form computation is τ''_{1to1}:

$$\tau''_{\mathsf{1to1}} := (\mathsf{write} \cdot \mathsf{popa} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \cdot \mathsf{load}) \cdot (\mathbf{popb} \cdot \mathsf{popb}).$$

As already noted in Example 8.8, it has the same trace as τ_{1to1} (Figure 8.3).

8.4 From Robustness to Language Emptiness

In this section we reduce the problem of checking the absence of normal-form computations with cyclic happens-before relation to the language emptiness

problem for multiheaded automata. First, we construct, for each program $(\mathcal{P}, N)$, a multiheaded automaton generating all normal-form computations of this program. Next, we intersect the language of this automaton with regular languages checking cyclicity of the happens-before relation. Altogether, the program is robust iff the intersection is empty.

8.4.1 Generating Normal-Form Computations

In this section we show how to describe the language of all normal-form computations of a given program using multiheaded automata. We could not employ pushdown automata (or finite-state automata) for this purpose, because the language consisting of all normal-form computations is generally not context-free (and, consequently, not regular), even if we exclude the (unbounded) serial numbers from the event tuples. Indeed, consider a program $\mathcal{P} := (\{q_0\}, \mathsf{CMD}, \{\mathsf{instr}\}, \{q_0\})$ with $\mathsf{instr} := q_0 \xrightarrow{\texttt{read}(x,1,x,0)} q_0$ running on a single node. The language $\mathcal{L}$ of all normal-form computations of degree 4 of $(\mathcal{P}, 1)$ is not context-free. Indeed, if it would be context-free, its intersection with a regular language would give a context-free language. However, assuming $a := (1, \mathsf{instr}, 0)$, $b := (1, \mathsf{popa}, (1, x))$, $c := (1, \mathsf{popb}, (1, x))$, we have $\mathcal{L} \cap a^*b^*c^* = \{a^n b^n c^n \mid n \in \mathbb{N}\}$, which is well-known to be non-context-free.

We define a 4-headed automaton $M(\mathcal{P}, N) := (S_M, \mathrm{E}, \Delta_M, s_{M0}, S_M)$ that generates all normal-form computations $\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$. In order to generate τ_1, the new automaton tracks the control and memory configurations in the way $X_{\mathsf{pgas}}(\mathcal{P}, N)$ does. For the remainder of the computation, these configurations are not needed. Indeed, τ_2 to τ_4 only consist of popa and popb events that are executable regardless of the control and memory configurations. However, $M(\mathcal{P}, N)$ has to take care of the ordering of popa and popb events from the same queue. In particular, if e_1 handles a request issued before the request of e_2 with $\mathsf{kind}(\mathsf{e}_1) = \mathsf{kind}(\mathsf{e}_2)$, then it cannot be the case that $\mathsf{e}_1 \in \tau_j$ and $\mathsf{e}_2 \in \tau_i$ with $i < j$.

Guided by this discussion, we define a state $s \in S_M$ as a tuple $s := (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{pa}, \mathsf{pb})$. The counter configuration sn, the state and memory configurations pc and mem, and their initial values are defined as in Section 8.1. They reflect the state of the program after it has generated a prefix of τ_1. Let $\mathsf{HEAD} := [1..4]$. The functions $\mathsf{pa}, \mathsf{pb} \colon \mathsf{RANK} \times \mathsf{QUE} \to \mathsf{HEAD}$ give, for each process and each queue, the part τ_1 to τ_4 of the computation where the next popa resp. popb event will be generated. The initial state is $s_{M0} := (\mathsf{pc}_0, \mathsf{mem}_0, \mathsf{pa}_0, \mathsf{pb}_0)$ with $\mathsf{pa}_0(\mathsf{r}, \mathsf{q}) := 1 =: \mathsf{pb}_0(\mathsf{r}, \mathsf{q})$ for all $\mathsf{r} \in \mathsf{RANK}$ and $\mathsf{q} \in \mathsf{QUE}$.

The transition relation Δ_M is the smallest relation defined by the rules in Table 8.2. We elaborate on the rules that are new or substantially different from those in Table 8.1. Rule (gha') lets the automaton choose the part of the computation to which the next popa event will be appended. The first restriction is that the index of the part can only increase, as events from the same queue are processed in order. The second restriction is that popa events cannot be generated to the right of popb events from the same queue. Rule (ghb') is the similar rule for popb events.

By the rules (read') and (write'), the automaton appends a read or write event to τ_1 and the corresponding popa and popb events in one shot to the parts determined by pa and pb. Since a single transition of a multiheaded

$$\frac{\mathsf{pa}(\mathsf{r},\mathsf{q}) < \mathsf{pb}(\mathsf{r},\mathsf{q})}{s \xrightarrow{\varepsilon} s',\ \mathsf{pa}' := \mathsf{pa}[(\mathsf{r},\mathsf{q}) := \mathsf{pa}(\mathsf{r},\mathsf{q}) + 1]} \qquad (\mathsf{gha}')$$

$$\frac{\mathsf{pb}(\mathsf{r},\mathsf{q}) < 4}{s \xrightarrow{\varepsilon} s',\ \mathsf{pb}' := \mathsf{pb}[(\mathsf{r},\mathsf{q}) := \mathsf{pb}(\mathsf{r},\mathsf{q}) + 1]} \qquad (\mathsf{ghb}')$$

$$\frac{\mathsf{cmd} = \mathtt{read}(e_{\mathsf{a}}^{\mathsf{loc}}, e_{\mathsf{r}}^{\mathsf{rem}}, e_{\mathsf{a}}^{\mathsf{rem}}, e_{\mathsf{q}}),\ \mathsf{pa}(\mathsf{r},\widehat{e_{\mathsf{q}}}) = m,\ \mathsf{pb}(\mathsf{r},\widehat{e_{\mathsf{q}}}) = n}{\begin{array}{c} s \xrightarrow{(1,(\mathsf{r},\mathsf{id},\mathsf{instr},\widehat{e_{\mathsf{q}}}))\cdot(m,(\mathsf{r},\mathsf{id},\mathsf{popa},(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}},\widehat{e_{\mathsf{a}}^{\mathsf{rem}}})))\cdot(n,(\mathsf{r},\mathsf{id},\mathsf{popb},(\mathsf{r},\widehat{e_{\mathsf{a}}^{\mathsf{loc}}})))} s', \\ \mathsf{pc}' := \mathsf{pc}[\mathsf{r} := q_2], \\ \text{if } n = 1 \text{ then } \mathsf{mem}' := \mathsf{mem}[(\mathsf{r}, \widehat{e_{\mathsf{a}}^{\mathsf{loc}}}) := \mathsf{mem}(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}}, \widehat{e_{\mathsf{a}}^{\mathsf{rem}}})] \end{array}} \qquad (\mathsf{read}')$$

$$\frac{\mathsf{cmd} = \mathtt{write}(e_{\mathsf{a}}^{\mathsf{loc}}, e_{\mathsf{r}}^{\mathsf{rem}}, e_{\mathsf{a}}^{\mathsf{rem}}, e_{\mathsf{q}}),\ \mathsf{pa}(\mathsf{r},\widehat{e_{\mathsf{q}}}) = m,\ \mathsf{pb}(\mathsf{r},\widehat{e_{\mathsf{q}}}) = n}{\begin{array}{c} s \xrightarrow{(1,(\mathsf{r},\mathsf{id},\mathsf{instr},\widehat{e_{\mathsf{q}}}))\cdot(m,(\mathsf{r},\mathsf{id},\mathsf{popa},(\mathsf{r},\widehat{e_{\mathsf{a}}^{\mathsf{loc}}})))\cdot(n,(\mathsf{r},\mathsf{id},\mathsf{popb},(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}},\widehat{e_{\mathsf{a}}^{\mathsf{rem}}})))} s', \\ \mathsf{pc}' := \mathsf{pc}[\mathsf{r} := q_2], \\ \text{if } n = 1 \text{ then } \mathsf{mem}' := \mathsf{mem}[(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}}, \widehat{e_{\mathsf{a}}^{\mathsf{rem}}}) := \mathsf{mem}(\mathsf{r}, \widehat{e_{\mathsf{a}}^{\mathsf{loc}}})] \end{array}} \qquad (\mathsf{write}')$$

$$\frac{\mathsf{cmd} = r \leftarrow \mathtt{mem}[e]}{s \xrightarrow{1,(\mathsf{r},\mathsf{id},\mathsf{instr},(\mathsf{r},\widehat{e}))} s',\ \mathsf{mem}' := \mathsf{mem}[(\mathsf{r}, r) := \mathsf{mem}(\mathsf{r},\widehat{e})]} \qquad (\mathsf{load}')$$

$$\frac{\mathsf{cmd} = \mathtt{mem}[e_{\mathsf{a}}] \leftarrow e_{\mathsf{v}}}{s \xrightarrow{1,(\mathsf{r},\mathsf{id},\mathsf{instr},(\mathsf{r},\widehat{e_{\mathsf{a}}}))} s',\ \mathsf{mem}' := \mathsf{mem}[(\mathsf{r},\widehat{e_{\mathsf{a}}}) := \widehat{e_{\mathsf{v}}}]} \qquad (\mathsf{store}')$$

$$\frac{\mathsf{cmd} = r \leftarrow e}{s \xrightarrow{1,(\mathsf{r},\mathsf{id},\mathsf{instr})} s',\ \mathsf{mem}' := \mathsf{mem}[(\mathsf{r}, r) := \widehat{e}]} \qquad (\mathsf{assign}')$$

$$\frac{\mathsf{cmd} = \mathtt{assume}(e),\ \widehat{e} \neq 0}{s \xrightarrow{1,(\mathsf{r},\mathsf{id},\mathsf{instr})} s'} \qquad (\mathsf{assume}')$$

$$\frac{\mathsf{instr}_{\mathsf{r}} := \mathsf{pc}(\mathsf{r}) \xrightarrow{\mathtt{barrier}} \mathsf{pc}'(\mathsf{r}) \text{ for each } \mathsf{r} \in \mathsf{RANK}}{\begin{array}{c} s \xrightarrow{(1,(1,\mathsf{sn}(1),\mathsf{instr}_1))\cdots(1,(N,\mathsf{sn}(N),\mathsf{instr}_N))} s' \\ \mathsf{sn}' := \mathsf{sn}[1 := \mathsf{sn}(1) + 1] \ldots [N := \mathsf{sn}(N) + 1] \end{array}} \qquad (\mathsf{bar}')$$

Table 8.2: Transition rules for $M(\mathcal{P}, N)$, given instruction $\mathsf{instr} := q_1 \xrightarrow{\mathsf{cmd}} q_2$ and current state $s = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{pa}, \mathsf{pb})$ with $\mathsf{pc}(\mathsf{r}) = q_1$. The destination state is $s' = (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}', \mathsf{pa}', \mathsf{pb}')$. Unless stated otherwise in the rule, $\mathsf{sn}' := \mathsf{sn}[\mathsf{tid} := \mathsf{sn}(\mathsf{tid}) + 1]$, $\mathsf{pc}' := \mathsf{pc}$, $\mathsf{mem}' := \mathsf{mem}$, $\mathsf{pa}' := \mathsf{pa}$, $\mathsf{pb}' := \mathsf{pb}$, $\mathsf{id} := \mathsf{sn}(\mathsf{r})$.

automaton can generate at most one letter, the rule actually defines multiple consecutive transitions that use fresh intermediary states. If popb is added to τ_1, the memory configuration is updated accordingly. Note that the generation in one shot causes pop events within the same part τ_i to follow in the order of the corresponding read/write events in τ_1. Fortunately, this is always the case in normal-form computations by PGAS-NF-B. Computations that are not in normal form, e.g., τ_{1to1} from Example 8.2, cannot be generated by $M(\mathcal{P}, N)$.

The set of final states of $M(\mathcal{P}, N)$ is S_M.

Lemma 8.12. *$M(\mathcal{P}, N)$ only generates computations of $(\mathcal{P}, N)$: $\mathcal{L}(M(\mathcal{P}, N)) \subseteq \mathcal{C}_{pgas}(\mathcal{P}, N)$.*

Proof. Consider $s_{M0} \xrightarrow{\sigma} s_M$ with $s_M = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{pa}, \mathsf{pb}) \in S_M$. Let $\tau = \mathsf{word}(\sigma) = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4$ with $\tau_i = \mathsf{take2nd}(\sigma{\downarrow}(\{i\} \times \mathrm{E}))$. We prove the following by induction on the length of the computation.

IS1 $s_{\mathsf{pgas}_0} \xrightarrow{\tau} s_{\mathsf{pgas}}$ for some $s_{\mathsf{pgas}} \in F_{\mathsf{pgas}}$. Membership in F_{pgas} means the queues of s_{pgas} are empty.

IS2 $s_{\mathsf{pgas}_0} \xrightarrow{\tau_1} (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$ for some fa, fb, but with the same sn, pc, mem as in s_M above.

IS3 Let $\mathsf{pa}(\mathsf{r}, \mathsf{q}) = k$. Then no τ_i with $i > k$ contains an event e with $\mathsf{kind}(\mathsf{e}) = \mathsf{popa}$, $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$, and $\mathsf{que}(\mathsf{e}) = \mathsf{q}$. A similar statement holds for fb.

IS4 For all $\mathsf{e} \in \tau_2 \cdot \tau_3 \cdot \tau_4$ we have $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{popa}, \mathsf{popb}\}$.

In the base case with $\sigma = \varepsilon$ the inductive invariant trivially holds.

Assume the statement holds for σ. Consider $s_{M0} \xrightarrow{\sigma'} s'_M = (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}', \mathsf{pa}', \mathsf{pb}')$ which extends σ with Rule (read′): $\sigma' = \sigma \cdot (1, \mathsf{e}_1) \cdot (2, \mathsf{e}_2) \cdot (3, \mathsf{e}_3)$, where $\mathsf{kind}(\mathsf{e}_1) = \mathsf{read}$, $\mathsf{kind}(\mathsf{e}_2) = \mathsf{popa}$, $\mathsf{kind}(\mathsf{e}_3) = \mathsf{popb}$. Then $\mathsf{mem}' = \mathsf{mem}$, $\mathsf{pa}' = \mathsf{pa}$, $\mathsf{pb}' = \mathsf{pb}$, and $\tau' = \mathsf{word}(\sigma') = \tau'_1 \cdot \tau'_2 \cdot \tau'_3 \cdot \tau'_4$, where $\tau'_i = \mathsf{take2nd}(\sigma' \downarrow (\{i\} \times \mathrm{E}))$ are $\tau'_1 = \tau_1 \cdot \mathsf{e}_1$, $\tau'_2 = \tau_2 \cdot \mathsf{e}_2$, $\tau'_3 = \tau_3 \cdot \mathsf{e}_3$, and $\tau'_4 = \tau_4$. Since **IS4** and **IS3** hold for σ, they also hold for σ' by definition of σ' and Rule (read′).

It remains to check the behavior of automaton $X_{\mathsf{pgas}}(\mathcal{P}, N)$. By **IS2** from the induction hypothesis and the Rules (read) and (read'), we have $s_{\mathsf{pgas}_0} \xrightarrow{\tau_1 \cdot \mathsf{e}_1} (\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}', \mathsf{fb})$. So **IS2** holds for σ' as well. To check **IS1** for σ', we consider the content of fa'. According to Rule (read), we have $\mathsf{fa}' := \mathsf{fa}[(\mathsf{rank}(\mathsf{e}_1), \mathsf{que}(\mathsf{e}_1)) := \mathsf{fa}(\mathsf{rank}(\mathsf{e}_1), \mathsf{que}(\mathsf{e}_1)) \cdot (\mathsf{id}, \mathsf{r}_{\mathsf{rem}}, \mathsf{a}_{\mathsf{rem}}, \mathsf{r}_{\mathsf{loc}}, \mathsf{a}_{\mathsf{loc}})]$. By the induction hypothesis, we can generate τ_2 from $(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$. The state $(\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}', \mathsf{fb})$, in comparison to the predecessor, appends a single element to fa. Since τ_2 only consists of popa and popb events, we can still generate the computation from $(\mathsf{sn}', \mathsf{pc}', \mathsf{mem}, \mathsf{fa}', \mathsf{fb})$. This yields $s_{\mathsf{pgas}_0} \xrightarrow{\tau_1 \cdot \mathsf{e}_1 \cdot \tau_2} s_1$ for some s_1.

We now show that $s_1 \xrightarrow{\mathsf{e}_2} s_2$ for some s_2. Let $s_1 = (\mathsf{sn}'', \mathsf{pc}'', \mathsf{mem}'', \mathsf{fa}'', \mathsf{fb}'')$. When checking **IS3** for σ', we noted that $\tau_3 \cdot \tau_4$ does not contain popa events $\tilde{\mathsf{e}}$ with rank $\mathsf{rank}(\tilde{\mathsf{e}}) = \mathsf{rank}(\mathsf{e}_1)$ and queue id $\mathsf{que}(\tilde{\mathsf{e}}) = \mathsf{que}(\mathsf{e}_1)$. Therefore, by **IS1** from the induction hypothesis, all elements in $\mathsf{fa}(\mathsf{rank}(\mathsf{e}_1), \mathsf{que}(\mathsf{e}_1))$ are popped by popa transitions in τ_2. As a result, $\mathsf{fa}''(\mathsf{rank}(\mathsf{e}_1), \mathsf{que}(\mathsf{e}_1))$ contains only the single element added by e_1. Comparing Rules (read), (popa), and (read'), shows

$s_1 \xrightarrow{\mathsf{e}_2} s_2$. Note that we need to take the read-rules into account to make sure the contents of the tuple e_2 coincide for $M(\mathcal{P}, N)$ and $X_{\mathsf{pgas}}(\mathcal{P}, N)$.

The fact that $X_{\mathsf{pgas}}(\mathcal{P}, N)$ can accept the rest of computation τ' ($s_2 \xrightarrow{\tau_3 \cdot \mathsf{e}_3 \cdot \tau_4} s_3$ for some s_3) is proven similarly. Emptiness of the queues in s_3 follows from Rule (read') and **IS1** for τ.

The argumentation for write events, $\mathsf{kind}(\mathsf{e}_1) = \mathsf{write}$, is the same. For the remaining kinds of events e_1, the proofs are simpler. There we only need to make use of state and memory configurations, which coincide in $M(\mathcal{P}, N)$ and $X_{\mathsf{pgas}}(\mathcal{P}, N)$. □

Lemma 8.13. *Automaton $M(\mathcal{P}, N)$ generates all normal-form computations of the program: $\{\tau \in \mathsf{C}_{pgas}(\mathcal{P}, N) \mid \tau \text{ is in normal form of degree } 4\} \subseteq \mathcal{L}(M(\mathcal{P}, N))$.*

Proof. Consider a normal-form computation $\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$ with $s_{\mathsf{pgas}_0} \xrightarrow{\tau_1} s_{\mathsf{pgas}}$ for some $s_{\mathsf{pgas}} = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb})$. To prove that $M(\mathcal{P}, N)$ can generate τ, we show the following by induction on the length of the computation. (Note that by PGAS-NF-B we can extend normal-form computations inductively.)

IS1 $s_{M0} \xrightarrow{\sigma} s_M = (\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{pa}, \mathsf{pb})$ with sn, pc and mem from s_{pgas} above.

IS2 We have $\mathsf{take2nd}(\sigma{\downarrow}(\{i\} \times \mathrm{E})) = \tau_i$ for all $i \in \mathsf{HEAD}$.

IS3 Let the last e with $\mathsf{kind}(\mathsf{e}) = \mathsf{popa}$, $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$, $\mathsf{que}(\mathsf{e}) = \mathsf{q}$ be in τ_k. Then $\mathsf{pa}(\mathsf{r}, \mathsf{q}) = k$. If there is no such event, $\mathsf{pa}(\mathsf{r}, \mathsf{q}) = 1$. There is a similar requirement for popb events.

Note that computation ε satisfies all the constraints. Assume the constraints hold for computation τ. We extend τ to a computation $\tau' = \tau'_1 \cdot \tau'_2 \cdot \tau'_3 \cdot \tau'_4$, and show that it also satisfies **IS1** to **IS3**. Extending τ adds an event to the first part of the computation, $s_{\mathsf{pgas}} \xrightarrow{\mathsf{e}_1} s'_{\mathsf{pgas}}$. We do a case distinction based on $\mathsf{kind}(\mathsf{e}_1)$.

Consider the case when $\mathsf{kind}(\mathsf{e}_1) = \mathsf{read}$. Let e_2 and e_3 be the matching popa and popb events and $\tau'_2 = \tau_2 \cdot \mathsf{e}_2$ and $\tau'_3 = \tau_3 \cdot \mathsf{e}_3$. Assume e_1 was generated by the transition $q_1 \xrightarrow{\mathsf{cmd}} q_2$. This means $\mathsf{pc}(\mathsf{rank}(\mathsf{e}_1)) = q_1$. By **IS1** in the induction hypothesis, s_{pgas} and s_M share the same sn, pc and mem. Therefore, by Rules (read) and (read'), $M(\mathcal{P}, N)$ can mimic the read in $X_{\mathsf{pgas}}(\mathcal{P}, N)$. To make sure we append e_2 to τ_2, we have to check the requirements on pa. If $\mathsf{pa}(\mathsf{rank}(\mathsf{e}_2), \mathsf{que}(\mathsf{e}_2)) < 2$, we can use Rule (gha') to adapt the counter. If we assume that $\mathsf{pa}(\mathsf{rank}(\mathsf{e}_2), \mathsf{que}(\mathsf{e}_2)) = k > 2$, we derive a contradiction as follows. By the induction hypothesis, there is an event e' in τ_k with $\mathsf{rank}(\mathsf{e}') = \mathsf{rank}(\mathsf{e}_2)$, $\mathsf{que}(\mathsf{e}') = \mathsf{que}(\mathsf{e}_2)$, and $\mathsf{kind}(\mathsf{e}') = \mathsf{kind}(\mathsf{e}_2) = \mathsf{popa}$. This event has a matching event e in τ_1. Summing up, e, e_1, e_2, e' are contained in τ in this order. Moreover, the latter two events are added to the same queue in reverse order: e' before e_2. A contradiction to the definition of FIFO. The event e_3 is considered similarly. We conclude

$$s_M \xrightarrow{(1,\mathsf{e}_1)\cdot(2,\mathsf{e}_2)\cdot(3,\mathsf{e}_3)} s'_M.$$

The requirements **IS1** to **IS3** are readily checked. The argumentation for write events is the same. For the remaining kinds of events, the induction step is simpler since sn, pc, and mem coincide in s_{pgas} and s_M. □

Lemma 8.14. $\{\tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N) \mid \tau \text{ is in normal form of degree } 4\} = \mathcal{L}(M(\mathcal{P}, N))$.

Proof. The inclusion from left to right is Lemma 8.13. The inclusion from right to left holds by Lemma 8.12 and the observation that $M(\mathcal{P}, N)$ only generates normal-form computations. □

The following lemma states that $M(\mathcal{P}, N)$ generates events in program order.

Lemma 8.15. *Consider computation* $s_{M0} \xrightarrow{\sigma} s_M$ *with events* $(1, \mathsf{e}_1) <_\sigma (1, \mathsf{e}_2)$ *so that* $\mathsf{kind}(\mathsf{e}_1), \mathsf{kind}(\mathsf{e}_2) \notin \{\mathsf{popa}, \mathsf{popb}\}$ *and* $\mathsf{rank}(\mathsf{e}_1) = \mathsf{rank}(\mathsf{e}_2)$. *Then* $\mathsf{e}_1 \rightarrow_{po}^{+} \mathsf{e}_2$ *in* $\tau = \mathsf{word}(\sigma)$.

Proof. By definition of the transition relation Δ_M and $\rightarrow_{po}$. □

The following lemma states that $M(\mathcal{P}, N)$ generates the events popa and popb immediately after the corresponding read or write event.

Lemma 8.16. *Let* $s_{M0} \xrightarrow{\sigma} s_M$, $\tau = \mathsf{word}(\sigma)$, *and* $\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3 \in \tau$ *with* $\mathsf{kind}(\mathsf{e}_1) \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{kind}(\mathsf{e}_2) = \mathsf{popa}$, *and* $\mathsf{kind}(\mathsf{e}_3) = \mathsf{popb}$. *Then* e_1, e_2, *and* e_3 *are matching events in* τ *iff* $\sigma = \sigma_1 \cdot (1, \mathsf{e}_1) \cdot (m, \mathsf{e}_2) \cdot (n, \mathsf{e}_3) \cdot \sigma_2$ *for some* σ_1, σ_2 *and* $m, n \in \mathsf{HEAD}$ *with* $m \leq n$.

Proof. By Rules (read) and (write), the preconditions on (gha) and (ghb), and the definition of matching events. □

8.4.2 Checking Cyclicity of the Happens-Before Relation

In this subsection we show that if a computation has a happens-before cycle, it has a happens-before cycle, in which each process contributes only once. We call such a cycle *beautiful*. Next, we show how to detect beautiful cycles using finite automata.

We call a happens-before cycle *beautiful*, if it has the following form:

$$(\mathsf{r}_1, i_1, \mathsf{instr}_1) \rightarrow_{po}^{*} (\mathsf{r}_1, i_1', \mathsf{instr}_1') \rightarrow_{hop} \ldots \rightarrow_{hop} (\mathsf{r}_n, i_n, \mathsf{instr}_n) \rightarrow_{po}^{*} (\mathsf{r}_n, i_n', \mathsf{instr}_n') \rightarrow_{hop} (\mathsf{r}_1, i_1, \mathsf{instr}_1).$$

Here, $\rightarrow_{hop} := (\rightarrow_{cf} \cup \leftrightarrow_{bar})$ and $\mathsf{r}_k \neq \mathsf{r}_l$ for $k \neq l$. We call $\theta := \mathsf{r}_1 \ldots \mathsf{r}_n$ the *profile* of the cycle.

Example 8.17. Computations τ_{1to1} (Example 8.2) and τ_{1to1}'' (Example 8.8) have a single happens-before cycle (Figure 8.3). This cycle is beautiful.

Lemma 8.18. *Computation* $\tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P})$ *has a happens-before cycle iff it has a beautiful happens-before cycle.*

Proof. Similar to the proof of Lemma 3.7. □

In spite of the additional restrictions, beautiful cycles are not trivial to recognize. The reason is that the events yielding the trace nodes belonging to the cycle are not necessarily contained in the computation in the order in which they appear in the cycle. The idea of our cycle detection is to guess a cycle profile θ, then guess two events in program order in each process belonging to the

profile, and finally check that these events are $\rightarrow_{hop}$-related, as needed according to θ. We accomplish the former by an augmented multiheaded automaton $M'(\mathcal{P}, N, \theta)$. The latter check is performed by a finite automaton.

The automaton $M'(\mathcal{P}, N, \theta)$ generates computations over the alphabet $\mathrm{E} \times \mathsf{M}$ with $\mathsf{M} := 2^{\{\mathsf{enter},\mathsf{leave}\}}$. The automaton non-deterministically guesses and marks the first and the last event in each process that contribute to a cycle. The events marked by enter yield the nodes $(\mathsf{tid}_j, i_j, \mathsf{instr}_j)$ of the beautiful cycle. The events marked by leave yield the nodes $(\mathsf{tid}_j, i'_j, \mathsf{instr}'_j)$ of the beautiful cycle. Note that automaton $M(\mathcal{P}, N)$ executes instructions of each process in program order (Lemma 8.15) and generates matching events in one shot (Lemma 8.16). Therefore, if $M'(\mathcal{P}, N)$ produces an event e_1 marked with enter and later (or while handling the same instruction) produces an event e_2 of the same process marked with leave, then it is guaranteed that $\mathsf{e}_1 \rightarrow_{po}^* \mathsf{e}_2$.

We set $M'(\mathcal{P}, N, \theta) := (S'_M, \mathrm{E} \times \mathsf{M}, \Delta_{M'}, s_{M'0}, F_{M'})$, where events are optionally marked by enter and/or leave from $\mathsf{M} := 2^{\{\mathsf{enter},\mathsf{leave}\}}$. The set of states S'_M consists of the states S_M extended by information about which marked events have been issued for each process: $S'_M := S_M \times \{\bot, \mathsf{enter}, \mathsf{leave}\}^{\mathsf{RANK}}$. The initial state is $s_{M'0} := (s_{M0}, \mu_0)$ with $\mu_0 := \lambda \mathsf{r}.\bot$. The transition relation $\Delta_{M'}$ is defined as follows:

M1 $(s, \mu) \xrightarrow{\varepsilon} (s', \mu)$ if $s \xrightarrow{\varepsilon} s'$.

M2 $(s, \mu) \xrightarrow{i,(\mathsf{e},\emptyset)} (s', \mu)$ if $s \xrightarrow{i,\mathsf{e}} s'$.

M3 $(s, \mu) \xrightarrow{i,(\mathsf{e},\{\mathsf{enter}\})} (s', \mu[\mathsf{rank}(\mathsf{e}) := \mathsf{enter}])$ if $s \xrightarrow{i,\mathsf{e}} s'$, $\mathsf{addr}(\mathsf{e}) \neq \bot$ or $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, $\mu(\mathsf{rank}(\mathsf{e})) = \bot$, and $\mathsf{rank}(\mathsf{e}) \in \theta$.

M4 $(s, \mu) \xrightarrow{i,(\mathsf{e},\{\mathsf{enter},\mathsf{leave}\})} (s', \mu[\mathsf{rank}(\mathsf{e}) := \mathsf{leave}])$ if $s \xrightarrow{i,\mathsf{e}} s'$, $\mathsf{addr}(\mathsf{e}) \neq \bot$ or $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, $\mu(\mathsf{rank}(\mathsf{e})) = \bot$, and $\mathsf{rank}(\mathsf{e}) \in \theta$.

M5 $(s, \mu) \xrightarrow{i,(\mathsf{e},\{\mathsf{leave}\})} (s, \mu[\mathsf{rank}(\mathsf{e}) := \mathsf{leave}])$ if $s \xrightarrow{i,\mathsf{e}} s'$, $\mathsf{addr}(\mathsf{e}) \neq \bot$ or $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, $\mu(\mathsf{rank}(\mathsf{e})) = \mathsf{enter}$, and $\mathsf{rank}(\mathsf{e}) \in \theta$.

M6 $(s, \mu) \xrightarrow{(i,(\mathsf{e}_1,\{\mathsf{leave}\}))\cdot(j,(\mathsf{e}_2,\{\mathsf{enter}\}))} (s', \mu[\mathsf{rank}(\mathsf{e}_1) := \mathsf{leave}])$ if $s \xrightarrow{(i,\mathsf{e}_1)\cdot(j,\mathsf{e}_2)} s'$, $\mathsf{kind}(\mathsf{e}_1) = \mathsf{popa}$, $\mathsf{kind}(\mathsf{e}_2) = \mathsf{popb}$, $\mu(\mathsf{rank}(\mathsf{e}_1)) = \bot$, and $\mathsf{rank}(\mathsf{e}) \in \theta$.

The set of final states is $F_{M'} := \{(s, \mu) \mid s \in F_M \text{ and } \mu(\mathsf{r}) = \mathsf{leave} \text{ for all } \mathsf{r} \in \theta\}$.

Lemma 8.19. *The languages of $M(\mathcal{P}, N)$ and $M'(\mathcal{P}, N, \theta)$ match up to the markings: $\mathcal{L}(M(\mathcal{P}, N)) = \mathit{take1st}(\mathcal{L}(M'(\mathcal{P}, N, \theta)))$.*

Proof. The inclusion $\mathcal{L}(M(\mathcal{P}, N)) \subseteq \mathsf{take1st}(\mathcal{L}(M'(\mathcal{P}, N)))$ holds due to the Rules **M1** and **M2** in the definition of $\Delta_{M'}$. The reverse inclusion $\mathcal{L}(M(\mathcal{P}, N)) \supseteq \mathsf{take1st}(\mathcal{L}(M'(\mathcal{P}, N)))$ follows from the fact that $(s, \mu) \xrightarrow{i,(\mathsf{e},m)} (s', \mu')$ requires $s \xrightarrow{i,\mathsf{e}} s'$ (**M2**-**M6**). □

Lemma 8.20. *Consider a marked computation $\tau \in \mathcal{L}(M'(\mathcal{P}, N, \theta))$ and events (e_1, m_1) and (e_2, m_2) in τ with $\mathsf{rank}(\mathsf{e}_1) = \mathsf{rank}(\mathsf{e}_2)$ and $\mathsf{enter} \in m_1$, $\mathsf{leave} \in m_2$. Then $\mathsf{e}_1 \rightarrow_{po}^* \mathsf{e}_2$.*

Proof. Consider $s_{M'0} \xrightarrow{\sigma} s_{M'}$ and let $\tau = \mathsf{word}(\sigma)$. Let (e_1, m_1) and (e_2, m_2) be two events in τ with $\mathsf{rank}(\mathsf{e}_1) = \mathsf{rank}(\mathsf{e}_2)$ and $\mathsf{enter} \in m_1$, $\mathsf{leave} \in m_2$. Then, σ contains (i, e_1, m_1) and (j, e_2, m_2) for some $i, j \in \mathsf{HEAD}$.

- If $(i, \mathsf{e}_1, m_1) >_\sigma (j, \mathsf{e}_2, m_2)$, then e_1 and e_2 were generated by the two transitions defined by Rule **M6**. By construction of $M(\mathcal{P}, N)$, $\mathsf{rank}(\mathsf{e}_1) = \mathsf{rank}(\mathsf{e}_2)$ and $\mathsf{id}(\mathsf{e}_1) = \mathsf{id}(\mathsf{e}_2)$, and $\mathsf{e}_1 \rightarrow_{po}^* \mathsf{e}_2$ trivially holds.
- If $(i, \mathsf{e}_1, m_1) = (j, \mathsf{e}_2, m_2)$, then $m_1 = m_2 = \{\mathsf{enter}, \mathsf{leave}\}$ and $\mathsf{e}_1 = \mathsf{e}_2$ is the event generated by **M4**. Clearly, $\mathsf{e}_1 \rightarrow_{po}^* \mathsf{e}_2$.
- If $(i, \mathsf{e}_1, m_1) <_\sigma (j, \mathsf{e}_2, m_2)$, then e_1 was generated by **M3**, and e_2 was generated by Rule **M5**. By Lemma 8.16 and Lemma 8.15, $\mathsf{e}_1 \rightarrow_{po}^* \mathsf{e}_2$.

□

For the next lemma, consider a normal-form computation $\tau \in \mathsf{C}_{\mathsf{pgas}}(\mathcal{P}, N)$ and a cycle profile $\theta = \mathsf{r}_1 \dots \mathsf{r}_n$. Moreover, assume that for each rank r_i there are $\mathsf{e}_i, \mathsf{e}_i' \in \tau$ with $\mathsf{rank}(\mathsf{e}_i) = \mathsf{rank}(\mathsf{e}_i') = \mathsf{r}_i$ that satisfy $\mathsf{e}_i \rightarrow_{po} \mathsf{e}_i'$ and

$$(\mathsf{addr}(\mathsf{e}_i) \neq \bot \text{ or } \mathsf{kind}(\mathsf{e}_i) = \mathsf{bar}) \text{ and } (\mathsf{addr}(\mathsf{e}_i') \neq \bot \text{ or } \mathsf{kind}(\mathsf{e}_i') = \mathsf{bar}).$$

Lemma 8.21. *Under the above assumptions, there is a marked computation $\tau' \in \mathcal{L}(M'(\mathcal{P}, N, \theta))$ with $\mathsf{take1st}(\tau') = \tau$ that contains, for each $i \in [1..n]$, a marked event (e_i, m_i) with $\mathsf{enter} \in m_i$ and (e_i', m_i') with $\mathsf{leave} \in m_i'$. All other marked events $(\mathsf{e}, m) \in \tau$ have $m = \emptyset$.*

Proof. We prove the statement of the lemma by induction on the size n of the cycle profile. The base case $n = 0$ is due to Lemma 8.13 and the Rules **M1** and **M2**: $M'(\mathcal{P}, N)$ can generate a marked computation τ_0 with $\mathsf{take1st}(\tau_0) = \tau$ and all markings being $\emptyset$. Formally, there is $s_{M'0} \xrightarrow{\sigma_0} s_{M'}$ for some $s_{M'} \in F_{M'}$ with $\tau_0 = \mathsf{word}(\sigma_0)$.

In the induction step, assume the claim holds for sets of ranks of size $n-1$ and consider $\{\mathsf{r}_1, \dots, \mathsf{r}_n\} \subseteq \mathsf{RANK}$. By the hypothesis, there is $s_{M'0} \xrightarrow{\sigma_{n-1}} s_{M'}$ for some $s_{M'} \in F_{M'}$. Moreover, for each $i \in [1..n-1]$ it holds that $\tau_{n-1} = \mathsf{word}(\sigma_{n-1})$ contains a marked event (e_i, m_i) with $\mathsf{enter} \in m_i$ and a marked event (e_i', m_i') with $\mathsf{leave} \in m_i'$. All other events in τ_{n-1} have empty markings. To prove the statement for n, consider the possible mutual dispositions of $(i, \mathsf{e}_n, \emptyset)$ and $(j, \mathsf{e}_n', \emptyset)$ in σ_{n-1}.

- If $(i, \mathsf{e}_n, \emptyset)$ and $(j, \mathsf{e}_n', \emptyset)$ are the same event, we have $\sigma_{n-1} = \sigma' \cdot (i, \mathsf{e}_n, \emptyset) \cdot \sigma''$ and $(i, \mathsf{e}_n, \emptyset)$ was generated by Rule **M2**. This transition can be replaced by **M4**, which yields $\sigma_n = \sigma' \cdot (i, \mathsf{e}_n, \{\mathsf{enter}, \mathsf{leave}\}) \cdot \sigma''$.
- If $(i, \mathsf{e}_n, \emptyset) <_{\sigma_{n-1}} (j, \mathsf{e}_n', \emptyset)$, then $\sigma_{n-1} = \sigma' \cdot (i, \mathsf{e}_n, \emptyset) \cdot \sigma'' \cdot (j, \mathsf{e}_n', \emptyset) \cdot \sigma'''$, where $(i, \mathsf{e}_n, \emptyset)$ and $(j, \mathsf{e}_n', \emptyset)$ were generated by **M2**. These transitions can be replaced by **M3** and **M5** transitions, resulting in $\sigma_n = \sigma' \cdot (i, \mathsf{e}_n, \{\mathsf{enter}\}) \cdot \sigma'' \cdot (j, \mathsf{e}_n', \{\mathsf{leave}\}) \cdot \sigma'''$.
- Consider $(i, \mathsf{e}_n, \emptyset) >_{\sigma_{n-1}} (j, \mathsf{e}_n', \emptyset)$. By Lemma 8.15 and Lemma 8.16, we know that e_n' and e_n are matching events. We derive $\mathsf{addr}(\mathsf{e}_n') \neq \bot \neq \mathsf{addr}(\mathsf{e}_n)$. This gives $\mathsf{kind}(\mathsf{e}_n) = \mathsf{popb}$ and $\mathsf{kind}(\mathsf{e}_n') = \mathsf{popa}$. With Lemma 8.16, $\sigma_{n-1} = \sigma' \cdot (j, \mathsf{e}_n', \emptyset) \cdot (i, \mathsf{e}_n, \emptyset) \cdot \sigma''$. The events were generated by two consecutive **M2** transitions. These transitions can be replaced by **M6**, which yields $\sigma_n = \sigma' \cdot (j, \mathsf{e}_n', \{\mathsf{leave}\}) \cdot (i, \mathsf{e}_n, \{\mathsf{enter}\}) \cdot \sigma''$.

Since σ_n is obtained from σ_{n-1} by replacing one or two marked events of rank r_n, and generation of the other events does not rely on $\mu(\mathsf{r}_n)$ (all other events of rank r_n are not marked), we have $s_{M'0} \xrightarrow{\sigma_n} s_{M'}$ for some $s_{M'} \in F_{M'}$. □

Example 8.22. The normal-form computation τ''_{1to1} from Example 8.8 has the cycle shown in Figure 8.3. This cycle has profile $\theta := 1, 2$ (or its cyclic rotation $\theta' := 2, 1$). The marked computation that marks the first and the last event in each process that belong to the cycle is

$$(\mathsf{write}, \emptyset) \cdot (\mathsf{popa}, \emptyset) \cdot (\mathbf{write}, \emptyset) \cdot (\mathbf{popa}, \emptyset) \cdot (\mathsf{bar}, \{\mathsf{enter}\}) \cdot (\mathbf{bar}, \{\mathsf{leave}\}) \cdot (\mathsf{load}, \{\mathsf{leave}\}) \cdot (\mathsf{popb}, \emptyset) \cdot (\mathbf{popb}, \{\mathsf{enter}\}).$$

Lemma 8.20 and Lemma 8.21 essentially say that the augmented multi-headed automaton $M'(\mathcal{P}, N, \theta)$, for any beautiful cycle with profile θ, can guess the first and the last event that belong to the cycle in each process. What is left is to check that the last event of process $\theta[k]$ belonging to the cycle happens before the first event of process $\theta[k+1]$ belonging to the cycle. We check this with a finite automaton $H^{\mathsf{r}_k, \mathsf{r}_{k+1}}$.

The finite automaton $H^{\mathsf{r}_i, \mathsf{r}_{i+1}}$ accepts a marked computation if there is a conflict or barrier edge from the leave-marked event of process r_i to the enter-marked event of process r_{i+1}. Consider the case of conflicts. The automaton looks for a marked event (e_i, m_i) with $\mathsf{rank}(\mathsf{e}_i) = \mathsf{r}_i$ marked by $\mathsf{leave} \in m_i$. It remembers the kind and the address of this event. Then, it seeks a marked event $(\mathsf{e}_{i+1}, m_{i+1})$ with $\mathsf{rank}(\mathsf{e}_{i+1}) = \mathsf{r}_{i+1}$ marked by $\mathsf{enter} \in m_{i+1}$. If both events are found, they touch the same address, and one of them is a write, the automaton reaches the accepting state. Formally, we define $H^{\mathsf{r}_1, \mathsf{r}_2} := (S_H, \mathrm{E} \times \mathsf{M}, \Delta_H, s_{H0}, F_H)$. The set of states $S_H := \{\mathsf{init}, \mathsf{accept}\} \cup (\mathsf{K} \times \mathsf{RANK} \times \mathsf{ADDR})$. The initial state is $s_{H0} := \mathsf{init}$. The set of final states is $F_H := \{\mathsf{accept}\}$. The transition relation Δ_H is defined as follows:

HB1 $\mathsf{init} \xrightarrow{\mathsf{e}, m} \mathsf{init}$ with $\mathsf{rank}(\mathsf{e}) \neq \mathsf{r}_1$ or $\mathsf{enter} \notin m$.

HB2 $\mathsf{init} \xrightarrow{\mathsf{e}, m} (\mathsf{kind}(\mathsf{e}), \mathsf{addr}(\mathsf{e}))$ for $\mathsf{kind}(\mathsf{e}) \neq \mathsf{bar}$ if $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$ and $\mathsf{leave} \in m$.

HB3 $(\mathsf{k}, \mathsf{r}, \mathsf{a}) \xrightarrow{\mathsf{e}, m} (\mathsf{k}, \mathsf{r}, \mathsf{a})$ for $\mathsf{k} \neq \mathsf{bar}$ if $\mathsf{addr}(\mathsf{e}) \neq (\mathsf{r}, \mathsf{a})$ or $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{store}, \mathsf{popb}\}$.

HB4 $(\mathsf{k}, \mathsf{r}, \mathsf{a}) \xrightarrow{\mathsf{e}, m} (\mathsf{accept})$ for $\mathsf{k} \neq \mathsf{bar}$ if $\mathsf{addr}(\mathsf{e}) = (\mathsf{r}, \mathsf{a})$, $\mathsf{rank}(\mathsf{e}) = \mathsf{r}_2$, $\mathsf{enter} \in m$, and $\{\mathsf{k}, \mathsf{kind}(\mathsf{e})\} \cap \{\mathsf{store}, \mathsf{popb}\} \neq \emptyset$.

HB5 $\mathsf{accept} \xrightarrow{\mathsf{e}, m} \mathsf{accept}$ for all $(\mathsf{e}, m) \in \mathrm{E} \times \mathsf{M}$.

HB6 $\mathsf{init} \xrightarrow{\mathsf{e}, m} \mathsf{barrier}$ if $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, ($\mathsf{rank}(\mathsf{e}) = \mathsf{r}_1$ and $\mathsf{leave} \in m$) or ($\mathsf{rank}(\mathsf{e}) = \mathsf{r}_2$ and $\mathsf{enter} \in m$).

HB7 $\mathsf{barrier} \xrightarrow{\mathsf{e}, m} \mathsf{barrier}$ if $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, $\mathsf{rank}(\mathsf{e}) \notin \{\mathsf{r}_1, \mathsf{r}_2\}$.

HB8 $\mathsf{barrier} \xrightarrow{\mathsf{e}, m} \mathsf{accept}$ if $\mathsf{kind}(\mathsf{e}) = \mathsf{bar}$, ($\mathsf{rank}(\mathsf{e}) = \mathsf{r}_1$ and $\mathsf{leave} \in m$) or ($\mathsf{rank}(\mathsf{e}) = \mathsf{r}_2$ and $\mathsf{enter} \in m$).

Lemma 8.23. *Consider $\mathsf{r}_1, \mathsf{r}_2 \in \mathsf{RANK}$ and $\tau \in \mathcal{L}(M'(\mathcal{P}, N))$ that has a single marked event (e_i, m_i) with $\mathsf{leave} \in m_i$ and $\mathsf{rank}(\mathsf{e}_i) = \mathsf{r}_1$ and a single (e_j, m_j) with $\mathsf{enter} \in m_j$ and $\mathsf{rank}(\mathsf{e}_j) = \mathsf{r}_2$. Then $\tau \in \mathcal{L}(H^{\mathsf{r}_1, \mathsf{r}_2})$ iff $\mathsf{e}_i \rightarrow_{hop} \mathsf{e}_j$.*

Proof. We give the proof for memory accesses, the argumentation in the case of barriers is similar. We start with the implication from left to right. In order to reach the accepting state accept the first time, the automaton must have reached a state $(\mathsf{k}, \mathsf{r}, \mathsf{a})$ and performed a transition defined by **HB4**. This transition had to consume the symbol (e_j, m_j) which is, according to the statement of the lemma, the only marked event in τ with $\mathsf{rank}(\mathsf{e}_j) = \mathsf{r}_2$ and $\mathsf{enter} \in m_j$. The state $(\mathsf{k}, \mathsf{r}, \mathsf{a})$ was reached the first time via a transition defined by **HB2**. This transition had to consume the symbol (e_i, m_i) which is, according to the statement of the lemma, the only marked event in τ with $\mathsf{rank}(\mathsf{e}_i) = \mathsf{r}_1$ and $\mathsf{leave} \in m_i$. According to **HB2**, $\mathsf{k} = \mathsf{kind}(\mathsf{e})$ and $(\mathsf{r}, \mathsf{a}) = \mathsf{addr}(\mathsf{e})$. Therefore, **HB4** requires that e_i and e_j access the same address and at least one of them is a write. Moreover, according to Rule **HB3**, the automaton could not consume a marked event which is a write to (r, a) after reading (e_i, m_i) and before reading (e_j, m_j). Altogether, by definition of the conflict relation, $\mathsf{e}_i \rightarrow_{cf} \mathsf{e}_j$.

For the proof from right to left, let $\tau = \tau_1 \cdot (\mathsf{e}_i, m_i) \cdot \tau_2 \cdot (\mathsf{e}_j, m_j) \cdot \tau_3$. The first part, τ_1, is read by the transitions defined by **HB1**. Indeed, (e_i, m_i) is the only marked event in τ that does not satisfy the requirements of this rule. Then the automaton performs the transition defined by **HB2**, reads (e_i, m_i), and reaches the state $(\mathsf{k}, \mathsf{r}, \mathsf{a})$ with $\mathsf{k} = \mathsf{kind}(\mathsf{e})$ and $(\mathsf{r}, \mathsf{a}) = \mathsf{addr}(\mathsf{e})$. Since $\mathsf{e}_i \rightarrow_{cf} \mathsf{e}_j$, part τ_2 does not contain writes to $\mathsf{addr}(\mathsf{e}_i)$. It is consumed by the transitions defined by **HB3**. Finally, the automaton performs the transition defined by **HB4** and reaches the accepting state. There it loops on the symbols from τ_3. □

Since finite automata are closed under intersection, we can define the *finite automaton for cycle profile* $\theta = \mathsf{r}_1 \dots \mathsf{r}_k$ as

$$H^\theta := H^{\mathsf{r}_1,\mathsf{r}_2} \cap \ldots \cap H^{\mathsf{r}_{k-1},\mathsf{r}_k} \cap H^{\mathsf{r}_k,\mathsf{r}_1}.$$

Lemma 8.24. *Consider a cycle type θ and let $\tau \in \mathsf{take1st}\big(\big(\mathcal{L}(M'(\mathcal{P}, N, \theta)) \cap \mathcal{L}(H^\theta)\big)\big)$. Then τ is a computation of $(\mathcal{P}, N)$ and has a beautiful cycle with profile θ.*

Proof. By Lemma 8.12 and Lemma 8.19, τ is a computation of program $(\mathcal{P}, N)$. By Lemma 8.20 and Lemma 8.23, τ has a beautiful cycle with profile θ. □

Lemma 8.25. *Consider a cycle profile θ and let τ be a normal-form computation of $(\mathcal{P}, N)$ that has a cycle with this profile. Then $\tau \in \mathsf{take1st}\big(\big(\mathcal{L}(M'(\mathcal{P}, N, \theta)) \cap \mathcal{L}(H^\theta)\big)\big)$.*

Proof. By Lemma 8.21, $M'(\mathcal{P}, N)$ can generate τ' with $\mathsf{take1st}(\tau') = \tau$, the events $\mathsf{e}_i, \mathsf{e}'_i$ from the definition of a beautiful cycle marked with enter and leave respectively, and the other events marked with $\emptyset$. By Lemma 8.23, the automata $H^{\mathsf{r}_i,\mathsf{r}_{i+1}}$ will accept τ', due to $\mathsf{e}'_i \rightarrow_{hop} \mathsf{e}_{i+1}$. □

Theorem 8.26. *A program $(\mathcal{P}, N)$ is robust iff $\mathcal{L}(M'(\mathcal{P}, N)) \cap \mathcal{L}(H^\theta) = \emptyset$ for all cycle types θ.*

Proof. The statement follows from Theorem 8.7, Lemma 8.24, Lemma 8.18, and Lemma 8.25. □

We can now prove our main result in this chapter.

Theorem 8.27. *Robustness against PGAS is* PSPACE*-complete.*

Proof. The PSPACE lower bound follows from PSPACE-hardness of SC state reachability (Lemma 2.7). To reduce reachability to robustness, we introduce an artificial happens-before cycle in the target state, by e.g., executing the OneToOne program when the state is reached.

We can enumerate all cycle profiles in space polynomial in the size of $(\mathcal{P}, N)$. For each profile, we can check emptiness of the intersection $\mathcal{L}(M'(\mathcal{P}, N)) \cap \mathcal{L}(H^\theta)$. By Theorem 8.26, the program is robust if all intersections are empty. By Lemma 3.3, the size of the intersection is polynomial in the size of the automata, i.e., exponential in the size of the program $(\mathcal{P}, N)$. By Lemma 3.4, its emptiness can be decided in NL, i.e., in space polynomial in the size of the program. Altogether, robustness is decidable in PSPACE. □

8.5 Parameterized Reachability and Robustness

Defining a parameterized version of the robustness and reachability problems for PGAS, along with Sections 2.5 and 2.7, comes across the following difficulty. When we had a finite number of processes, each process or node could have been identified by a rank from a finite set of ranks RANK. For identifying an infinite number of nodes we need an infinite set of ranks and, consequently, an infinite data domain DOM, since $\mathsf{RANK} \subseteq \mathsf{DOM}$. However, then the computational model becomes immediately Turing-complete, and state reachability and robustness become undecidable.

Alternatively, we can specify that a process can only communicate with a finite number of its neighbors. In particular, we can assume that processes are totally ordered, and each process can communicate only with the previous (left neighbor) and the next (right neighbor) process in this order (if these processes exist). This is an example of a communication pattern typical for stencil code — a class of algorithms that iteratively recompute values of an array based on the values in the arrays on neighbor nodes. Because of the bounded DOM we can no longer provide expressions that would return the rank of the current process and the total number of processes. To keep the model useful, we assume there is a special expression that returns a non-zero value for a single dedicated process. This dedicated process may then distribute unbounded amount of work among the other unboundedly many processes.

Having said the above, we define the parameterized reachability and robustness problem for PGAS.

Problem 8.28 (Parameterized state reachability under PGAS)**.** Given a program $\mathcal{P}$ and a control state $q \in Q$, to check whether $(\mathcal{P}, N)$ reaches a state $(\mathsf{sn}, \mathsf{pc}, \mathsf{mem}, \mathsf{fa}, \mathsf{fb}) \in F_{\mathsf{pgas}}$ with $\mathsf{pc}(\mathsf{tid}) = q$ for any $N \in \mathbb{N}$.

Problem 8.29 (Parameterized robustness against PGAS)**.** Given a program $\mathcal{P}$, to check whether $T_{\mathsf{sc}}(\mathcal{P}(I, N)) = T_{\mathsf{tso}}(\mathcal{P}(I, N))$ for all $N \in \mathbb{N}$.

Despite the very restricted allowed communication pattern, the parameterized reachability and robustness are undecidable.

Theorem 8.30. *Parameterized state reachability under PGAS is undecidable.*

Proof. Consider a Turing machine operating on a tape with alphabet DOM, $|\mathsf{DOM}| \geq 2$. Execution of the machine using at most N cells of a tape can be

simulated by a program $(\mathcal{P}, N)$, where $\mathcal{P}$ implements the Turing machine as follows.

Each process of program $(\mathcal{P}, N)$ corresponds to a single cell of the tape. Each process uses the following local addresses: q, x and y. Address q is used for sending and receiving the current control state of the Turing machine. The value at address x is the value stored in the tape cell corresponding to the process. The value at address y is non-zero iff the Turing machine is currently at the cell corresponding to the process.

Program $\mathcal{P}$ consists of two parts: initialization and a loop. During the initialization each process checks if it is a distinguished process. If yes, it sets q to the initial control state of the Turing machine, y to 1, and proceeds to the loop. Otherwise, y remains 0, and the process proceeds to the loop anyway. In the loop the process waits until the value at y becomes non-zero. Once it has become non-zero, the process reads the control state from q, value from x, writes new values into q and x in accordance with the transition function of the Turing machine, and performs a shift to the left or right. The shift is implemented as follows. The process sets local y to 0, copies the value at q to q on the remote node, writes 1 to y on that node (using the same queue), and returns to waiting until the value at address y becomes non-zero.

By construction, the Turing machine reaches an accepting state q_f iff the program $(\mathcal{P}, N)$ reaches a state with $\mathsf{mem}(\mathsf{r}, q) = q_f$ for some r and N. □

Theorem 8.31. *Parameterized robustness against PGAS is undecidable.*

Proof. Note that the program $\mathcal{P}$ constructed in the proof of Theorem 8.30 is robust. Similar to the proof of Theorem 8.27, we can reduce parameterized state reachability for program $\mathcal{P}$ to parameterized robustness for a slightly extended program. This renders parameterized robustness against PGAS undecidable. □

Chapter 9

Conclusion

In the thesis we thoroughly studied robustness, a natural correctness criterion for concurrent programs running on architectures with relaxed memory models. A program is robust against a memory model if each its computation under this model has the same dataflow and control-flow dependencies as some sequentially consistent (SC) computation. Robust programs produce the same results under a relaxed memory model and under SC (Theorem 2.19), a property normally expected by a programmer.

Prior studies of robustness concentrated mainly either on detecting non-robustness [25, 26] or ensuring that a program is robust [10, 7, 71]. They did not provide means for deciding whether a given program is robust. The first decidability result for robustness was presented by Bouajjani et al. [21], for TSO memory model only.

The thesis complements the existing knowledge with novel decidability and complexity results for robustness. Our first contribution is a generic approach to solving robustness against a given relaxed memory model. According to this approach, robustness is reduced to checking whether a given program has computations in a particular normal form. The latter is accomplished by checking language emptiness for multiheaded automata. The second contribution is an application of this approach to well-known memory models, including Power, SPARC memory models, and PGAS. For all the above models we derived PSpace algorithms for checking robustness, which makes robustness PSpace-complete for these models, i.e., as hard as SC state reachability.

The relatively low complexity of the robustness problem contrasts with the high complexity of checking state reachability under relaxed memory models. So, state reachability is already non-primitive-recursive for TSO and PSO [14] and undecidable for Power and RMO (Theorems 4.25 and 5.3). As we have shown, checking state-robustness (whether the program has the same set of reachable states under SC and under a relaxed memory model) is just as hard as checking state reachability (Theorem 2.12). All this makes verification under SC, followed by a robustness check, a potentially better alternative to verifying program correctness under a relaxed memory model directly. We support this claim not only by theoretical reasoning about complexity classes, but also by an experimental evidence. Our tool Trencher implements the algorithms for checking and enforcing robustness against TSO and is capable of analyzing a collection of classical concurrent algorithms and data structures in a matter of

several seconds.

9.1 Limitations

We would like to highlight few features of our generic approach to robustness that might limit its applicability.

First, we note that the approach is based on the notion of computations and, therefore, needs an operational specification of the memory model's semantics. This is not a severe limitation, as most memory models have both operational and axiomatic definitions. Moreover, an axiomatic definition usually can be translated to an operational one, similar to how it is done in, e.g., [11] for Power.

Second, the multiheaded automata generating normal-form computations are generally highly non-deterministic: the automata have to guess the memory state at the beginning of each (but the first one) part of the computation (PGAS is a lucky exception from this rule). For a similar reason the non-determinism substantially increases when the multiheaded automaton is intersected with finite automata checking cyclicity of the happens-before relation. This limits the practicality of the algorithms developed within this approach. Furthermore, guessing becomes impossible in the case of infinite data domain or unbounded number of threads, which makes the approach not directly applicable to solving parameterized robustness.

9.2 Future Work

The first direction of possible future work is a corollary of the limitations described above. It would be interesting to check whether memory models other than TSO have the locality property (if a program is not robust, there is a computation with cyclic happens-before relation where at most one thread does reorderings). Locality of PSO would immediately lead to a reduction from robustness against PSO to state reachability in an instrumented program, similar to the reduction for TSO, and would allow to solve parameterized robustness for PSO. Locality of Power could significantly reduce non-determinism in the multiheaded automata generating normal-form computations and make the robustness checking algorithm more practical.

Robustness against ARM memory model is another unexplored area. ARM has a memory model similar to Power [11, 66]. However, some existing ARM processors are known to violate the Power coherence guarantee [11]. In the absence of a well-tested and generally accepted model for ARM we did not consider it in the thesis. Nonetheless, it might be worth doing this in the future, when such a model appears.

Finally, we would like to establish more fine-grained requirements that a memory model must satisfy in order for our generic approach to apply to this memory model. Currently, some requirements are formulated as theorems (Cancellation, Reinsertion), some are used implicitly, in the constructions of the multiheaded automata and in the correctness proofs for these constructions. Crystallizing and assembling these requirements in one place could help us avoid repeating the development done for Power for other memory models.

Bibliography

[1] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *Static Analysis*, pages 164–180. Springer, 2012.

[2] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Counter-example guided fence insertion under TSO. In *TACAS*, LNCS 7214, pages 204–219. Springer, 2012.

[3] S. V. Adve and M. D. Hill. Weak ordering — a new definition. *ACM SIGARCH Computer Architecture News*, 18(2SI):2–14, 1990.

[4] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[5] J. Alglave. *A Shared Memory Poetics*. PhD thesis, University Paris 7, 2010.

[6] J. Alglave, October 2013. Personal communication.

[7] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence. A static analysis approach to automatic fence insertion. In *CAV*, 2014.

[8] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013.

[9] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157. Springer, 2013.

[10] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.

[11] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM TOPLAS*, 36(2):7, 2014.

[12] ARM. ARM architecture reference manual. ARMv7-A and ARMv7-R edition, 2012.

[13] D. Aspinall and J. Ševčík. Formalising Java's data race free guarantee. In *Theorem Proving in Higher Order Logics*, pages 22–37. Springer, 2007.

[14] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[15] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models. In *ESOP*, LNCS. Springer, 2012.

[16] M. F. Atig, A. Bouajjani, C. Leonardsson, and R. Meyer, May 2013. Personal communication.

[17] M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, pages 99–115. Springer, 2011.

[18] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Model Checking and Software Verification*, pages 113–130. Springer, 2000.

[19] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, 2002.

[20] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[21] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against Total Store Ordering. In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.

[22] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403. ACM, 2009.

[23] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In *Static Analysis*, pages 164–178. Springer, 1999.

[24] S. Burckhardt, R. Alur, and M. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.

[25] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

[26] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*, volume 6605 of *LNCS*, pages 11–25. Springer, 2011.

[27] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[28] G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. A theory of partitioned global address spaces. In *FSTTCS*, volume 24 of *LIPIcs*, pages 127–139, 2013.

[29] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *PGAS*, page 2. ACM, 2010.

[30] UPC Consortium. UPC language specification v1.2. Technical report, Lawrence Berkeley National Laboratory, 2005.

[31] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Software Engineering*, pages 439–448. IEEE, 2000.

[32] International Business Machines Corporation. Power ISA^{TM} version 2.07, May 2013.

[33] E. Derevenetc and R. Meyer. Robustness against Power is PSPACE-complete. In *ICALP*, volume 8573 of *LNCS*, pages 158–170. Springer, 2014.

[34] D. Dice. A race in locksupport park() arising from weak memory models. `https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park`, Nov 2009.

[35] E. W. Dijkstra. *Cooperating sequential processes*. Springer, 2002.

[36] Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 241–256. Springer, 1999.

[37] MPI Forum. MPI: A message-passing interface standard version 3.0. Technical report, University of Tennessee, Knoxville, 2012.

[38] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[39] Global address space programming interface. `http://www.gaspi.de/`.

[40] J. Gosling, B. Joy, G. Steele, G. Brancha, and A. Buckley. Java language specification, 2014.

[41] S. Greibach and J. Hopcroft. Scattered context grammars. *Journal of Computer and System Sciences*, 3(3):233–247, 1969.

[42] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. MKP, 2008.

[43] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Zh. Su, and K. A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, UC Berkeley, 2005.

[44] G. J. Holzmann. The model checker SPIN. *IEEE Tr. Soft. Eng.*, 23:279–295, 1997.

[45] G. J. Holzmann. The engineering of a model checker: the Gnu i-protocol case study revisited. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 232–244. Springer, 1999.

[46] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on computing*, 17(5):935–938, 1988.

[47] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, February 2014.

[48] JTC1/SC22/WG14. ISO/IEC 9899:2011 Information technology — Programming languages — C. Technical report, JTC/ISO, 2011.

[49] JTC1/SC22/WG21. ISO/IEC 14882:2011, Information technology — Programming languages — C++. Technical report, JTC/ISO, 2011.

[50] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[51] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *IEEE Real-Time Systems Symposium*, pages 131–137. IEEE, 1993.

[52] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.

[53] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FMCAD*, pages 111–119. IEEE, 2010.

[54] M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, pages 187–198. ACM, 2011.

[55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[56] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[57] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1), 1987.

[58] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pages 227–242. ACM, 2009.

[59] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–353. Springer, 2013.

[60] R. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

[61] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12):717–721, 1975.

[62] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.

[63] R. Machado and C. Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science — Research and Development*, 23(3-4):125–132, 2009.

[64] S. Mador-Haim, R. Alur, and M. M. K Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, pages 273–287. Springer, 2010.

[65] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, pages 495–512. Springer, 2012.

[66] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`. Draft.

[67] P. E. McKenney. Memory ordering in modern microprocessors, part II. *Linux Journal*, 137, September 2005.

[68] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, volume 1586 of *LNCS*, pages 533–546. Springer, 1999.

[69] The UPC NAS parallel benchmarks. `http://upc.gwu.edu/download.html`.

[70] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[71] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

[72] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report CL-TR-745, University of Cambridge, 2009.

[73] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *SC'11*, page 51. ACM, 2011.

[74] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[75] G. Petri. *Operational semantics of relaxed memory models.* PhD thesis, Nice, 2010.

[76] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comp. Sci.*, 6:223–231, 1978.

[77] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP*, pages 161–172. ACM, 2007.

[78] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

[79] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM TOPLAS*, 10(2):282–312, 1988.

[80] R. L. Sites and R. T. Witek. *Alpha AXP architecture reference manual*. Digital Press, second edition, 1995.

[81] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP*, pages 2–13. ACM, 2005.

[82] R. Szelepcstnyi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–100, 1987.

[83] V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *SAS*, volume 6887 of *LNCS*, pages 146–162. Springer, 2011.

[84] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

List of Acronyms

API Application Programming Interface

ARM Advanced RISC Machine

ARMCI Aggregate Remote Memory Copy Interface

CPU Central Processing Unit

DEC Digital Equipment Corporation

DRF Data-Race Freedom

FIFO First In — First Out

GASNet Global-Address Space Networking

GASPI Global Address Space Programming Interface

GPI Global address space Programming Interface

HPC High-Performance Computing

MP Message Passing

MPI Message Passing Interface

NIC Network Interface Controller

PGAS Partitioned Global Address Space

POWER Performance Optimization With Enhanced RISC

PSO Partial Store Order

RAM Random Access Memory

RDMA Remote Direct Memory Access

RISC Restricted (Reduced) Instruction Set Computer

RMO Relaxed Memory Order

SAT SATisfiability problem

SB Store Buffering

SC Sequential Consistency

SHMEM Symmetric Hierarchical MEMory access

SMT Satisfiability Modulo Theory

SPARC Scalable Processor ARChitecture

SPMD Single Program, Multiple Data

TSO Total Store Order

UPC Unified Parallel C

Appendix A

Benchmarking Memory Fences

In order to estimate the overhead introduced by the insertion of memory fences on x86 processors, we developed a tool called `rdtsc-mfence`. The tool executes a pattern consisting of several stores, optionally followed by a memory fence, a great number of times and measures the average number of cycles required for one execution of the pattern.

Benchmarking We ran the tool on several x86-64 CPUs running in 64-bit mode and used the following patterns.

- `movl; ... movl; mfence;`
- `movl; ... movl; lock addl $0,(%rsp);`
- `movl; ... movl;`

Here, `movl` are stores of a 32-bit constant to a global variable, addressed relative to `rip`. The instruction `lock addl $0,(%rsp)` is an atomic increment of the value at the stack top by 0. It has no side-effects on memory other than flushing the store buffer. Surprisingly, it turns out to be faster than the specially designed `mfence` on some processors. We ran the above code in a single thread and in two threads, running on the same physical CPU and on different CPUs (where applicable). See the charts Fig. A.1–A.10 with the benchmarking results below.

Discussion From the charts we can draw the following empirical conclusions.

- Inserting a memory fence never improves the running time in the absence of the memory contention (when there is a single thread running).
- Inserting a memory fence almost never improves the running time when two threads are running.
- Executing a memory fence instruction takes some significant time even when the TSO store buffer is empty.
- Performance of a single thread cannot be improved by executing more fences, even with smaller number of buffered stores.

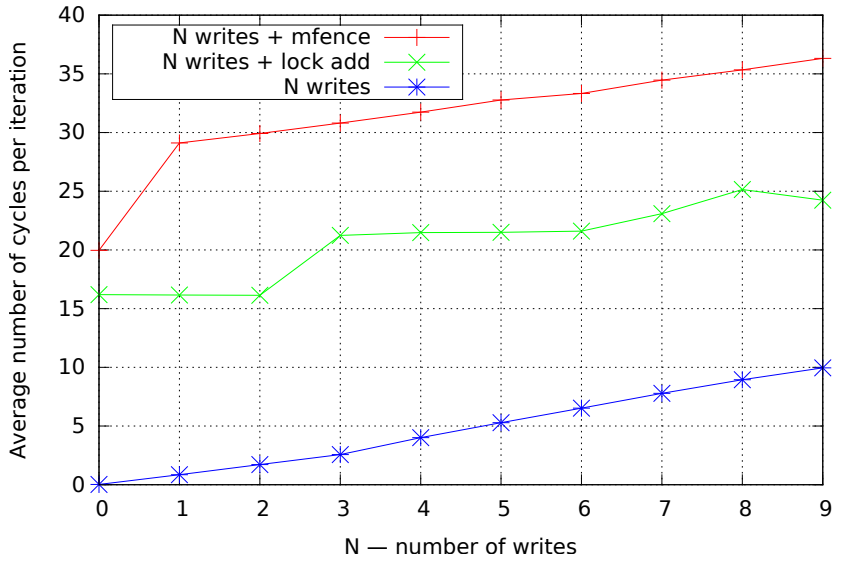

Figure A.1: Results of benchmarking memory fences on Intel Core i5 M650 CPU @ 2.67GHz (1 thread).

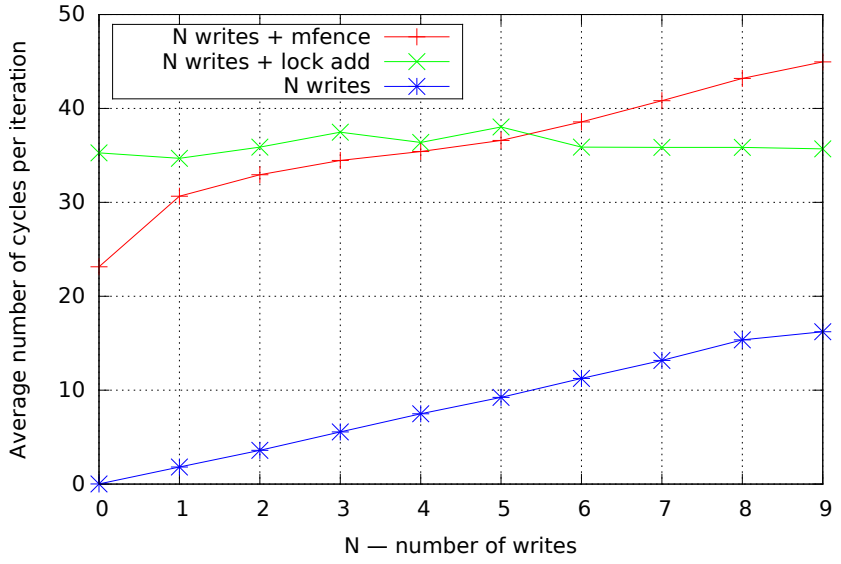

Figure A.2: Results of benchmarking memory fences on Intel Core i5 M650 CPU @ 2.67GHz (2 threads).

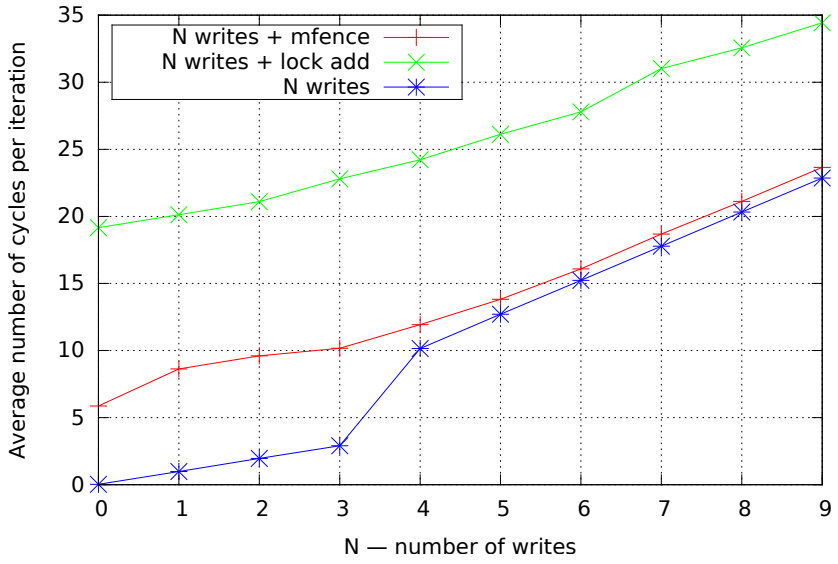

Figure A.3: Results of benchmarking memory fences on Intel Core 2 Duo P8700 CPU @ 2.53GHz (1 thread).

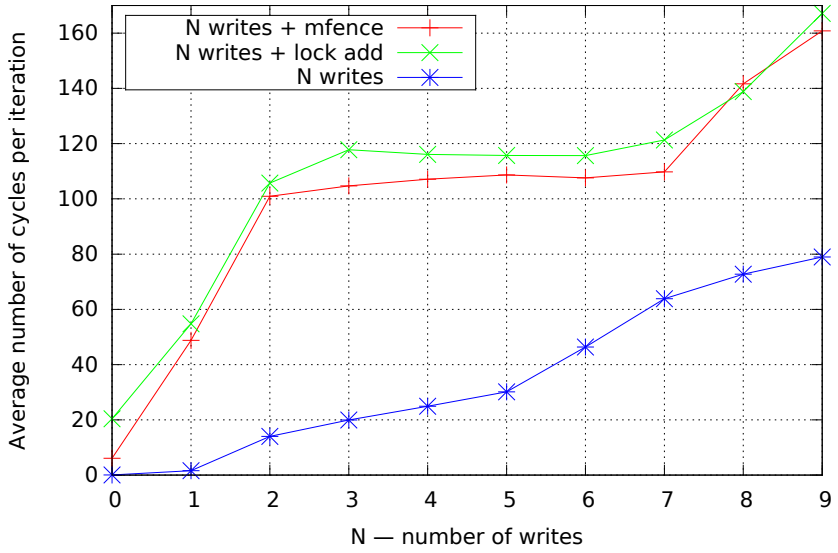

Figure A.4: Results of benchmarking memory fences on Intel Core 2 Duo P8700 CPU @ 2.53GHz (2 threads).

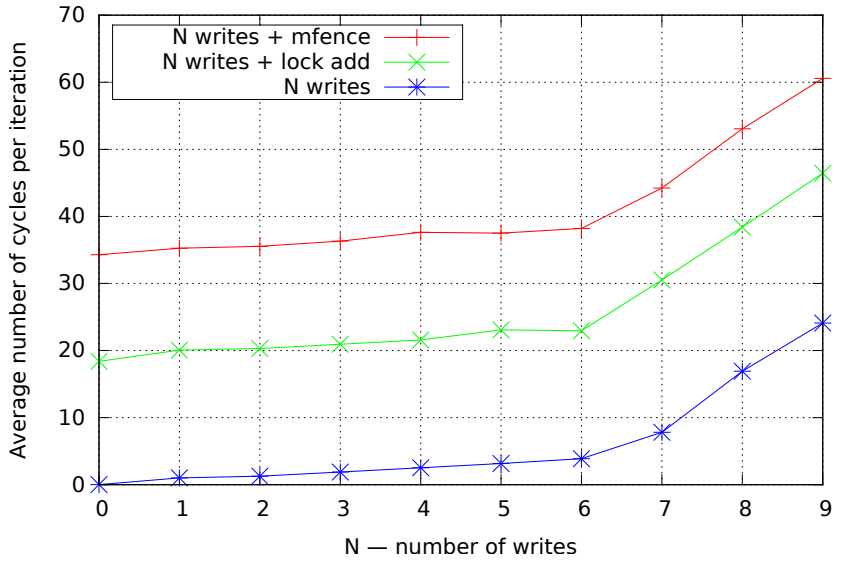

Figure A.5: Results of benchmarking memory fences on Mobile AMD Sempron Processor 3400+ (1 thread).

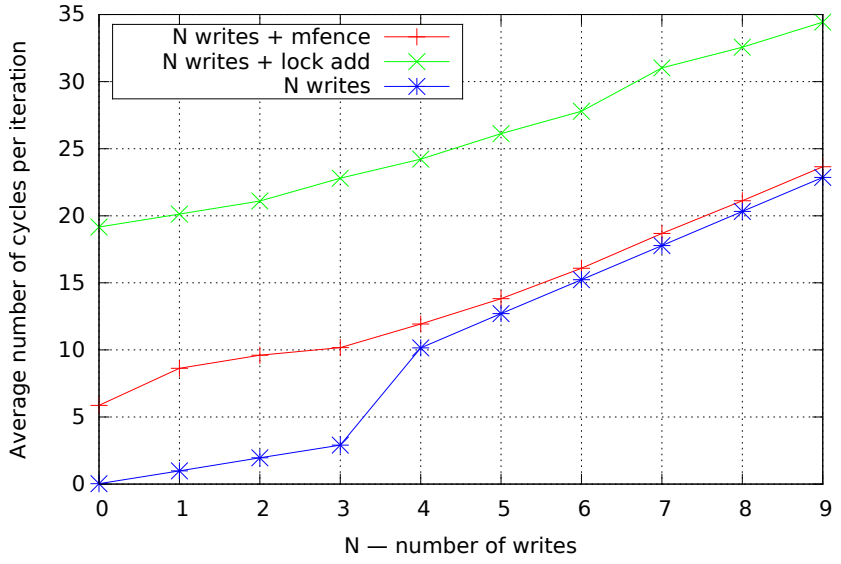

Figure A.6: Results of benchmarking memory fences on Intel Xeon X5650 CPU @ 2.67GHz (1 thread).

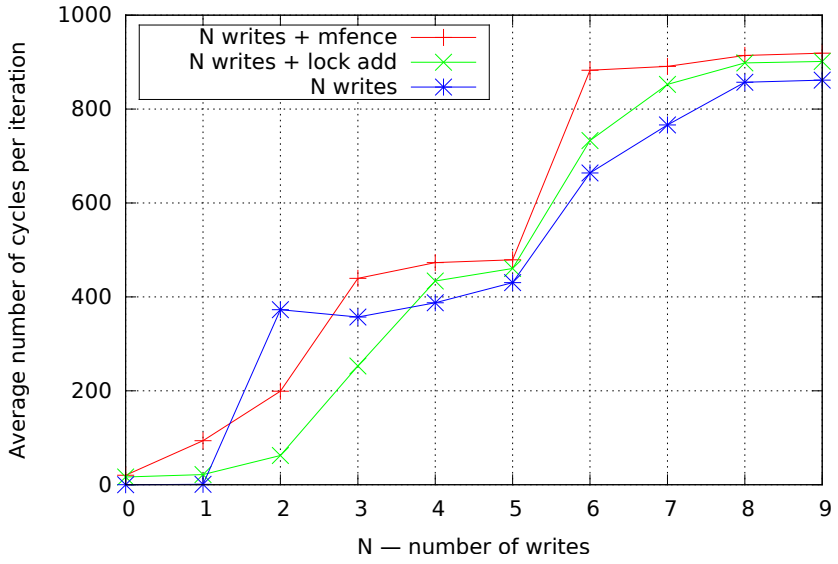

Figure A.7: Results of benchmarking memory fences on Intel Xeon X5650 CPU @ 2.67GHz (2 threads on different CPU sockets).

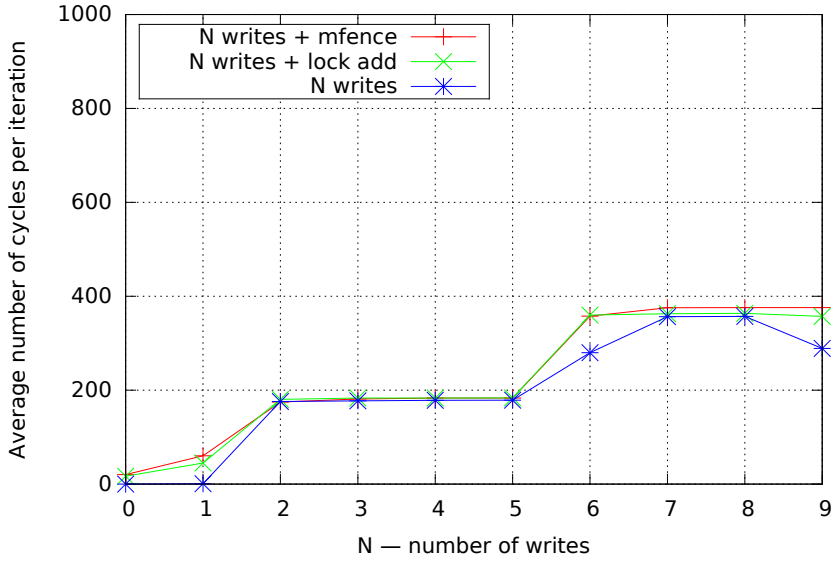

Figure A.8: Results of benchmarking memory fences on Intel Xeon X5650 CPU @ 2.67GHz (2 threads on the same CPU socket).

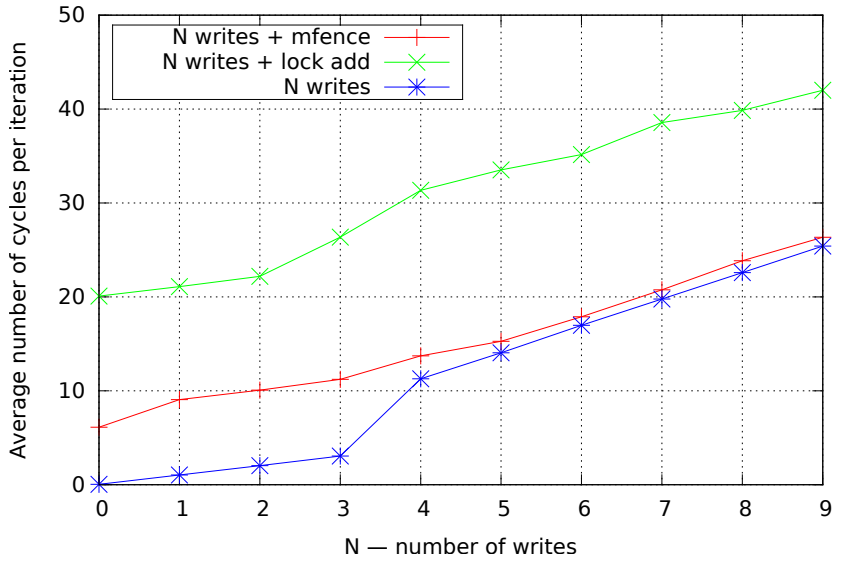

Figure A.9: Results of benchmarking memory fences on Intel Xeon E5420 CPU @ 2.50GHz (1 thread).

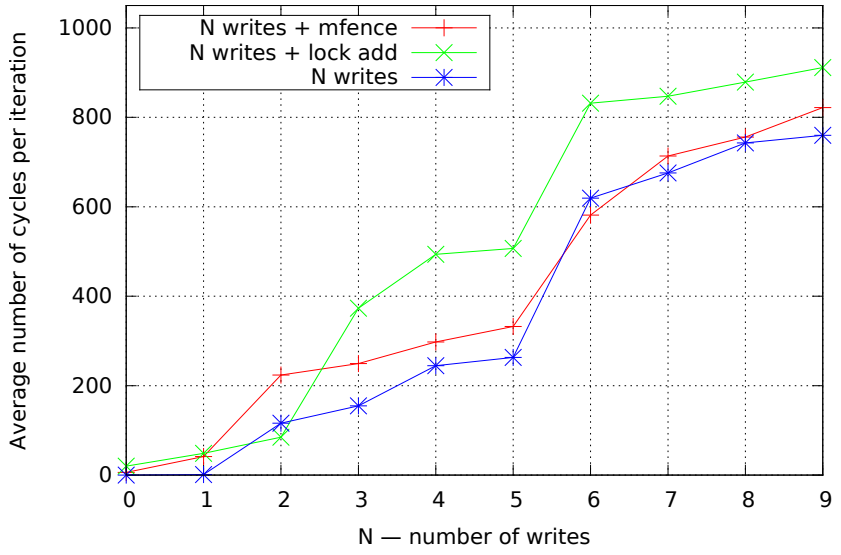

Figure A.10: Results of benchmarking memory fences on Intel Xeon E5420 CPU @ 2.50GHz (2 threads).

Appendix B

Benchmarking Trencher with SPIN

Resolving a reachability query using the SPIN [44] model checker involves several steps. First, the query must be translated to a Promela program. The Promela program is given to SPIN, which generates a C program called verifier. Next, the verifier is translated to an executable file using a C compiler. Finally, running the verifier answers the query.

The Trencher tool, described in Chapter 7, initially used SPIN as a back-end model checker. We ran that version of Trencher on the examples from Section 7.2 and measured the time taken by Trencher, SPIN, C compiler, and the verifier. The execution was performed on a 4-core machine equipped with Intel(R) Core(TM) i5 CPU M 560 @ 2.67GHz. The results are shown in Figure B.1 and Figure B.2. Clearly, the compilation time dominates the time spent in all other phases on virtually all tests.

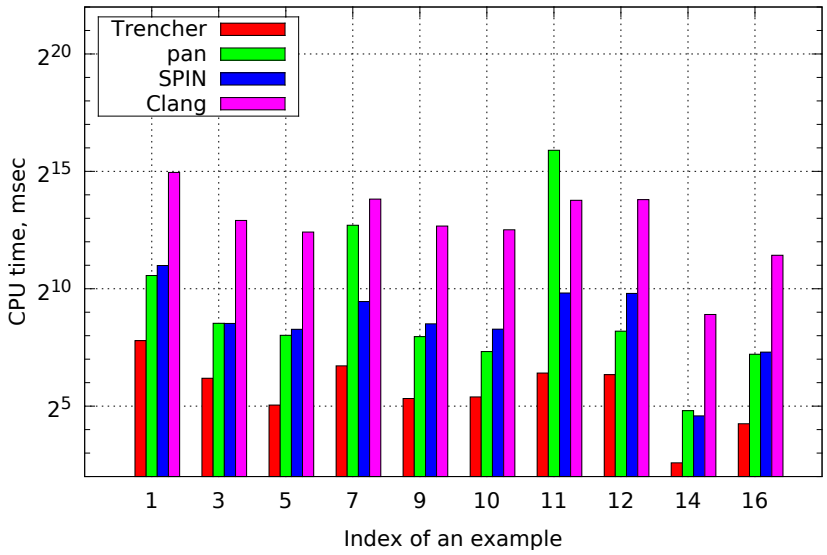

Figure B.1: Time spent by Trencher (using SPIN as a back-end model checker), verifier (pan), SPIN, and the C compiler (Clang 3.5) while computing minimal fence sets for the examples from Table 7.3. The verifier was compiled without optimizations. Only examples with the total running time over 10ms are shown.

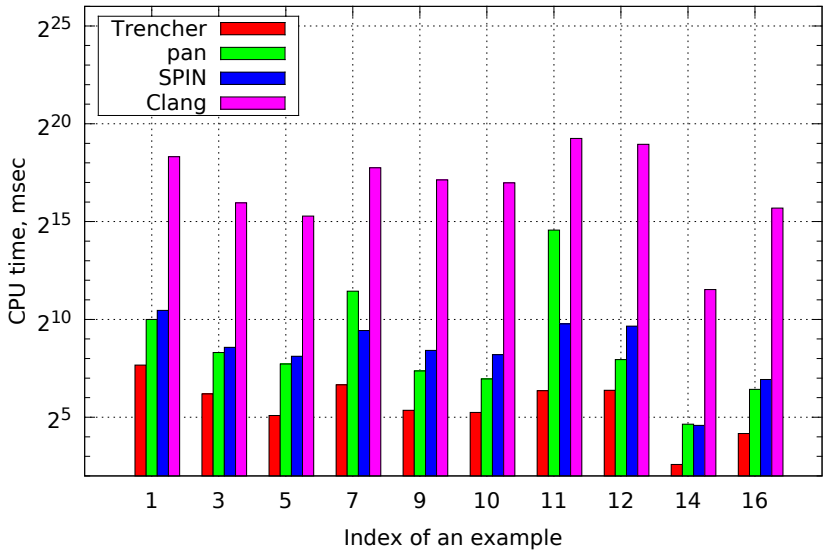

Figure B.2: Time spent by Trencher (using SPIN as a back-end model checker), verifier (pan), SPIN, and the C compiler (Clang 3.5) on computing minimal fence sets for the examples from Table 7.3. The verifier was compiled with `-O2`. Only examples with the total running time over 10ms are shown.

Egor Derevenetc

✉ *yegor.derevenets@gmail.com*

Education

Mar 2011–Apr 2015 **Ph.D. Student**, *University of Kaiserslautern*, Computer Science Department, Concurrency Theory Group.
Topic: Robustness against relaxed memory models.
Supervisor: Prof. Dr. Roland Meyer.

Sep 2005–Jun 2010 **Specialist**, *Lomonosov Moscow State University*, Computational Mathematics and Cybernetics Department.
Graduated with honours (diploma GPA is 4.96 of 5.0).
Speciality: Applied Mathematics and Computer Science.
Qualifications: Mathematician, System Programmer.
Diploma thesis topic: Reconstruction of C++ control flow structures from a low-level program.
Supervisor: Alexander Chernov.

Sep 1995–May 2005 **School Education**.