A Theory of Partitioned Global Address Spaces

Georgel Calin\textsuperscript{1}, Egor Derevenetc\textsuperscript{2}, Rupak Majumdar\textsuperscript{3}, and Roland Meyer\textsuperscript{1}

\begin{itemize}
\item \textsuperscript{1} University of Kaiserslautern, \{calin, meyer\}@cs.uni-kl.de
\item \textsuperscript{2} Fraunhofer ITWM, egor.derevenetc@itwm.fraunhofer.de
\item \textsuperscript{3} MPI-SWS, rupak@mpi-sws.org
\end{itemize}

Abstract

Partitioned global address space (PGAS) is a parallel programming model for the development of high-performance applications on clusters. It provides a global address space partitioned among the cluster nodes, and is supported in programming languages like C, C++, and Fortran by means of APIs. In this paper we provide a formal model for the semantics of single instruction, multiple data programs using PGAS APIs. Our model reflects the main features of popular real-world APIs such as SHMEM, ARMCI, GASNet, GPI, and GASPI.

A key feature of PGAS is the support for one-sided communication: a node may directly read and write the memory located at a remote node, without explicit synchronization with the processes running on the remote side. One-sided communication increases performance by decoupling process synchronization from data transfer, but requires the programmer to reason about appropriate synchronizations between reads and writes. As a second contribution, we propose and investigate robustness, a criterion for correct synchronization of PGAS programs. Robustness corresponds to acyclicity of a suitable happens-before relation defined on PGAS computations. The requirement is finer than the classical data race freedom and rules out most false error reports.

Our main technical result is an algorithm for checking robustness of PGAS programs. The algorithm makes use of two insights. Using combinatorial arguments we first show that, if a PGAS program is not robust, then there are computations in a certain normal form that violate happens-before acyclicity. Intuitively, normal-form computations delay remote accesses in an ordered way. We then devise an algorithm that checks for cyclic normal-form computations. Essentially, the algorithm is an emptiness check for a novel automaton model that accepts normal-form computations in streaming fashion. Altogether, we prove the robustness problem is \textsf{PSPACE}-complete.

1 Introduction

Partitioned global address space (PGAS) is a parallel programming model for the development of high-performance software on clusters. The PGAS model provides a global address space to the programmer that is partitioned among the cluster nodes (see Figure 1(b)). Nodes can read and write their local memories, but additionally access the remote address space through (synchronous or asynchronous) API calls. PGAS is a popular programming model, and supported by many PGAS APIs, such as SHMEM \textsuperscript{9}, ARMCI \textsuperscript{20}, GASNET \textsuperscript{4}, GPI \textsuperscript{18}, and GASPI \textsuperscript{14}, as well as by languages for high-performance computing, such as UPC \textsuperscript{10}, Titanium \textsuperscript{15}, and Co-Array Fortran \textsuperscript{22}.

A key ingredient of PGAS APIs is their support for one-sided communication. Unlike in traditional message passing interfaces, a node may directly read and write the memory located at a remote node without explicit synchronization with the remote side. One-sided communication can be efficiently implemented on top of networking hardware featuring

\textsuperscript{*} The second author was granted by the Competence Center High Performance Computing and Visualization (CC-HPC) of the Fraunhofer Institute for Industrial Mathematics (ITWM). The work was partially supported by the PROCOPE project ROIS: Robustness under Realistic Instruction Sets.
**A Theory of Partitioned Global Address Spaces**

```plaintext
int x = 1, y = 0;
write(x, rightNeighbourRank, y, myWriteQ);
barrier();
assert(y == 1);
```

**Figure 1** (a) Program `1to1` is the **compute and exchange results** idiom often found in PGAS applications. Each process copies an integer value to its neighbour. `write` asks the hardware to copy the value of address `x` to `y` on the right neighbouring node. `barrier` blocks until all processes reach the barrier. The assertion can fail, as the `barrier` may execute before the `write` completes. (b) PGAS architecture — NIC stands for network interface controller.

remote direct memory access (RDMA), and increases performance of PGAS programs by avoiding unnecessary synchronization between the sender and the receiver [18, 13].

However, the use of one-sided communication introduces additional non-determinism in the ordering of memory reads and writes, and makes reasoning about program correctness harder. Figure 1(a) demonstrates a subtle bug arising out of improper synchronizations: while the barriers ensure all processes are at the same control location, the remote writes may or may not have completed when address `y` is accessed after the barrier.

We make two contributions in this paper. First, we provide a core calculus of PGAS APIs that models concurrent processes sharing a global address space and accessing remote memory through one-sided reads and writes. Despite the popularity of PGAS APIs in the high-performance computing community, to the best of our knowledge, there are no formal models for common PGAS APIs.

Second, we define and study a correctness criterion called robustness for PGAS programs. To understand robustness, we begin with a classical and intuitive correctness condition, sequential consistency [17]. A computation is sequentially consistent if its memory accesses happen atomically and in the order in which they are issued. Sequential consistency is too strong a criterion for PGAS programs, where time is required to access remote memory and accesses themselves can be reordered. Robustness is the weaker notion that all computations of the program have the same happens-before (data and control) dependencies [25] as some sequentially consistent computation. Our notion of robustness captures common programming error patterns [12, 19], and is derived from a similar notion in shared memory multiprocessing [25]. Related correctness criteria have been proposed for weak memory models [7, 23, 2, 3, 6, 8, 5].

A simpler correctness property would be data race freedom (DRF), in which no two processes access the same address at the same time, with at least one access being a write [1]. Indeed, data race free programs are sequentially consistent. Unfortunately, DRF is too strong a requirement in practice [24], and leads to numerous false alarms. Many common synchronization idioms for PGAS programs, such as producer-consumer synchronization, and many concurrent data structure implementations, contain benign data races. Instead, the notion of robustness captures the intuitive requirement that, even when events are reordered in a computation, there are no causality cycles. Our notion of causality is the standard happens-before relation from [25].

We study the algorithmic verification of robustness. Our main result is that robustness is decidable (actually PSPACE-complete) for PGAS programs, assuming a finite data domain and finite memory. Note that our model of PGAS programs is infinite-state even when the data domain is finite: one-sided communication allows unboundedly many requests to be in flight simultaneously (a feature modeled in our formalism using unbounded queues).
Our decidability result uses two technical ingredients. First, we show that among all computations violating robustness, there is always one in a certain normal form. The normal form partitions the violating computation into phases: the first phase initiates memory reads and writes, and the latter phases complete the reads and writes in the same order in which they were initiated.

Second, we provide an algorithm to detect violating computations in this normal form. We take a language-theoretic view, and introduce a multiheaded automaton model which can accept violating computations in normal form. Then the problem of checking robustness reduces to checking emptiness for multiheaded automata. Interestingly, since the normal form maintains orderings of accesses, the multiple heads can be exploited to accept violating computations without explicitly modeling unbounded queues of memory access requests. The resulting class of languages contains non-context-free ones (such as $a^n b^n c^n$), but retains sufficient decidability properties. Altogether this yields a PSPACE decision procedure for checking robustness of programs using PGAS APIs.

For lack of space, full constructions and proofs are given in the appendix.

Related Work Although PGAS APIs are popular in the high-performance computing community [4, 9, 14, 18, 20], to the best of our knowledge, no previous work provides a unifying formal semantics that incorporates one-sided asynchronous communication. As for synchronization correctness, only recently Park et al. proposed a testing framework for data race detection and implemented it for the UPC language [24]. However, the authors argue that many data races are actually not harmful, a claim they support through the analysis of the NAS Parallel Benchmarks [21]. For this reason, in contrast to data race freedom [1], we consider robustness as a more precise notion of appropriate synchronization.

The robustness problem was posed by Shasha and Snir [25] for shared memory multiprocessors. They showed that non sequentially consistent computations have a happens-before cycle. Alglave and Maranget [2, 3] extended this result. They developed a general theory for reasoning about robustness problems, even among different architectures. Owens [23] proposed a notion of appropriate synchronization that is based on triangular data races. Compared to robustness, triangular race freedom requires heavier synchronization, which is undesirable for performance reasons.

We consider here the algorithmic problem of checking robustness. For programs running on weak memory models the problem has been addressed in [7, 8, 3], but none of these works provides a (sound and complete) decision procedure. The first complete algorithm for checking robustness of programs running on Total Store Ordering (TSO) architectures was given in [6]. It is based on the following locality property. If a TSO program is not robust, then there is a violating computation where only one process delays commands. This insight leads to a reduction of robustness to reachability in the sequential consistency model [5]. PGAS programs allow more reorderings than TSO ones and, as a consequence, locality does not hold. Instead, our decision procedure relies on a complex normal form for computations and on a sophisticated automata-theoretic algorithm to look for normal-form violations.

2 PGAS Programs

2.1 Features of PGAS Programs

PGAS programs are single instruction, multiple data programs running on a cluster (see Figure 1b). At run time, a PGAS program consists of multiple processes executing the same code on different nodes. Each process has a rank, which is the index of the node it runs on. The processes can access a global address space partitioned into local address spaces.
for each process. Local addresses can be accessed directly. Remote addresses (addresses belonging to different processes) are accessed using API calls, which come in different flavors.

SHMEM [9] provides synchronous remote reads where the invoking process waits for completion of the command. Remote write commands are asynchronous, and no ordering is guaranteed between writes, even to the same remote node. The ordering can, however, be enforced by a special fence command.

ARMCI [20] features synchronous as well as asynchronous read and write commands. The asynchronous variants of the commands return a handle that can be waited upon. When the wait on a read handle is over, the data being read has arrived and is accessible. When the wait on a write handle is over, the data being written has been sent to the network but might not have reached its destination. Unlike operations to different nodes, operations to the same remote node are executed in their issuing order.

GASNet [4], like ARMCI, provides both synchronous and asynchronous versions of reads and writes. Commands return a handle that can be waited upon, and a return from a wait implies full completion of the operation. The order in which asynchronous operations complete is intentionally left unspecified.

GPI [18] and GASPI [14] only support asynchronous read and write commands. Each read or write operation is assigned a queue identifier. In GPI, operations with the same queue id and to the same remote node are executed in the order in which they were issued; in GASPI this guarantee does not hold. One can wait on a queue id, and the wait returns when all commands in the queue are fully completed, on both the local and the remote side.

Summing up, in a uniform PGAS programming model it should be possible to perform synchronous and asynchronous data transfers,

assign an asynchronous operation a handle or a queue id,

wait for completion of an individual command or of all commands in a given queue,

enforce ordering between operations.

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

2.2 Syntax of PGAS Programs

We define PGAS programs and their semantics in terms of automata. A (non-deterministic) automaton is a tuple \( A = (S, \Sigma, \Delta, s_0, F) \), where \( S \) is a set of states, \( \Sigma \) is a finite alphabet, \( \Delta \subseteq S \times (\Sigma \cup \{\epsilon\}) \times S \) 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 the set of states is finite. We write \( s_1 \xrightarrow{a} s_2 \) if \((s_1, a, s_2) \in \Delta\), and extend the relation to computations \( \sigma \in \Sigma^* \) in the expected way. The language of the automaton is \( L(A) := \{ \sigma \in \Sigma^* \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F \} \). We write \(|\sigma|\) for the length of a computation \( \sigma \in \Sigma^* \), and use \( \text{succ}(\sigma) \) to denote the successor relation among the letters in \( \sigma \). We write \( a <_\sigma b \) if \( \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3 \) for some \( \sigma_1, \sigma_2, \sigma_3 \in \Sigma^* \).

A PGAS program \((P, N)\) consists of a program code \( P \) and a fixed number \( N \geq 1 \) of cluster nodes. The program code \( P := (Q, \text{CMD}, \mathcal{I}, q_0, Q) \) is a finite automaton with a set of control states \( Q \), all of them are final, initial state \( q_0 \), and a set of transitions \( \mathcal{I} \) labeled with commands CMD.

Let \( \text{DOM}, \text{ADR}, \) and \( \text{QUE} \) be finite domains of values (containing a value \( 0 \)), addresses, and queue identifiers, respectively. Let \( \text{REG} \) be a finite set registers that take values from \( \text{DOM} \). The grammar of commands is given in Figure 2. For simplicity, we will assume \( \text{DOM} = \text{ADR} = \text{QUE} \). The set of expressions is defined over constants from \( \text{DOM} \), registers from \( \text{REG} \), and (unspecified) operators over \( \text{DOM} \). The set of commands \( \text{CMD} \) includes local
Figure 2 Syntax of commands. (reg) ranges over REG; expressions (expr), local addresses (local-adr), remote addresses (remote-adr), and queue identifiers (que-id) range over expressions; ranks (rank) over $\overline{1,N}$-valued expressions.

The semantics of a PGAS program $(\mathcal{P}, N)$ is defined using a state-space automaton $X(\mathcal{P}, N) := (S_X, E, \Delta_X, s_0_X, F_X)$. A state $s \in S_X$ is a tuple $s = (st, m, fa, fb)$, where state configuration $st \colon RNK \rightarrow Q$ maps each process to its current control state, memory configuration $m \colon RNK \times (REG \cup ADR) \rightarrow DOM$ maps each process to the values stored in each register and at each address, queue configuration $fa \colon RNK \times QUE \rightarrow (RNK \times ADR \times RNK \times ADR)^*$ maps each process to remote read and write requests that were issued, and $fb \colon RNK \times QUE \rightarrow (RNK \times ADR \times DOM)^*$ contains values to be transferred.

The initial state is $s_{0_X} := (st_0, m_0, fa_0, fb_0)$, where for all ranks $r \in RNK$, registers and addresses $a \in REG \cup ADR$, and queue identifiers $q \in QUE$, we have $st_0(r) := q_0$, $m_0(r, a) := 0$, and $fa_0(r, q) := \epsilon := fb_0(r, q)$. The set of final states is $F_X := \{(st, m, fa, fb) \in S_X \mid fa(r, q) = \epsilon = fb(r, q) \text{ for all } r \in RNK, q \in QUE\}$. The semantics of commands ensures queues can always be emptied, so acceptance with empty queues is not a restriction.

The alphabet of $X(\mathcal{P}, N)$ is the set of events $E := K \times RNK \times ((RNK \times ADR) \cup \{\bot\})$ with event kinds $K := \{load, store, assign, assume, read, write, popa, popb, bar\}$. Consider an event $e = (k, r, (r_a, a)) \in E$. We use $kind(e) = k$ to determine the kind of the event, $rank(e) = r$ for the rank of the process that produced the event, and $addr(e) = (r_a, a)$ to obtain the rank and the address that are accessed by the event. If $kind(e) \in \{load, popa\}$, then $e$ is said to be a read of $(r_a, a)$. If $kind(e) \in \{store, popb\}$, then $e$ is a write of address $addr(e)$.

Table 1 shows a subset of the transition relation $\Delta_X$; other rules are similar. 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.
A Theory of Partitioned Global Address Spaces

\[ \text{cmd} = r \leftarrow \text{mem}[e_a] \]
\[ s \xrightarrow{\text{cmd} = \text{write}(e_{a_2}, e_{a_3}, e_{a_4})} (st', m((r, r) := m(r, e_{a_2})), fa, fb) \]
\[ s \xrightarrow{\text{cmd} = \text{write}(e_{a_2}, e_{a_3}, e_{a_4})} (st', m, fa[(r, e_{a_2}) := \alpha \cdot (r, e_{a_3}, e_{a_4})], fb) \]
\[ fa(r, q) = (r_1, a_1, r_2, a_2) \cdot \alpha \quad fb(r, q) = \beta' \]
\[ s \xrightarrow{\text{popa}, r, (r, a_1)} (st, m, fa[(r, q) := \alpha], fb[(r, q) := \beta \cdot (r_1, a_1, m(r_1, a_1))]) \]
\[ fb(r, q) = (r_2, a_2, v) \cdot \beta \]
\[ s \xrightarrow{\text{popb}, r, (r, a_1)} (st, fa[(r, q) := v], fb[(r, q) := \beta]) \]
\[ s \xrightarrow{\text{bar} \cdot \text{bar} \cdot \text{load} \cdot \text{popb} \cdot \text{popb}} (st', m, fb) \]

Table 1: Transition rules for \( X(P, N) \), given \( q_1 \xrightarrow{\text{cmd}} q_2 \) and current state \( s = (st, m, fa, fb) \) with \( st(r) = q_1 \). We set \( st' := st\hat{r} := q_2 \) to update \( st \) so that process \( r \) is at \( q_2 \). \( \hat{e} \) denotes the evaluation of expression \( e \) in memory configuration \( m \).

Example 1. Consider PGAS program \( C(P, N) := \mathcal{L}(X(P, N)) \subseteq E^* \) is the set of computations of the state-space automaton.

Example 1. Consider PGAS program \((\text{1to1}, 2)\) with the program code from Figure 1(a) being run on two nodes. It has the following computation:

\[ \tau_{\text{1to1}} = \text{write} \cdot \text{write} \cdot \text{popa} \cdot \text{popa} \cdot \text{bar} \cdot \text{bar} \cdot \text{load} \cdot \text{popb} \cdot \text{popb} \]

Bold events belong to the process with rank 2, the other events to the process with rank 1. We have \( \text{addr}(\text{popa}) = (1, x) \), \( \text{addr}(\text{popb}) = (2, y) \). Symmetrically, \( \text{addr}(\text{popa}) = (2, x) \) and \( \text{addr}(\text{popb}) = (1, y) \). The \text{assert} in Figure 1 is a shortcut for a combination of load and assume, and in this computation \( \text{addr}(\text{load}) = (1, y) \).

2.4 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 ARMC1 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.

3 Robustness: A Notion of Appropriate Synchronization

We now define robustness, a correctness condition for PGAS programs. Robustness is a weaker criterion than requiring all computations to be sequentially consistent [17]; it allows for reordering of events as long as there are no causality cycles. As causality relation, we
We show that a PGAS program is not robust if and only if it has a violating computation. adopt the happens-before relation \[ \tau \in C(P, N). \] Its happens-before relation is the union of the three relations we define next, \[ \rightarrow_h (\tau) := \rightarrow_{po} \cup \rightarrow_{cf} \cup \rightarrow_{\leftrightarrow}. \]

The program order relation \( \rightarrow_{po} \) is the union of the program order relations for all processes: \[ \rightarrow_{po} := \bigcup_{r \in \text{RNC}} \rightarrow'_{po}. \] Relation \( \rightarrow'_{po} \) gives the order in which events were issued in process \( r \). Formally, let \( \tau' \) be the subsequence of all events \( e \) in \( \tau \) such that \[ \text{rank}(e) = r \text{ and kind}(e) \notin \{\text{popa}, \text{popb}\}. \] Then \[ \rightarrow'_{po} := \text{succ}(\tau'). \]

The conflict relation \( \rightarrow_{cf} \) orders conflicting accesses to the same address. Let \( \tau := a \cdot e_1 \cdot b \cdot e_2 \cdot c_3 \cdot a_1 \cdot e_3 \cdot b_4 \cdot c_5 \cdot a_2 \cdot e_4 \cdot b_5 \cdot c_6 \cdot a_3 \cdot e_5 \cdot b_6 \cdot c_7 \cdot a_4 \cdot e_6 \cdot b_7 \cdot c_8 \); where \( e_1 \) and \( e_2 \) access the same address, and at least one of them is a write: \[ \text{addr}(e_1) = \text{addr}(e_2) = (r, a), \text{kind}(e_1) \in \{\text{store, popb}\} \text{ or kind}(e_2) \in \{\text{store, popb}\}. \] If there is no \( e \in \beta \) such that \[ \text{addr}(e) = (r, a) \text{ and kind}(e) \in \{\text{store, popb}\}, \] then \( e_1 \rightarrow_{cf} e_2 \).

The identity relation \( \leftrightarrow \) identifies events corresponding to the same command. Let \( e \) be a remote read or write event, \( \text{kind}(e) \in \{\text{read, write}\} \), and \( e_1 \) and \( e_2 \) be the corresponding requests, \( \text{kind}(e_1) = \text{popa} \) and \( \text{kind}(e_2) = \text{popb} \). Then we have \( e \leftrightarrow e_1 \leftrightarrow e_2 \). In a similar way, \( \leftrightarrow \) identifies matching barrier events in different processes.

We say a computation \( \tau \) is violating if the associated happens-before relation contains a non-trivial cycle, i.e., a cycle that is not included in \( \leftrightarrow \). Violating computations violate sequential consistency. The robustness problem amounts to proving the absence of violations.

**ROB**

Given a program \((P, N)\), show that no computation \( \tau \in C(P, N) \) is violating.

**Example 2.** The happens-before relation of computation \( R_{1to1} \) is depicted in Figure 3. It is cyclic, therefore \( R_{1to1} \) is violating and \( (1to1, 2) \) is not robust. Indeed, no sequentially consistent execution of \( 1to1 \) allows the assert statements to load the initial value of \( y \).

Our main result is the following.

**Theorem 3.** ROB is PSPACE-complete.

The PSPACE lower bound follows from PSPACE-hardness of control state reachability in sequentially consistent programs [16]. To reduce to robustness, we add an artificial happens-before cycle starting in the target control state. The rest of the paper shows a PSPACE algorithm, and hence upper bound, for the problem.

## 4 Normal-Form Violations

We show that a PGAS program is not robust if and only if it has a violating computation of the following normal form.

**Definition 4.** Computation \( \tau := \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in C(P, N) \) is in normal form if all \( e \in \tau_2 \cdot \tau_3 \cdot \tau_4 \) satisfy \( \text{kind}(e) \in \{\text{popa, popb}\} \) and for all \( a, b \in \tau_1 \) with \( \text{kind}(a), \text{kind}(b) \notin \{\text{popa, popb}\} \) and all \( a', b' \in \tau_i \) with \( i \in \{1, 2\} \) we have:

\[ a \prec_{\tau_1} b, a \not\leftrightarrow^* b, a \leftrightarrow^* a', b \leftrightarrow^* b' \implies a' <_{\tau_i} b'. \]  

(NF)

We explain the normal-form requirement [NF]. Consider two accesses \( a \) and \( b \) to remote processes that can be found in the first part of the computation \( \tau_1 \). Assume corresponding pop events \( a' \) and \( b' \) are delayed and can both be found in a later part of the computation, say \( \tau_2 \). Then the ordering of \( a' \) and \( b' \) in \( \tau_2 \) coincides with the order of \( a \) and \( b \) in \( \tau_1 \). Computation \( R_{1to1} \) is not in normal-form whereas \( R_{1to1}^p \) in Figure 4 is. The following theorem guarantees that, in case of non-robustness, normal-form violations always exist.

**Theorem 5.** A PGAS program \((P, N)\) is robust iff it has no normal-form violation.
A Theory of Partitioned Global Address Spaces

Therefore, the resulting computation of $e$ is not robust if and only if it has a violating computation $\tau$. Phrased differently, to decide robustness our procedure should look for normal-form violations. The remainder of the section is devoted to proving Theorem 5. We make use of the following property of PGAS programs: every computation contains an event that can be deleted, in the sense that the result is again a computation of the program.

Lemma 6 (Cancellation). Consider $\epsilon \neq \tau \in C(\mathcal{P}, N)$. There is an event $e \in \tau$ so that $\tau \setminus e \in C(\mathcal{P}, N)$. Computation $\tau \setminus e$ is defined to remove $e$ and all $\leftrightarrow$-related events from $\tau$.

Proof. Take as $e$ the last event in $\tau$ with $\text{kind}(e) \notin \{\text{popa, popb}\}$. All events to the right of $e$ are unconditionally executable. Moreover, $\tau$ does not have $\rightarrow_{\text{po}}$-successors following $e$. Therefore, the resulting computation $\tau \setminus e$ is in $C(\mathcal{P}, N)$. ▶

A PGAS program is not robust if and only if it has a violating computation $\tau$ of minimal length. Let $e \in \tau$ be the event determined by Lemma 6. If $\text{kind}(e) \notin \{\text{read, write}\}$, then $\tau = \tau_1 \cdot e \cdot \tau_2$. Otherwise $\tau = \tau_1 \cdot e \cdot \tau_2 \cdot e' \cdot \tau_3 \cdot e'' \cdot \tau_4$ with $e \leftrightarrow e' \leftrightarrow e''$. Consider the latter case where $\tau \setminus e = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4$. Since $|\tau \setminus e| < |\tau|$, the new computation is not violating and $\rightarrow_{\text{hb}} (\tau \setminus e)$ is acyclic. This acyclicity guarantees we find a computation $\sigma \in E^*$ with the same happens-before relation as $\tau \setminus e$ and where pop events directly follow their remote accesses. Intuitively, $\sigma$ is a sequentially consistent computation corresponding to $\tau \setminus e$.

Lemma 7 (Reinsertion). There is $\sigma \in C(\mathcal{P}, N)$ with $\rightarrow_{\text{hb}} (\sigma) = \rightarrow_{\text{hb}} (\tau \setminus e)$ and $\sigma = \sigma_1 \cdot e_1 \cdot \ldots \cdot e_n \cdot \sigma_2$ for all $e_1 \leftrightarrow \ldots \leftrightarrow e_n$.

We now use $\sigma$ to rearrange the events in $\tau \setminus e$ and guarantee the normal-form requirement. The idea is to project $\sigma$ to the events in $\tau_1$ to $\tau_4$. Reinserting $e$ yields a normal-form violation:

$$
\tau^{nf} := (\sigma \downarrow \tau_1) \cdot e \cdot (\sigma \downarrow \tau_2) \cdot e' \cdot (\sigma \downarrow \tau_3) \cdot e'' \cdot (\sigma \downarrow \tau_4).
$$

The following lemma concludes the proof of Theorem 5.

Lemma 8 (Reinsertion). $\tau^{nf} \in C(\mathcal{P}, N)$, $\rightarrow_{\text{hb}} (\tau^{nf}) = \rightarrow_{\text{hb}} (\tau)$, and $\tau^{nf}$ is in normal form.

Example 9. Computation $\tau_{1to1}$ in Example 1 is a shortest violation. The event determined by Lemma 6 is $e = \text{load}$. Therefore, $\tau \setminus e = \tau_1 \cdot \tau_2$ with

$$
\tau_1 = \text{write} \cdot \text{write} \cdot \text{popa} \cdot \text{popa} \cdot \text{bar} \cdot \text{bar} \quad \text{and} \quad \tau_2 = \text{popb} \cdot \text{popb}.
$$

A sequentially consistent computation corresponding to $\tau \setminus e$ is

$$
\sigma = \text{write} \cdot \text{popa} \cdot \text{popb} \cdot \text{write} \cdot \text{popa} \cdot \text{popb} \cdot \text{bar} \cdot \text{bar}.
$$

The normal-form violation $\tau_{1to1}^{nf}$ is depicted in Figure 4. Note that $\tau_{1to1}^{nf}$ is indeed in $C(1to1, 2)$. Moreover, $\text{popa}$ and $\text{popb}$ immediately follow $\text{write}$ and $\text{write}$, respectively. Similarly, the $\text{popb}$ and $\text{popb}$ events in the second part of the computation respect the order of $\text{write}$ and $\text{write}$ in the first part of the computation. This means, (NF) holds.
5 From Normal-Form Violations to Language Emptiness

We now reduce checking the absence of normal-form violations to the emptiness problem in a suitable automaton model. We introduce multiheaded automata and construct, for each program \((P,N)\), a multiheaded automaton accepting all normal-form computations. To verify robustness, we check that the intersection of this automaton with regular languages accepting cyclic happens-before relations is empty.

5.1 Multiheaded Automata

Multiheaded automata are an extension of finite automata. Intuitively, instead of generating just a single computation, they generate several computations in one pass, each by a separate head. The language of the multiheaded automaton then consists of the concatenations of the computations generated by each head.

Syntactically, an \(n\)-headed finite automaton over alphabet \(\Sigma\) is a finite automaton that uses the extended alphabet \(\Gamma,n \times \Sigma\). We have \(A=(S,(\Gamma,n \times \Sigma),\Delta,s_0,F)\). The semantics, however, is different from finite automata. Given \(\sigma \in (\Gamma,n \times \Sigma)^*\), we use \(\sigma \downarrow k\) to project \(\sigma\) to the letters \((k,a)\), and afterwards cut away the index \(k\). So \(((1, a) \cdot (2, b) \cdot (1, c)) \downarrow 1 = a \cdot c\). With this, the language of \(A\) is \(L(A) := \{\text{comp}(\sigma) \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}\) where \(\text{comp}(\sigma) := \sigma \downarrow 1 \cdots \sigma \downarrow n\).

Multiheaded automata are closed under regular intersection, and emptiness is decidable in non-deterministic logarithmic space. Indeed, checking emptiness reduces to finding a path from an initial to a final node in a directed graph.

- Lemma 10. Consider an \(n\)-headed automaton \(U\) and a finite automaton \(V\) over a common alphabet \(\Sigma\). There is an \(n\)-headed automaton \(W\) with \(L(W) = L(U) \cap L(V)\).

- Lemma 11. Emptiness for \(n\)-headed automata is \(\text{NL-complete}\).

Multiheaded automata are incomparable with context-free grammars, and indeed the normal-form computations of a program may be non-context-free. Multiheaded automata can be understood as a restriction of matrix grammars. Matrix grammars, productions simultaneously rewrite multiple non-terminals. Roughly, each production can be understood as a Petri net transition, and emptiness is decidable as Petri net reachability is. Since we target a \(\text{PSPACE}\) result, matrix grammars are too expressive for our purposes.

5.2 Detecting Normal-Form Computations

We define a 4-headed automaton \(Y(P,N):= (S_Y \cup S_Y^{\text{aux}}, E, \Delta_Y, s_0, Y_S)\) that accepts all normal-form computations \(\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in C(P,N)\). In order to accept \(\tau_1\), the new automaton tracks the control and memory configurations in the way \(X(P,N)\) does. For the remainder of the computation, these configurations are not needed. Indeed, \(\tau_2\) to \(\tau_4\) only consist of \(\text{popa}\) and \(\text{popb}\) events that are executable independently of the control and memory configurations. However, \(Y(P,N)\) has to take care of the ordering of \(\text{popa}\) and \(\text{popb}\) events from the same queue. In particular, if \(e_1\) handles a request issued before the request of \(e_2\) with \(\text{kind}(e_1) = \text{kind}(e_2)\), then it cannot be the case that \(e_1 \in \tau_j\) and \(e_2 \in \tau_i\) with \(i < j\).

\(^1\) Consider \(P := (\{q_0\}, \text{CMD}, \{q_0 \xrightarrow{\text{read}(0,0,0,0)} q_0\}, \{q_0\})\) running on a single node. The language \(C(P,1)\) is not context-free. To see this, let \(\text{kind}(a) = \text{read}, \text{kind}(b) = \text{popa}, \text{and kind}(c) = \text{popb}.\) Then \(C(P,1) \cap a^*b^*c^*\) is the non-context-free language \(\{a^pb^pc^p \mid p \geq 0\}\).
A Theory of Partitioned Global Address Spaces

\begin{align*}
\text{(gpa')} & \quad \text{pa}(r, q) < \text{pb}(r, q) \\
\quad s \xrightarrow{s'} \text{pa} := \text{pa}(r, q) := \text{pa}(r, q) + 1 & \qquad \text{pb}(r, q) < 4 \\
\quad s \xrightarrow{s'} \text{pb} := \text{pb}(r, q) := \text{pb}(r, q) + 1 & \qquad \text{(gpb')} \end{align*}

\begin{tabular}{l}
\text{cmd} = \text{write}(c_{\text{loc}}^{\text{old}}, c_{\text{rem}}^{\text{new}}, c_{\text{rem}}^{\text{new}}, e_{\text{q}}) \\
\quad \text{pa}(r, e_{\text{q}}) = m & \quad \text{pb}(r, e_{\text{q}}) = n
\end{tabular}

\[ s \xrightarrow{1, \text{cmd}} s_{\text{aux}1} \xrightarrow{m, (\text{popa}(r, e_{\text{q}}^{\text{old}}))} s_{\text{aux}2} \xrightarrow{n, (\text{popb}(r, e_{\text{q}}^{\text{old}}))} s' \]

\[ \text{if } n = 1 \text{ then } m' := m[(c_{\text{rem}}^{\text{new}}, c_{\text{rem}}^{\text{new}}) := m((r, e_{\text{q}}^{\text{old}}))] \]

\[ \text{Table 2 Transition rules for } \mathcal{Y}(P, N), \text{ given } q_1 \xrightarrow{\text{cmd}} q_2 \text{ and current state } s = (s, m, pa, pb) \text{ with } \text{st}(r) = q_1. \text{ The target is } s' = (s', m', pa', pb'). \]

\[ \text{The multiheaded automaton accepts all normal form computations by } (\mathcal{NF}). \text{ Computations that are not in normal form, e.g. } \tau_{\text{to1}}, \text{ cannot be generated by } \mathcal{Y}(P, N). \]

\section{5.3 Detecting Violations}

The multiheaded automaton accepts all normal form computations, and we would like to check if one of those computations is violating. In general, violating computations can contain complicated cycles in the happens-before relation. However, we now show that whenever a computation has a happens-before cycle, it has a cycle in which each process is entered and left at most once. Our algorithm for robustness will look for happens-before cycles of this special form that, as we will show, can be captured by a regular language.

\[ a_1 \leftrightarrow a_2 \leftrightarrow a_3 \leftrightarrow \ldots \leftrightarrow \text{CYC} \]

where \( \text{rank}(x_i) = \text{rank}(y_j) \) \text{ iff } i = j, \text{ for all } x_i, y_j \in \{a_1, \ldots, a_k\}, \text{ and } \sim := \rightarrow_{cf} \cup \leftrightarrow. \]
Example 14. The computations $\tau_{1to1}$ (Example 1) and $\tau^{nf}_{1to1}$ (Example 9) have a cycle of the form (CYC) depicted in Figure 3. $n = 2$, $a_1 = b_1 = \text{bar}$, $c_1 = d_1 = \text{load}$, $a_2 = \text{popb}$, $b_2 = \text{write}$, $c_2 = d_2 = \text{bar}$.

Note that $d_i \leftrightarrow a_{i+1}$ means both are barriers, kind($d_i$) = bar = kind($a_{i+1}$). This holds as the ranks are different. In spite of the additional restrictions, cycles (CYC) are not trivial to recognize. The reason is that the events constituting the cycle are not necessarily contained in the computation in the order in which they appear in the cycle, see Figure 3. The idea of our cycle detection is to first guess the events $a_i$ and $d_i$ for each process and then check that $d_i \rightarrow_{cf} a_{i+1}$ holds. The former can be accomplished by an extension $Y^M(P, N)$ of the multiheaded automaton $Y(P, N)$, the latter by a regular intersection.

The automaton $Y^M(P, N)$ accepts computations over the alphabet $E \times M$ with $M := 2^{\{\text{enter}, \text{leave}\}}$. The events marked by enter are the guessed $a_i$ events in (CYC) and those marked by leave are the $d_i$ events in (CYC). We still have to guarantee we only mark $a_i$ and $d_i$ that satisfy $a_i \leftrightarrow^* b_i \rightarrow_{po} c_i \leftrightarrow^* d_i$. This is straightforward thanks to the fact that $Y(P, N)$ generates the events of each process in program order, and generates events related by $\leftrightarrow$ in one shot. The full construction of $Y^M(P, N)$ is given in the appendix.

Example 15. Consider the normal-form computation $\tau^{nf}_{1to1}$ (Example 9) that has the cycle (CYC) given in Figure 3. A corresponding marked computation of $Y^M(P, N)$ is

\[
\begin{align*}
\text{(write, 0)} & \cdot (\text{popa, 0}) \cdot \text{(write, 0)} \cdot (\text{popa, 0}) \cdot \\
\text{(bar, \{enter\})} & \cdot (\text{bar, \{leave\}}) \cdot (\text{load, \{leave\}}) \cdot (\text{popb, 0}) \cdot (\text{popb, \{enter\}}).
\end{align*}
\]

Every cycle of the form (CYC) has a cycle type cyc, which is a sequence $\text{cyc} = r_1 \ldots r_k$ of ranks from $1, N$ with $r_i \neq r_j$ for $i \neq j$. The idea is that the events $a_i, b_i, c_i, d_i$ belong to rank $r_i$. For each pair $r_i, r_{i+1}$ in this sequence, we construct a finite automaton $Z^{r_i,r_{i+1}}$ over the alphabet $E \times M$. It checks whether there is a conflict or identity 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 $\text{rank}(e_i) = r_i$ marked by leave in $m_i$. It remembers the kind and the address of this event. Then, it seeks a marked event $(e_{i+1}, m_{i+1})$ with $\text{rank}(e_{i+1}) = r_{i+1}$ marked by 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. Since finite automata are closed under intersection, we can define the finite automaton of cycle type cyc as $Z^{\text{cyc}} := Z^{r_1, r_2} \cap \ldots \cap Z^{r_{k-1}, r_k} \cap Z^{r_k, r_1}$.

Theorem 16. $P$ is robust iff $\mathcal{L}(Y^M(P, N)) \cap \mathcal{L}(Z^{\text{cyc}}) = \emptyset$ for all cycle types cyc.

We can now prove Theorem 3. To check whether $(P, N)$ is robust, we go over all cycle types cyc $= r_1 \ldots r_k$. This enumeration of cycle types can be done in space that is polynomial in $N$. For each such sequence, we check if $\mathcal{L}(Y^M(P, N)) \cap \mathcal{L}(Z^{\text{cyc}}) = \emptyset$. By Theorem 16, the program is robust iff all intersections are empty. By Lemma 10, there is a 4-headed finite state automaton $W$ with $\mathcal{L}(W) = \mathcal{L}(Y^M(P, N)) \cap \mathcal{L}(Z^{\text{cyc}})$. Since the size of $W$ is exponential in the size of $(P, N)$ and emptiness is in NL by Lemma 11, deciding $\mathcal{L}(W) = \emptyset$ can be done in space that is polynomial in $(P, N)$. This shows robustness is in PSPACE.

References

1. 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.
2. J. Alglave. A Shared Memory Poetics. PhD thesis, University Paris 7, 2010.
A Theory of Partitioned Global Address Spaces

3. J. Alglave and L. Maranget. Stability in weak memory models. In CAV, volume 6806 of LNCS, pages 50–66. Springer, 2011.

4. D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, 2002.

5. A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In ESOP, LNCS, pages 553–553. Springer, 2013.

6. A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against Total Store Ordering. InICALP, volume 6756 of LNCS, pages 428–440. Springer, 2011.

7. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In CAV, volume 5123 of LNCS, pages 107–120. Springer, 2008.

8. J. Burnim, C. Stergiou, and K. Sen. Sound and complete monitoring of sequential consistency for relaxed memory models. In TACAS, volume 6605 of LNCS, pages 11–25. Springer, 2011.

9. 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.

10. UPC Consortium. UPC language specification v1.2. Technical report, 2005.

11. J. Dassow and G. Păun. Regulated Rewriting in Formal Language Theory, volume 18 of Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1999.

12. 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.

13. J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. www.mcs.anl.gov/ uploads/cels/papers/P4014-0113.pdf.

14. Global address space programming interface. http://www.gaspi.de/

15. 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.

16. D. Kozen. Lower bounds for natural proof systems. InFOCS, pages 254–266. IEEE, 1977.

17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, 28(9):690–691, 1979.

18. 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.

19. A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. InMICRO, pages 363–375. IEEE, 2012.

20. J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. InParallel and Distributed Processing, volume 1586 of LNCS, pages 533–546. Springer, 1999.

21. The UPC NAS parallel benchmarks. http://upc.gwu.edu/download.html.

22. R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. InACM Sigplan Fortran Forum, volume 17, pages 1–31. ACM, 1998.

23. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. InECOOP, volume 6183 of LNCS, pages 478–503. Springer, 2010.

24. C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. InSC’11, page 51. ACM, 2011.

25. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. ACM TOPLAS, 10(2):282–312, 1988.
For some of the following proofs, we assume that Table 3 and Table 4 associate with each event $e$ in memory configuration $m$.

**Table 3** Transition rules for $X(P, N)$, given $q_1 \xrightarrow{cmd} q_2$ and current state $s = (st, m, fa, fb)$ with $st(r) = q_1$. We set $st' := st[r := q_2]$ to update $st$ so that process $r$ is at $q_2$. $\hat{e}$ denotes the evaluation of expression $e$ in memory configuration $m$.

**Table 4** Transition rules for $Y(P, N)$, given $q_1 \xrightarrow{cmd} q_2$ and current state $s = (st, m, pa, pb)$ with $st(r) = q_1$. The target is $st' = (st', m', pa', pb')$ where, unless otherwise stated, $st' = st$, $m' = m$, $pa' = pa$, $pb' = pb$. The auxiliary states $saux_1, saux_2 \in SY$ are unique for each rule application.

## Missing Proofs

For some of the following proofs, we assume that Table 3 and Table 4 associate with each event $e$ the transition in the program that produced this event: $\text{inst}(e)$. Also, for a read, write, popa, or popb event we write $\text{que}(e)$ to denote the id of the queue being modified by this event.

**Proof of Lemma 8.** To relieve the reader from the burden of syntax, we consider the case when $\tau \backslash e = \tau_1 \backslash 2$. We start with the program order. Let $e_1, e_2 \in \tau_1$ with $e_1 \rightarrow_{po} e_2$ in $\tau$ and, consequently, in $\tau \backslash e$. By definition of $\sigma$, we have $e_1 \rightarrow_{po} e_2$ in $\sigma$. Since $\sigma \downarrow \tau_1$ contains $e_1$ and $e_2$ and does not add events between them, $e_1 \rightarrow_{po} e_2$ holds for $\sigma \downarrow \tau_1$ and, consequently, $\tau_{nf}$. Assume $e_1 \in \tau_1$ and $e_2 \in \tau_2$ with $e_1 \rightarrow_{po} e_2$ in $\tau$ and in $\tau \backslash e$. Then $e_1$ is the rightmost element in $\tau_1$ with its rank that is different from a pop. Similarly, $e_2$ is the leftmost element
in $\tau_2$ 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 $e_1 \rightarrow_{po} e_2$ in $\tau^{nf}$. The case when $e_1 \in \tau_1$ and $e_2 = e$ is similar.

Since $\tau$ and $\tau^{nf}$ 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 $e_1, e_2 \in \tau_1$ with $e_1 \rightarrow_{cf} e_2$ in $\tau$ and hence in $\tau \setminus e$. By definition of $\sigma$, we have $e_1 \rightarrow_{cf} e_2$ in $\sigma$. Since $\sigma \downarrow \tau_1$ contains $e_1$ and $e_2$ and does not add new actions between them, $e_1 \rightarrow_{cf} e_2$ holds for $\sigma \downarrow \tau_1$ and, consequently, for $\tau^{nf}$.

Assume $e_1, e_2 \in \tau_1$ and $e_1 \not\rightarrow_{cf} e_2$ in $\tau$. 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 $\tau^{nf}$. The other option is that $e_1 \rightarrow_{cf} e_3$ in $\tau$, where $e_3$ is a write to $\text{addr}(e_1) = \text{addr}(e_2)$ that is located between $e_1$ and $e_2$ in $\tau_1$. Then, as already proven, $e_1 \rightarrow_{cf} e_3$ will hold in $\tau^{nf}$. Consequently, $e_1 \rightarrow_{cf} e_2$ will not hold in $\tau^{nf}$. The case when $e_1, e_2 \in \tau_2$ is similar.

Assume $e_1 \in \tau_1$, $e_2 \in \tau_2$, and $e_1 \rightarrow_{cf} e_2$ in $\tau$. Then, $e$ is not a write to $\text{addr}(e_1) = \text{addr}(e_2)$, and $e_1 \rightarrow_{cf} e_2$ in $\tau \setminus e$. Note that $\sigma \downarrow \tau_1$ does not contain a write to $\text{addr}(e_1)$ to the right of $e_1$. Otherwise, $\tau_1$ would contain a write $e_3$ to $\text{addr}(e_1)$, and $e_1 \rightarrow_{cf} e_3$, which contradicts $e_1 \rightarrow_{cf} e_2$ in $\tau$. With a similar argument, $\sigma \downarrow \tau_2$ does not contain a write to $\text{addr}(e_1)$ to the left of $e_2$. Therefore, $e_1 \rightarrow_{cf} e_2$ in $\sigma \downarrow \tau_1 \cdot e \cdot \sigma \downarrow \tau_2$.

Assume $e_1 \in \tau_1$, $e_2 \in \tau_2$, and $e_1 \not\rightarrow_{cf} e_2$ in $\tau$. The proof of $e_1 \not\rightarrow_{cf} e_2$ in $\tau^{nf}$ is as in the case when $e_1, e_2 \in \tau_1$.

The case when $e_1 = e$ or $e_2 = e$ is no harder.

The formal definition of the identity relation takes a computation $\alpha$ and determines the three projections $\alpha \downarrow \{\text{write}, \text{read}\}$, $\alpha \downarrow \text{popa}$, and $\alpha \downarrow \text{popb}$. The identity relation then relates the $i$th elements in these projections. To show that the identity relations in $\tau$ and $\tau^{nf}$ coincide, one shows that the three projections coincide — using the same technique as for the program order. Therefore, the identity relations of both computations match. Also note that for each read or write event sequence $e_1 \leftarrow e_2 \leftrightarrow e_3$, we have $e_1 \prec_{nf} e_2 \prec_{nf} e_3$. This holds by the fact that $e_1 \prec_{\tau} e_2 \prec_{\tau} e_3$, and the fact that $\sigma = \sigma_1 \cdot \sigma_2$, and $\sigma_1 \cdot \sigma_2$ for some $\sigma_1$ and $\sigma_2$.

To prove that $\tau^{nf} \in C(P, N)$, we proceed by contradiction. Let $\alpha \neq \tau^{nf}$ be the longest prefix of $\tau^{nf}$ so that $s_0 X \overset{\alpha}{\rightarrow} s$ for some state $s$. Then $\tau^{nf} = \alpha \cdot \tilde{e}$ with $s_0 X \overset{\alpha}{\rightarrow} s$ and $s \not\overset{\tilde{e}}{\rightarrow}$. Let $s = (st, m, fa, fb)$. If $\text{kind}(\tilde{e}) \in \{\text{popa}, \text{popb}\}$, then $s \not\overset{\tilde{e}}{\rightarrow}$ means that the respective queue $fa$ or $fb$ contains an incorrect topmost element or is empty in $s$. But this contradicts to $e_1 \prec_{nf} e_2 \prec_{nf} e_3$ and equality of identity relations established above. If $\text{kind}(\tilde{e}) \notin \{\text{popa}, \text{popb}\}$, then $s \not\overset{\tilde{e}}{\rightarrow}$ may hold because the transition $q_1 \overset{\text{cmd}}{\rightarrow} q_2$ of $\tilde{e}$ requires a different source state, $q_1 \neq \text{st}(\text{rank}(\tilde{e}))$. But since $\text{st}(\text{rank}(\tilde{e}))$ is unambiguously determined by the $\text{instr}$ of $\rightarrow_{po}$-predecessor of $\tilde{e}$, which is the same in $\tau^{nf}$ and in $\tau$ due to the matching program-order relations, this is not the case. The last opportunity why $s \not\overset{\tilde{e}}{\rightarrow}$ may hold is because the transition producing $\tilde{e}$ reads different values from registers or memory, e.g. $\tilde{e}$ is an assertion $\text{assume}(c)$ and $\tilde{e} = 0$ in $s$. But since $\tau^{nf}$ consists of the same events as $\tau$, has the same program and conflict relations (i.e. reads receive values from the same writes in both computations), and $\tau \in C(P)$, this cannot be the case.

Finally, $\tau^{nf}$ is in normal-form. The condition on the shape of $\tau^{nf}$ is immediate, \[\text{NF}\] holds by the definitions of $\tau^{nf}$ and $\sigma$.

**Proof of Lemma 10.** Let $U = (S_U, \Sigma, \Delta_U, s_0U, F_U)$ and $V = (S_V, \Sigma, \Delta_V, s_0V, F_V)$. We set $W : = (S_W, \Sigma, \Delta_W, s_0W, F_W)$. Let $\Omega$ be the set of functions $\Omega : \tau \rightarrow S_V$. Then, the set of states is $S_W := \{s_0W\} \cup (S_U \times \Omega \times \Omega)$. 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:

- \[ s_0 \xrightarrow{} (s_0, \omega, \omega) \] for each \( \omega \in \Omega \) with \( \omega(1) = s_0 \).
- \[ (s_0, \omega_1, \omega_2) \xrightarrow{k,a} (s_0, \omega_1', \omega_2) \] if \( s_U \xrightarrow{k,a} s_U', \omega_1(k) \xrightarrow{\omega_1'}(k), \text{ and } \omega_1(i) = \omega_1'(i) \) for \( i \neq k \).
- \[ (s_0, \omega_1, \omega_2) \xrightarrow{} (s_0, \omega_1', \omega_2) \] if \( s_U \xrightarrow{} s_U' \).
- \[ (s_0, \omega_1, \omega_2) \xrightarrow{} (s_0, \omega_1', \omega_2) \] if \( \omega_1(k) \xrightarrow{} \omega_1'(k) \) 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 \( \alpha_k \) is produced by the \( k^{th} \) head of \( U \).

By the \( \varepsilon \)-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 \( \alpha \). The other transitions effectively execute the automaton \( U \) synchronously with \( n \) copies of the automaton \( V \), each matching its own \( \alpha_k \) subword of \( \alpha \), 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 \( k + 1^{th} \) copy, and the \( n^{th} \) copy has reached a final state in \( F_V \).

**Lemma 17.** \( Y(\mathcal{P}, N) \) only generates computations of \( (\mathcal{P}, N) : \mathcal{L}(Y(\mathcal{P}, N)) \subseteq C(\mathcal{P}, N) \).

**Proof.** Consider \( s_0 \xrightarrow{} s_V \) with \( s_V = (s, m, pa, pb) \in S_Y \).

Let \( \tau = \text{comp}(\sigma) = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \) with \( \tau_i = \varepsilon \downarrow i \). We prove the following by induction on the length of the computation.

**IS1** \( s_0 \xrightarrow{} s_X \) for some \( s_X \in F_X \). Membership in \( F_X \) means the queues of \( s_X \) are empty.

**IS2** \( s_0 \xrightarrow{\tau} (s, m, fa, fb) \) for some \( fa, fb \), but with the same \( s, m \) as in \( s_Y \) above.

**IS3** Let \( pa(r, q) = k \). Then no \( \tau_i \) with \( i > k \) contains an event \( e \) with \( \text{kind}(e) = \text{popa}, \text{rank}(e) = r \), and \( \text{que}(e) = q \). A similar statement holds for \( fb \).

**IS4** For all \( e \in \tau_2 \cdot \tau_3 \cdot \tau_4 \) we have \( \text{kind}(e) \in \{\text{popa}, \text{popb}\} \).

In the base case with \( \sigma = \varepsilon \) the inductive statement trivially holds.

Assume the statement holds for \( \sigma \). Consider \( s_0 \xrightarrow{} s' = (s', m', pa', pb') \) which extends \( \sigma \) with Rule (\text{read}):

\[ \sigma' = \sigma \cdot (1, e_1) \cdot (2, e_2) \cdot (3, e_3) \text{ kind}(e_1) = \text{read}, \text{kind}(e_2) = \text{popa}, \text{kind}(e_3) = \text{popb}. \]

Then \( m' = m, pa' = pa, pb' = pb \), and \( \tau' = \text{comp}(\sigma') = \tau_1' \cdot \tau_2' \cdot \tau_3' \cdot \tau_4' \), where \( \tau_i' = \sigma_i \downarrow i \) are \( \tau_1' = \tau_1 \cdot e_1, \tau_2' = \tau_2 \cdot e_2, \tau_3' = \tau_3 \cdot e_3, \text{ and } \tau_4' = \tau_4 \). Since **IS4** and **IS3** hold for \( \sigma \), they also hold for \( \sigma' \) by definition of \( \sigma' \) and Rule (\text{read}).

It remains to check the behaviour of the state-space automaton. By **IS2** from the induction hypothesis and the Rules (\text{read}) and (\text{read}'), we have \( s_0 \xrightarrow{\tau_1 \cdot e_i \cdot \tau_2} (s', m, fa, fb) \). So **IS2** holds for \( \sigma' \), as well. To check **IS1** for \( \sigma' \), we consider the content of \( fa' \). According to Rule (\text{read}), we have \( fa' := fa[(\text{rank}(e_1), \text{que}(e_1)) := fa(\text{rank}(e_1), \text{que}(e_1)) \cdot (\text{rem}, \text{aum}, \text{loc}, \text{aoc})] \).

By the induction hypothesis, we can generate \( \tau_2 \) from \( (s, m, fa, fb) \). In \( (s', m, fa, fb) \), we append an action to \( fa \). Since \( \tau_2 \) only consists of \( \text{popa} \) and \( \text{popb} \) events, we can still generate the computation from \( (s', m, fa, fb) \). This yields \( s_0 \xrightarrow{\tau_1 \cdot e_i \cdot \tau_2} s_1 \) for some \( s_1 \).

We now set \( s_1 \xrightarrow{} s_2 \) for some \( s_2 \). Let \( s_1 = (s'', m'', fa'', fb'') \). When checking **IS3** for \( \sigma' \), we noted that \( \tau_3 \cdot \tau_4 \) does not contain \( \text{popa} \) events \( e \) with \( \text{rank}(e) = \text{rank}(e_1) \) and \( \text{que}(e) = \text{que}(e_1) \). Therefore, by **IS1** from the induction hypothesis, all elements in \( fa(\text{rank}(e_1), \text{que}(e_1)) \) are popped by \( \text{popa} \) transitions in \( \tau_2 \). As a result, \( fa'(\text{rank}(e_1), \text{que}(e_1)) \) contains only the single element added by \( e_1 \). Comparing Rules (\text{read}), (\text{popa}), and (\text{read}'), shows \( s_1 \xrightarrow{} s_2 \). Note that we need to take the read-rules into account to make sure the contents of the tuple \( e_2 \) coincide for \( Y(\mathcal{P}, N) \) and \( X(\mathcal{P}, N) \).
The fact that $X(\mathcal{P}, N)$ can accept the rest of computation $\tau' (s_2 \xrightarrow{\tau_1 \cdot \tau_2 \cdot \tau_3} s_3$ for some $s_3)$ is proven similarly. Emptiness of the queues in $s_3$ follows from Rule (read') and IS1 for $\tau$.

The argumentation for write events, kind($e_1$) = 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 $Y(\mathcal{P}, N)$ and $X(\mathcal{P}, N)$.

Lemma 18. Automaton $Y(\mathcal{P}, N)$ generates all normal-form computations of the program: \[ \{ \tau \in C(\mathcal{P}, N) \mid \tau \text{ is in normal form} \} \subseteq \mathcal{L}(Y(\mathcal{P}, N)) \].

Proof. Consider a normal-form computation $\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in C(\mathcal{P}, N)$ with $s_0 X \xrightarrow{\tau_3} s_X$ for some $s_X = (s, m, fa, fb)$. To prove that $Y(\mathcal{P}, N)$ can generate $\tau$, we show the following by induction on the length of the computation. (Note that by [NE] we can extend normal-form computations inductively).

IS1 Suppose $s_Y \xrightarrow{\tau} s_Y = (s, m, pa, pb)$ with $st$ and $m$ from $s_X$ above.

IS2 We have $s \downarrow i = \tau_i$ for all $i \in \{1, 2\}$.

IS3 Let the last $e$ with kind($e$) = popa, rank($e$) = $r$, Que($e$) = $q$ be in $\tau_k$. Then $pa(r, q) = k$.

If there is no such event, $pa(r, q) = 1$. There is a similar requirement for popb events.

Note that computation $e$ satisfies all the constraints. Assume the constraints hold for computation $\tau$. We extend $\tau$ 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 $\tau$ adds an event to the first part of the computation, $s_X \xrightarrow{\tau_1} s_X'$. We do a case distinction based on kind($e_1$).

Consider the case kind($e_1$) = read. Let $e_1 \leftrightarrow e_2 \leftrightarrow e_3$ with $\tau_2' = \tau_2 \cdot e_2$ and $\tau_3' = \tau_3 \cdot e_3$. Assume $e_1$ was generated by the transition $\tau_1 \xrightarrow{cmd} \tau_2$. This means $st(rank(e_1)) = q_1$. By IS1 in the induction hypothesis, $s_X$ and $s_Y$ share the same $st$ and $m$. Therefore, by Rules (read) and (read'), $Y(\mathcal{P}, N)$ can mimic the read in $X(\mathcal{P}, N)$. To make sure we append $e_2$ to $\tau_2$, we have to check the requirements on $pa$. If $pa(rank(e_2)) < \tau_2$, we can use Rule (gpa') to adapt the counter. If we assume that $pa(rank(e_2), Que(e_2)) = k > 2$, we derive a contradiction as follows. By the induction hypothesis, there is an event $e' \in \tau_k$ with rank($e'$) = rank($e_2$), Que($e'$) = Que($e_2$), and kind($e'$) = kind($e_2$) = popa. This event has a corresponding event $e \leftrightarrow e'$ in $\tau_k$. Summing up, $e, e_1, e_2, e'$ are contained in $\tau$ 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. We conclude

\[ s_Y \xrightarrow{(1.e_1)(2.e_2)(3.e_3)} s_Y' \].

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 $st$ and $m$ coincide in $s_X$ and $s_Y$.

Proof of Lemma 12. The inclusion from left to right is Lemma [18] The inclusion from right to left holds by Lemma [17] and the observation that $Y(\mathcal{P}, N)$ only generates computations in normal form.

The following lemma states that $Y(\mathcal{P}, N)$ generates events in program order.

Lemma 19. Consider computation $s_Y \xrightarrow{\tau} s_Y$ with events $(1, e_1) \prec_\sigma (1, e_2)$ so that kind($e_1$), kind($e_2$) $\notin \{popa, popb\}$ and rank($e_1$) = rank($e_2$). Then $e_1 \rightarrow_{po} e_2$ in $\tau = \text{comp}(\sigma)$.

Proof. By definition of the transition relation $\Delta_Y$ and $\rightarrow_{po}$. ▫
The following lemma states that $Y(P, N)$ generates the events $\text{popa}$ and $\text{popb}$ immediately after the corresponding read or write event.

**Lemma 20.** Let $suv \xrightarrow{\sigma} suv', \tau = \text{comp}(\sigma)$, and $e_1, e_2, e_3 \in \tau$ with $\text{kind}(e_1) \in \{\text{read, write}\}$, $\text{kind}(e_2) = \text{popa}$, and $\text{kind}(e_3) = \text{popb}$. Then $e_1 \leftrightarrow e_2 \leftrightarrow e_3$ holds in $\tau$ if and only if $\sigma = \sigma_1 \cdot (1, e_1) \cdot (m, e_2) \cdot (n, e_3) \cdot \sigma_2$ for some $\sigma_1, \sigma_2$ and $m, n \in \{0, 1\}$ with $m \leq n$.

**Proof.** By Rules (read) and (write), the preconditions on (gpa) and (gpb), and the definition of $\leftrightarrow$.

**Proof of Lemma 13.** Consider an arbitrary cycle. It has the following form:

$$a_1 \leftrightarrow^* b_1 \rightarrow_{po}^* c_1 \leftrightarrow^* d_1 \rightarrow \ldots \rightarrow a_n \leftrightarrow^* b_n \rightarrow_{po}^* c_n \leftrightarrow^* d_n \rightarrow a_1.$$ 

Assume now $\text{rank}(a_1) = \ldots = \text{rank}(d_i) = \text{rank}(a_j) = \ldots = \text{rank}(d_j)$ for some $i < j$. Fix these $i$ and $j$. Then either $b_i \rightarrow_{po}^* c_j$ or $b_j \rightarrow_{po}^* c_i$. In the former case, $\tau$ has the following happens-before cycle:

$$a_1 \leftrightarrow^* b_1 \rightarrow_{po}^* c_1 \leftrightarrow^* d_1 \rightarrow \ldots \rightarrow a_i \leftrightarrow^* b_i \rightarrow_{po}^* c_j \leftrightarrow^* d_j \rightarrow \ldots \rightarrow a_n \leftrightarrow^* b_n \rightarrow_{po}^* c_n \leftrightarrow^* d_n \rightarrow a_1.$$ 

In the latter case, $\tau$ has the following cycle:

$$a_j \leftrightarrow^* b_j \rightarrow_{po}^* c_i \leftrightarrow^* d_i \rightarrow \ldots \rightarrow a_{j-1} \leftrightarrow^* b_{j-1} \rightarrow_{po}^* c_{j-1} \leftrightarrow^* d_{j-1} \rightarrow a_j.$$ 

Repeating the procedure for the new cycle until there is no $i \neq j$ with $\text{rank}(a_i) = \ldots = \text{rank}(d_i) = \text{rank}(a_j) = \ldots = \text{rank}(d_j)$, we get a cycle of the desired form.

Now we formally define the automaton $Y^M(P, N)$, which is an extension of $Y(P, N)$ that non-deterministically guesses and marks the first and the last event in each process that contribute to a cycle — if any. We set $Y^M(P, N) := (S^M, E \times M, \Delta_{YM}, s_{0YM}, F_{YM})$, where events are optionally marked by enter and/or leave from $M := 2\{\text{enter}, \text{leave}\}$. The events marked by enter are the $a_i$ events in (CYC) and those marked by leave are the $d_i$ events in (CYC). The set of states $S^M$ consists of the states $S_Y$ extended by information about which marked events have been issued for each process: $S^M := S_Y \times \{\bot, \text{enter}, \text{leave}\}^{RNK}$. The initial state is $s_{0YM} := (s_{0Y}, 0)$ with $\mu_0(r) := \bot$ for each rank. The transition relation $\Delta_{YM}$ is defined as follows:

**M1** $(s, \mu) \xrightarrow{i, (e, 0)} (s', \mu)$ if $s \xrightarrow{i, e} s'$.

**M2** $(s, \mu) \xrightarrow{i, (e, \text{enter})} (s', \mu)$ if $s \xrightarrow{i, e} s'$.

**M3** $(s, \mu) \xrightarrow{i, (e, \text{enter})} (s', \mu[\text{rank}(e) := \text{enter}])$ if $s \xrightarrow{i, e} s'$, $\text{addr}(e) \neq \bot$ or $\text{kind}(e) = \text{bar}$, and $\mu[\text{rank}(e)] := \bot$.

**M4** $(s, \mu) \xrightarrow{i, (e, \text{enter})} (s', \mu[\text{rank}(e) := \text{leave}])$ if $s \xrightarrow{i, e} s'$, $\text{addr}(e) \neq \bot$ or $\text{kind}(e) = \text{bar}$, and $\mu[\text{rank}(e)] := \bot$.

**M5** $(s, \mu) \xrightarrow{i, (e, \text{enter})} (s', \mu[\text{rank}(e) := \text{leave}])$ if $s \xrightarrow{i, e} s'$, $\text{addr}(e) \neq \bot$ or $\text{kind}(e) = \text{bar}$, and $\mu[\text{rank}(e)] := \text{enter}$.

**M6** $(s, \mu) \xrightarrow{i, (e, \text{leave})} (s', \mu)$ if $s \xrightarrow{i, e} s'$, $\text{kind}(e) = \text{popa}$, $\text{kind}(e_2) = \text{popb}$, and $\mu[\text{rank}(e_1)] := \bot$.

The set of final states is $F_{YM} := \{(s, \mu) \mid s \in F_Y$ and $\mu(r) \in \{\bot, \text{leave}\}$ for all $r \in RNK\}$. 


Lemma 21. The languages of $Y(P,N)$ and $Y^M(P,N)$ match up to the markings: $\mathcal{L}(Y(P,N)) = \mathcal{L}(Y^M(P,N)) \downarrow E$.

Proof. The inclusion $\mathcal{L}(Y(P,N)) \subseteq \mathcal{L}(Y^M(P,N)) \downarrow E$ holds due to the Rules M1 and M2 in the definition of $Y^M$. The reverse inclusion $\mathcal{L}(Y(P,N)) \supseteq \mathcal{L}(Y^M(P,N)) \downarrow E$ follows from the fact that $(s,\mu) \rightarrow e_{i,m} (s',\mu')$ requires $s \rightarrow s'$ (M2-M6).

Lemma 22. Consider a marked computation $\tau \in \mathcal{L}(Y^M(P,N))$ and events $(a,m_1)$ and $(d,m_4)$ in $\tau$ with $\text{rank}(a) = \text{rank}(d)$ and $\text{enter} \in m_1$, $\text{leave} \in m_4$. Then $a \leftrightarrow b \rightarrow^* c \leftrightarrow^* d$ for some $(b,m_2)$ and $(c,m_3)$ in $\tau$.

Proof. Consider $s_{Y^M} \xrightarrow{\sigma} s_{Y^M}$ and let $\tau = \text{comp}(\sigma)$. Let $(a,m_1)$ and $(d,m_4)$ be two events in $\tau$ with $\text{rank}(a) = \text{rank}(d)$ and $\text{enter} \in m_1$, $\text{leave} \in m_4$. Then, $\sigma$ contains $(i,a,m_1)$ and $(j,d,m_4)$ for some $i,j \in \overline{1,n}$.

If $(i,a,m_1) >_\sigma (j,d,m_2)$, then $a$ and $d$ were generated by the two transitions defined by Rule M6. This means $\sigma = \sigma_1 \cdot (1,b,\emptyset) \cdot (j,d,\{\text{leave}\}) \cdot (i,a,\{\text{enter}\}) \cdot \sigma_2$, where $\text{kind}(b) \in \{\text{read,write}\}$, $\text{kind}(d) = \text{popa}$, and $\text{kind}(a) = \text{popb}$. Therefore, $b \leftrightarrow d \leftrightarrow a$, which can be reformulated as $a \leftrightarrow b \rightarrow^*_b \leftrightarrow^* d$.

If $(i,a,m_1) = (j,d,m_4)$, then $m_1 = m_4 = \{\text{enter,leave}\}$ and $a = d$ is the event generated by M4. Clearly, $a \leftrightarrow^* a \rightarrow^*_d d \leftrightarrow^* d$.

If $(i,a,m_1) <_\sigma (j,d,m_4)$, then $a$ was generated by M5, and $d$ was generated by Rule M5. Let $b \leftrightarrow^* a$ with $\text{kind}(b) \notin \{\text{popa, popb}\}$, and similarly $c \leftrightarrow^* d$. The fact that $b \rightarrow^*_c d$ follows from Lemma 20 and Lemma 19. Altogether, $a \leftrightarrow^* b \rightarrow^*_c d \leftrightarrow^* d$.

For the next lemma, consider a normal-form computation $\tau \in C(P,N)$ and let $\{r_1, \ldots, r_n\} \subseteq \text{RANK}$ be a set of ranks. Moreover, assume that for each rank $r_i$ with $i \in \overline{1,n}$, there are $a_i, b_i, c_i, d_i \in \tau$ that have this rank, satisfy $a_i \leftrightarrow^* b_i \rightarrow^*_d c_i \leftrightarrow^* d_i$, and where

$$\text{(addr}(a_i) \neq \bot \text{ or kind}(a_i) = \text{bar}) \text{ and } \text{(addr}(d_i) \neq \bot \text{ or kind}(d_i) = \text{bar})$$

Lemma 23. Under these assumptions, there is a marked computation $\tau' \in \mathcal{L}(Y^M(P,N))$ with $\tau' \downarrow E = \tau$ that contains, for each $i \in \overline{1,n}$, a marked event $(a_i,m_1)$ with $\text{enter} \in m_1$ and $(d_i,m_4)$ with $\text{leave} \in m_4$. All other marked events $(e,m) \in \tau$ have $m = \emptyset$.

Proof. We prove the statement of the lemma by induction on the size $n$ of the set of ranks.

The base case $n = 0$ is due to Lemma 18 and the Rules M1 and M2: $Y^M(P,N)$ can generate a marked computation $\tau_0$ with $\tau_0 \downarrow E = \tau$ and all markings being $\emptyset$. Formally, there is $s_{Y^M} \xrightarrow{\sigma_0} s_{Y^M}$ for some $s_{Y^M} \in F_{Y^M}$ with $\tau_0 = \text{comp}(\sigma_0)$.

In the induction step, assume the claim holds for sets of ranks of size $n-1$ and consider $\{r_1, \ldots, r_n\} \subseteq \text{RANK}$. By the hypothesis, there is $s_{Y^M} \xrightarrow{\sigma_{n-1}} \tau_{n-1}$ for some $s_{Y^M} \in F_{Y^M}$. Moreover, for each $i \in \overline{1,n}$ it holds that $\tau_{n-1} = \text{comp}(\sigma_{n-1})$ contains a marked event $(a_i,m_1)$ with $\text{enter} \in m_1$ and a marked event $(d_i,m_4)$ with $\text{leave} \in m_4$. All other events in $\tau_{n-1}$ have empty markings. To prove the statement for $n$, consider the possible mutual positions of $(i,a_i,\emptyset)$ and $(j,d_i,\emptyset)$ in $\sigma_{n-1}$.

If $(i,a_i,\emptyset)$ and $(j,d_i,\emptyset)$ are the same event, we have $\sigma_{n-1} = \sigma' \cdot (i,a_i,\emptyset) \cdot \sigma''$ and $(i,a_i,\emptyset)$ was generated by Rule M2. This transition can be replaced by M4 and yields $\sigma_n = \sigma' \cdot (i,a_i,\{\text{enter,leave}\}) \cdot \sigma''$.

If $(i,a_i,\emptyset) <_{\sigma_{n-1}} (j,d_i,\emptyset)$, then $\sigma_{n-1} = \sigma' \cdot (i,a_i,\emptyset) \cdot \sigma'' \cdot (j,d_i,\emptyset) \cdot \sigma'''$, where $(i,a_i,\emptyset)$ and $(j,d_i,\emptyset)$ were generated by M2. These transitions can be replaced by M3 and M5 transitions, resulting in $\sigma_n = \sigma' \cdot (i,a_i,\{\text{enter}\}) \cdot \sigma'' \cdot (j,d_i,\{\text{leave}\}) \cdot \sigma'''$. 
Consider \((i, a_n, \emptyset) \succ_{\sigma_{n-1}} (j, d_n, \emptyset)\). With Lemma \ref{lemma:19} and Lemma \ref{lemma:20} we get \(d_n \leftrightarrow a_n\).

Since barriers are not related by identity, we derive \(\text{addr}(d_n) \neq \perp \neq \text{addr}(a_n)\). This gives \(\text{kind}(a_n) = \text{popb} \land \text{kind}(d_n) = \text{popa}\). With Lemma \ref{lemma:20} \(\sigma_{n-1} = \sigma' \cdot (j, d_n, \emptyset) \cdot (i, a_n, \emptyset) \cdot \sigma''\).

The events were generated by \(\text{M2}\) transitions. These transitions can be replaced by \(\text{M6}\), which yields \(\sigma_n = \sigma' \cdot (j, d_n, \{\text{leave}\}) \cdot (i, a_n, \{\text{enter}\}) \cdot \sigma''\).

Since \(\sigma_n\) is obtained from \(\sigma_{n-1}\) by replacing one or two marked events of rank \(r_n\), and generation of the other events does not rely on \(\mu(r_n)\) (all other events of rank \(r_n\) are not marked), we have \(s_{\gamma Y \mu} \xrightarrow{\sigma} s_{\gamma Y \mu}\) for some \(s_{\gamma Y \mu} \in F_{Y \mu}\). ▶

Now we formally define the automaton \(Z_{\tau \mu}^\tau\) that checks whether there is a conflict edge from the leave-marked event of process \(r_1\) to the enter-marked event of process \(r_2\). We define \(Z_{\tau \mu}^\tau := (S_Z, E \times M, \Delta_Z, s_{\mu}, F_Z)\). The set of states \(S_Z := \{\text{init}, \text{accept}\} \cup (K \times \text{RNK} \times \text{ADR})\).

The initial state is \(s_{\mu} := \text{init}\). The set of final states is \(F_Z := \{\text{accept}\}\). The transition relation \(\Delta_Z\) is defined as follows:

- \(\text{HB1} \quad \text{init} \xrightarrow{e_m} \text{init with rank}(e) \neq r_1 \lor \text{leave} \notin m\).
- \(\text{HB2} \quad \text{init} \xrightarrow{e_m} \text{kind}(e), \text{addr}(e) \text{ for kind}(e) \neq \text{bar} \text{ if rank}(e) = r \text{ and leave} \in m\).
- \(\text{HB3} \quad (k, r, a) \xrightarrow{e_m} (k, r, a) \text{ for } k \neq \text{bar} \text{ if addr}(e) \neq (r, a) \lor \text{kind}(e) \notin \{\text{store}, \text{popb}\}\).
- \(\text{HB4} \quad (k, r, a) \xrightarrow{e_m} \text{accept} \text{ for } k \neq \text{bar} \text{ if addr}(e) = (r, a), \text{rank}(e) = r_2, \text{enter} \in m, \text{ and } \{\text{kind}(e)\} \cap \{\text{store}, \text{popb}\} = \emptyset\).
- \(\text{HB5} \quad \text{accept} \xrightarrow{e_m} \text{accept for all } (e, m) \in E \times M\).
- \(\text{HB6} \quad \text{init} \xrightarrow{e_m} \text{barrier with rank}(e) = \text{bar}, \text{rank}(e) = r_1 \text{ and leave} \in m \text{ or } \text{rank}(e) = r_2 \text{ and } \text{enter} \in m\).
- \(\text{HB7} \quad \text{barrier} \xrightarrow{e_m} \text{barrier with rank}(e) = \text{bar}, \text{rank}(e) \notin \{r_1, r_2\}\).
- \(\text{HB8} \quad \text{barrier} \xrightarrow{e_m} \text{accept with rank}(e) = \text{bar}, \text{rank}(e) = r_1 \text{ and leave} \in m \text{ or } \text{rank}(e) = r_2 \text{ and } \text{enter} \in m\).

▶ Lemma 24. Consider \(r_1, r_2 \in \text{RNK}\) and \(\tau \in \mathcal{L}(Y^M(P, N))\) that has a single marked event \((e_i, m_i)\) with \(\text{leave} \in m_i\) and \(\text{rank}(e_i) = r_1\) and a single \((e_j, m_j)\) with \(\text{enter} \in m_j\) and \(\text{rank}(e_j) = r_2\). Then \(\tau \in \mathcal{L}(Z_{\tau \mu}^\tau)\) iff \(e_i \xrightarrow{\tau} e_j\).

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 \(\text{accept} \text{ the first time, the automaton must have reached a state } (k, r, a) \text{ and performed a transition defined by } \text{HB4}\). 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 \(\tau\) with \(\text{rank}(e_j) = r_2\) and \(\text{enter} \in m_j\). The state \((k, r, a)\) was reached the first time via a transition defined by \(\text{HB2}\). 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 \(\tau\) with \(\text{rank}(e_i) = r_1\) and \(\text{leave} \in m_i\). According to \(\text{HB2}\), \(k = \text{kind}(e)\) and \((r, a) = \text{addr}(e)\). Therefore, \(\text{HB4}\) requires that \(e_i\) and \(e_j\) access the same address and at least one of them is a write. Moreover, according to Rule \(\text{HB3}\), the automaton could not consume a marked event which is a write to \((r, a) \text{ after reading } (e_i, m_i)\) and before reading \((e_j, m_j)\). Altogether, by definition of the conflict relation, \(e_i \xrightarrow{\tau} e_j\).

For the proof from right to left, let \(\tau = \tau_1 \cdot (e_i, m_i) \cdot \tau_2 \cdot (e_j, m_j) \cdot \tau_3\). The first part, \(\tau_1\), is read by the transitions defined by \(\text{HB1}\). Indeed, \((e_i, m_i)\) is the only marked event in \(\tau\) that does not satisfy the requirements of this rule. Then the automaton performs the transition defined by \(\text{HB2}\), reads \((e_i, m_i)\), and reaches the state \((k, r, a)\) with \(k = \text{kind}(e)\) and \((r, a) = \text{addr}(e)\). Since \(e_i \xrightarrow{\tau} e_j\), part \(\tau_2\) does not contain writes to \(\text{addr}(e_i)\). It is consumed by the transitions defined by \(\text{HB3}\). Finally, the automaton performs the transition defined by \(\text{HB4}\) and reaches the accepting state. There it loops on the symbols from \(\tau_3\). ▶
Lemma 25. Consider a cycle type \( \text{cyc} \) and let \( \tau \in (\mathcal{L}(Y^M(P,N)) \cap \mathcal{L}(Z^{cyc})) \downarrow E \). Then \( \tau \) is a computation of \((P,N)\) and has a cycle \((\text{CYC})\) of type \( \text{cyc} \).

Proof. By Lemma 17 and Lemma 21, \( \tau \) is a computation of program \((P,N)\). By Lemma 22 and Lemma 24, \( \tau \) has a dependence chain \((\text{CYC})\) of type \( \text{cyc} \).

Lemma 26. Consider a cycle type \( \text{cyc} \) and let \( \tau \) be a normal-form computation of \((P,N)\) that has a cycle \((\text{CYC})\) of this type. Then \( \tau \in (\mathcal{L}(Y^M(P,N)) \cap \mathcal{L}(Z^{cyc})) \downarrow E \).

Proof. By Lemma 23, \( Y^M(P,N) \) can generate \( \tau' \) with \( \tau' \downarrow E = \tau \), the events \( a_i, d_i \) from \((\text{CYC})\) marked by enter and leave respectively, and the other events marked by \( \emptyset \). By Lemma 24, the automata \( Z^{i,i+1} \) will accept \( \tau' \), due to \( d_i \leadsto a_{i+1} \).

Proof of Theorem 16. The statement follows from Theorem 5, Lemma 25, Lemma 13, and Lemma 26.