Automata-Theoretic Control for Total Store Ordering Architectures

Florian Furbach

Kaiserslautern University, Department of Computer Science,
D 67653 Kaiserslautern,
Germany

Master's Thesis

Table of Contents

Abstract. Diese Arbeit befasst sich mit der Möglichkeit, konsistente Berechnungen in einem Total-Store-Ordering-System mithilfe eines Controllers für den Schreibbuffer zu erzwingen, anstatt robuste Programme mit Fence-Befehlen zu erstellen. Insbesondere wird untersucht, ob ein endlicher Automat als Controller verwendet werden kann und welche Komplexität er dabei haben muss.
Desweiteren wird eine verteilte Architektur betrachtet, in der mehrere Komponenten eines Controllers mittels Broadcasts kommunizieren und die Anzahl der Nachrichten minimiert werden muss. Der Cycle-Algorithm wird vorgestellt, ein verteilter Algorithmus, der eine minimale Anzahl an Nachrichten sended und eine modifizierte Version, die einen deterministischen endlichen Automaten mit minimaler Anzahl an Zuständen darstellt.

This work deals with the possiblity to guarantee consistent computations in a total store ordering system with a controller for the write buffer rather than by creating consistent programs using fence-commands. We explore whether a finite automaton can be used as a controller and its complexity.
We further examine a distributed architecture, where multiple elements of a controller communicate using broadcasts and the number of messages needs to be minimized. We introduce the Cycle-Algorithm, a distributed algorithm that minimizes the number of communications between its components. A modified version, that can be represented by a deterministic finite automaton with a minimal number of states, will be presented as well.

Ich erkläre hiermit, die vorliegende Masterarbeit selbstständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 3.12.2012

1 Introduction

In this thesis, we consider a system architecture that consists of a number of parallel running program components where each sends a sequence of read and write actions to a shared memory. Memory access is very slow compared to operations of the processor and thus, in modern processor architectures, the system does not always suspend the computation during a memory access. Instead, memory actions are placed into buffers and the computation is continued. Various relaxed memory models are implemented in order to increase efficiency [PD95,DPN93,AG96,SHW11].

In those models, some write or read actions on the memory can be delayed from the execution by the program. From the memory point-of-view, these delays are interpreted as reorderings of the write and read actions.

However, when designing a program, we work under the assumption of sequential consistency [BM08]. That means that the actions executed by a program component arrive at the memory in the same order they have been fired. A relaxed memory model may introduce unwanted behaviour by allowing computations that can not be executed in a sequentially consistent system. In particular data race freeness is no longer guaranteed [AA93].

In the following we will concentrate exclusively on the total store ordering architecture (TSO) which is used in in the x86 processors Intel64 and IA-32 [Int07]. In order to ensure sequential consistency of a program, it is statically analyzed and additional delays are added in the form of so called fence actions. We research the possibility to equip the write buffer of a TSO system with a controller that performs an online analysis of a program execution and ensures its sequential consistency.

This thesis is organized as follows: First we describe a formal framework for the controller in Sections 2 and 3. A definition of a basic controller that detects inconsistencies is given in Section 4 and it is shown that a controller can be a finite automaton if some system parameters are finitely bounded. In Section 6, we improve the controller such that it detects inconsistencies immediately and can be used to actively avoid them. In Section 7, we explore possibilities to enhance the efficiency of controllers by combining them with a preprocessing step that analyzes the program before its execution. Distributed controllers consisting of multiple agents that communicate via broadcasting are introduced in Section 8. We present the Cycle-Algorithm in Section 9, an efficient algorithm which implements a distributed controller and we examine its complexity as a finite automaton. Finally, we give an overview of a controller's time complexity and we show how to minimize it in Section 10. The thesis is concluded with a recapitulation and an outlook on future work in Section 11.

Notations

Let V be the set of variables, D a domain of possible values and $P = \{p_1, \ldots, p_n\}$ a program consisting of a set of n program components. The set of all programs

is $\mathcal{P}$. We differentiate the sets of write and read actions.

$$Write = \{w\} \times V \times D \times P$$

$$Read = \{r\} \times V \times D \times P$$

A *component* p of a program P consists of a finite sequence of actions on p. A *program* is a set of components. We restrict ourselves to finite computations. A *computation* of a program consists of a sequence of the programs actions in the order in which they access the shared memory. We say a computation of a program is *feasible* if the value of every read action coincides with the value of the variable that is accessed by the read. Since the value of an action is of interest only so far as it affects the feasibility of a computation, we will abbreviate a write action either to (w, x, p) or to w_x meaning a write on x. Similarly for reads (r, x, p) and r_x. Note that in a definition $w = (w, x, v, p_i)$ of a write w, the w in the tuple is not the write itself but denotes that the specified action is a write.

If we work with a relaxed memory model, the computations are rewritten according to the model. When a program is executed, a shuffle of these sequences is observed at the shared memory. The set of possible shuffles of two sequences is formally defined by the recursion $a.\alpha \sqcup\!\sqcup b.\beta := a.(\alpha \sqcup\!\sqcup b.\beta) \cup b.(a.\alpha \sqcup\!\sqcup \beta)$.

In a *total store ordering* architecture the system contains a buffer consisting of finitely many FIFO (first in, first out) queues, where the write actions are buffered before being sent to the memory. When a read occurs, it accesses first the write buffer of its component. If the buffer contains no write on the same variable, it accesses the shared memory. Constructing an order of the actions arrival at the memory from the order of actions fired by a component (or program order), the following rewritings are possible: a read action that follows a write action on a different variable may arrive before it at the memory if the write does not leave the buffer until the read has been executed. We call this a *reorder* and say the read *overtakes* the write.

$$(w, x, *, p).(r, y, *, p) \curvearrowright_{re} (r, y, *, p).(w, x, *, p) \text{ with } x \neq y \text{ (reorder)}$$

If a write action is followed by a read on the same variable and the write is the last write on that variable in the buffer, then the read is performed directly on the action in the buffer and it never reaches the memory. We call this an *early read*.

$$(w, x, *, p).(r, x, *, p) \curvearrowright_{pf} (w, x, *, p) \text{ (early read)}$$

A *TSO computation* is obtained by performing rewrites on the components and performing a shuffle. In the following example we demonstrate, how a TSO architecture may introduce behaviour that is not possible in a sequential consistent architecture. In previous works, the program is statically analyzed. In order to avoid all inconsistent computations, fences are added.

A *fence* f is an action that does not affect a computation itself. It restricts the possible computations of a program by forcing any delayed action on the component to be executed before it is processed. In terms of rewriting, this means no actions can be reordered or processed by the early read rule past the

fence. If a program allows for inconsistent TSO computations, we can enforce sequential behaviour by adding fences, such that these computations are no longer feasible. Note that a fence may force multiple memory accesses and thus may halt the execution for quite some time. Since we will analyze computations directly, we do not include fences in our definition of actions. In the sequential consistency model (SC), no rewriting occurs.

Example 1 (Dekker) [BMM11] Dekker's algorithm is a simple mutex to force mutual exclusion. In a sequential environment, two parallel running components of a program prevent entering a critical section at the same time. They each signal their wish to enter the critical section by setting its assigned variable, respectively x and y, to 1. Then they ensure that the other component is not entering the critical section by executing a read of 0 on its assigned variable. The fact that the write is performed before the read on the shared memory is critical to ensure mutual exclusion [Dij65]. In a TSO environment, where each component operates on a different write buffer, this property is not preserved. We examine the following program P which is illustrated in Figure 1. Note

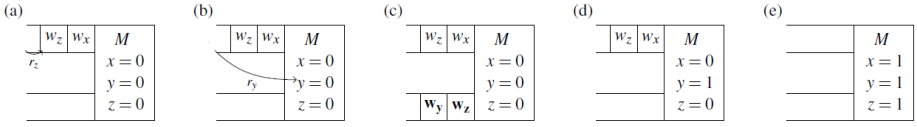

Fig. 1. [BMM11] An illustration of the inconsistent computation of Example 1 (mutex). In (a) w_x and w_z are added to the buffer of p_1 and r_z is an early read. In (b), r_y is read on the shared memory by p_1. Then $w_y.w_z$ are added to the buffer of p_2 (c). In (d), the buffer of p_2 is emptied and then r_x is read on the shared memory. The buffer of p_1 is emptied in (e). The computation is $\tau = r_y.\mathbf{w_z}.\mathbf{w_y}.\mathbf{r_x}.w_x.w_z$.

that in a 2-component system, we mark actions of the second component bold. $P = \{p_1, \mathbf{p_2}\}$,

$$p_1 = w_x := (w, x, 1, p_1).w_z := (w, z, 1, p_1).r_z := (r, z, 1, p_1).r_y := (r, y, 0, p_1);$$
$$\mathbf{p2} = \mathbf{w_z} := (w, z, 0, p_2).\mathbf{w_y} := (w, y, 1, p_2).\mathbf{r_x} := (r, x, 0, p_2)$$

The resulting TSO computation is obtained by applying the rewriting rules to the components. We apply a reorder and let r_y overtake w_x and w_z. We perform an early read on r_z. This results in

$$\sigma(p_1) = r_y.w_x.w_z$$

$$\sigma(p_2) = \mathbf{w_z}.\mathbf{w_y}.\mathbf{r_x},$$

A possible shuffle of this is the following computation:

$$\tau = r_y, \mathbf{w_z}, \mathbf{w_y}, \mathbf{r_x}, w_x, w_z$$

Note that in this computation, both components enter the critical section. The computation is not sequentially consistent and mutual exclusion fails.

However, since inconsistencies only occur when different components perform actions on the same variables within short intervals, the ratio of inconsistent computations occurring in practice is very small. This means most fence executions lead to unnecessary delays [AA93].

In this work, we explore the notion of dynamically analyzing a computation while it is executed. This will remove delays that are necessary when using static program analysis. We will introduce controllers that interface with the write buffer and process a program execution sequentially. The controller forces a delay if the computation would be infeasible without it.

Related work

There has been a lot of research exploring the static analysis of programs under different relaxed memory models. Other basic relaxed memory models in use are the TSO with read-read reordering (TSOR), where a read can also overtake another read, the partial-store ordering (PSO) where write actions can be reordered or performed on a buffer or PSO with read-read reordering (PSOR). As most modern processor architectures use complex combinations and variations of relaxed memory models, that expand the total store ordering, research on the relatively strong TSO model forms a basis to build upon in order to make statements about highly relaxed memory models [AM11].

The main problems when analyzing programs under TSO are reachability and robustness. Reachability asks which states of the system can be reached by executions of a given program. Robustness asks, whether all computations of a given program are consistent with sequentially consistent computations of the program. When using a state-based notion of consistency, the reachability problem is already non-primitive recursive and thus robustness is as well [ABBM10]. When using the stronger trace-based notion of consistency, reachability is PSPACE-complete [BMM11].

Ladan-Mozes et al. have taken a first step towards a dynamic online analysis of a computation by introducing the notion of location-based memory fences [LMLV11]. The location-based memory fence, or *l-mfence*, directly analyses the computation and adapts delays accordingly. *l-mfences* are added to a program to ensure robustness in the same way as traditional memory fences, but their effect on the computations are different. When a location-based memory fence is executed, it does not necessarily stall the computation of other processors until the buffer is emptied. Instead it guards a location in the memory and enforces a serialization only if another component attempts to read the guarded memory location. The *l-mfence* of a component p does not require the writes stored in the buffer of p to be executed before any other action can be executed. Instead it monitors other components for reads on the variable it guards and halts if it finds one. The *l-mfence* enforces that from every components perspective, an action on the guarded location is only observed after any write action leaving the buffer of p. In order to guard a specific memory location, the *l-mfence* has to be able to

effectively monitor other components for attempts to read this location. In order to achieve this, the *l-mfence* coordinates with the cache controller. It retains the exclusive state of the location for its component. When another component q attempts to read this location, it causes the buffer of p to be emptied before the p releases the state of the location to q.

This thesis builds on that idea and introduces a controller that performs an online analysis of a computation and enforces sequential consistency. This enables us to abandon fences altogether.

2 Operational Semantics of the Total Store Ordering

In the definition of computations, we have modeled the system from the perspective of the shared memory. In order to argue about a controller, we need to look at a TSO-system from the perspective of the write buffers. We now look at the actions as they enter, leave or access the buffer. We model the interactions of the buffer with the remaining system such that a computation is analyzed from the point of view of the write buffer. We first need to formalize the system with the buffer as its center.

We define the *input* sequence of a controller attached to the buffer as the write and read actions as they are executed by the program and the write actions as they are leaving the buffer. The write actions of a computation enter the buffer sequentially. Together with the read actions they form a computation in a sequential environment and represent the user perspective. The sequence of writes, in the order they leave the buffer, and the reads form the corresponding TSO computation. They represent the computation from the memory point-of-view. The controller analyzes the order in which the actions are processed in the TSO computation and checks for inconsistencies with SC computations, i.e. if a processed sequence of actions leads to a computation that can not be performed on the program without write buffers.

Every write action interacts twice with the buffer and thus occurs twice in the input of a controller. It enters the buffer and leaves it at some later time. We extend the notions of write actions. For every write action $w \in Write$, we denote the corresponding input element of the controller that signals w leaving the write buffer by w^{out}. A write w entering the buffer is denoted by w^{in}. The set of possible *buffer actions* is

$$Act := Write \times \{in, out\} \cup Read$$

The input sequence In of a controller generated by a computation is a sequence of actions on the buffer $In \in Act^*$. Note that in the input, we do not distinguish between a read action that is an early read and a read on the shared memory. This is not necessary, since the controller can keep track of the buffer-content and knows if there is a write in the buffer on the same variable as the current read. We call an an action w^{in} an *incoming write* and w^{out} an *outgoing write*.

We obtain an input sequence for a component p by replacing every write w with $w^{in}.w^{out}$ and applying the modified rewriting rules

$$(w, x, *, p)^{out}.(r, y, *, p) \curvearrowright_{re} (r, y, *, p).(w, x, *, p)^{out} \text{ with } x \neq y \text{ (reorder)}$$

$$(w, x, *, p)^{out}.(r, x, *, p) \curvearrowright_{pf} (w, x, *, p)^{out} \text{ (early read)}$$

Obviously, we can directly obtain the computation of a given input by removing all w^{in} and change all w^{out} to w. We can also reconstruct the program that generated an input by projecting the sequence onto the components, removing all outgoing writes w^{out} and changing all incoming writes w^{in} to w.

We also obtain the sequential computation that shows the actions in the order, they where fired by the program by removing the w^{out} and changing the w^{in} to w. However, when we test an input for sequential consistency, it is not sufficient to compare the TSO computation with the sequential computation obtained from the input, since the write-buffer can create additional shuffling without destroying sequential consistency.

Example 2 : We examine an input where the buffer retains consistency but changes the order in which the write actions arrive to the shared memory from the order in which they are fired by the program. It introduces additional shuffling to the computation. Given the program $\{p_1 := w_x,\ \mathbf{p_2} := \mathbf{w_x}\}$, with $w_x = (w, x, 1, p_1)$, $\mathbf{w_x} = (w, x, 0, p_2)$, there are two possible feasible consistent computations $\sigma = w_x.\mathbf{w_x}$ and $\tau = \mathbf{w_x}.w_x$ resulting in x with either 0 or 1 assigned to x. The input sequence is

$$In = w_x^{in}.\mathbf{w_x^{in}}.\mathbf{w_x^{out}}.w_x^{out}$$

This results in $x = 1$. The program sends a computation σ, that would have resulted in $x = 0$, but the buffer holds w_x until $\mathbf{w_x}$ is executed, which changes their order and generates τ.

The definition of buffer input sequences using rewritings are useful for constructing such sequences and to understand their similarity to computations, which are constructed by similar rewritings. However, in order to argue about a controller, that reads an input sequentially and performs an online analysis of the computation, we will introduce an equivalent definition of inputs.

For A being a transition system or finite automaton, $L(A)$ is the language of words accepted by A. A *transition system* $TS = (\Gamma, \gamma_0, \rightarrow)$ on an alphabet Δ consists of a set of states Γ, an initial state γ_0 and a labeled transition relation $\rightarrow \subseteq \Gamma \times \Delta \times \Gamma$ between the states. We restrict ourselves to deterministic transition systems.

We model different system architectures and programs as transition systems. Their languages consist of the input sequences for a controller, that can be generated by a given specified program and system architecture. This allows us to make formal statements about different models for controllers.

Let F be some function $F : A \rightarrow B$. Let $b \in B$, the function that assigns b to every input in A is denoted $F[b]$. It holds $\forall a \in A : F[b](a) = b$.

The function that differs from F only at the input $a \in A$ by assigning some $b \in B$ to it is denoted by $F[a \to b]$. It holds $F[a \to b](a) = b \wedge \forall a \neq \tilde{a} \in A : F[a \to b](\tilde{a}) = F(\tilde{a})$

Definition 1 *The transition system implementing a total store ordering architecture with a shared memory containing the variable set V with assigned values of D for a set of components P is defined as $Sys_{TSO} := (\Gamma, \gamma_0, \to)$, where Γ is the set of configurations, γ_0 is a starting configuration and $\to$ is a set of transitions.*

A configuration $\gamma = (Mem, W) \in \Gamma$ is a pair consisting of a

- *variable assignment Mem, which is a function $Mem : V \to D$ and a*
- *function $W : P \to Write^*$, assigning to each component p_i the content of its buffer $b_i \in Write^*$.*

The initial configuration $\gamma_0 = (Mem[0], W[\epsilon])$ has an initial value $0 \in D$ assigned to every variable and the empty word to every buffer. Let $\gamma, \gamma' \in \Gamma, a \in D, i \in \mathbb{N}, w = (w, x, v, p_i), r = (r, x, a, p_i)$. The transitions $\to \subseteq \Gamma \times Act \times \Gamma$ are as follows:

$$
\begin{aligned}
\gamma \xrightarrow{w^{out}} \gamma' \quad & \textit{if} \quad \gamma = (Mem, W),\ W(p_i) = w.\omega,\ \omega \in Write^*, \\
& \qquad \gamma' = (V[x \to v], W[p_i \to \omega]) \\
\gamma \xrightarrow{w^{in}} \gamma' \quad & \textit{if} \quad \gamma = (Mem, W), W(p_i) = \omega,\ w \in Write, \gamma' = (Mem, W[p_i \to \omega.w]) \\
\gamma \xrightarrow{r} \gamma' \quad & \textit{if} \quad \gamma = \gamma' = (Mem[x \to a], W),\ \nexists (w, x, p_i) \in W(p_i) \\
& \textit{or} \quad \gamma = \gamma' = (Mem, W[p_i \to \alpha.(w, x, a, p_i).\beta]),\ \alpha, \beta \in Write^*,\ \nexists (w, x, p_i) \in \beta
\end{aligned}
$$

Such a system allows any feasible computation and it enforces the correct use of the buffer. In order to mark the configurations that represent possible endings of computations, we denote by $\Gamma_F \subseteq \Gamma$ the configurations where all buffers are empty. We call them complete.

We model the programs as finite automata defining the language of all sequences of actions that can be fired by the program. Note that an automaton does not enforce correct behaviour of the buffer. This is done by the TS modeling the system.

A *finite automaton* A over an alphabet Σ consists of a tuple $A = (Q, q_o, \to, Q_F)$ of a finite set of states Q, an initial state $q_o \in Q$, a set of transitions $\to \subseteq Q \times \Sigma \times Q$ and a set of final states Q_F. If no set Q_F is specified, $Q = Q_F$ holds.

We only use *deterministic* automata where the transition $\to$ such that

$$q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'' \Rightarrow q' = q''$$

For two automata $A = (Q, q_0, \to, Q_F)$ and $A' = (Q', q'_0, \to', Q'_F)$, we define their interleaving $A|||A' := (Q \times Q', (q_o, q'_0), \to'', Q_F \times Q'_F)$ with

$$(q, q') \to'' (q_1, q'_1) \Leftrightarrow [q \to q_1 \wedge q' = q'_1] \vee [q' \to' q'_1 \wedge q = q_1]$$

It holds $L(A|||A') = L(A) \sqcup L(A')$.

A component p contains a sequence of write and reads. We give an automaton that ensures that the component sends those actions in the correct order. A program P is modeled to consist of its sequential components that are executed in parallel. We define the *program automaton* A_P as the interleaving of the automata of the components $A_P := |||_{p \in P} A_p$.

Definition 2 *Given a component $p = a_1 \dots a_n$, its corresponding automaton A_p is defined by $A_p := (Q, q_0, \rightarrow, \{q_n\})$ such that $Q = \{q_0, \dots, q_n\}$ and $\rightarrow$ as follows*

$$q_{i-1} \xrightarrow{r} q_i \;\; \textit{if} \;\; 1 \leq i \leq n,\; r = a_i,\; r \in Read$$
$$q_{i-1} \xrightarrow{w^{in}} q_i \;\; \textit{if} \;\; 1 \leq i \leq n,\; w = a_i,\; w \in Write$$
$$q \xrightarrow{w^{out}} q \;\; \textit{if} \;\; w \in Write,\; q \in Q$$

Example 1a (Dekker) The finite automaton for the program $p_1 = w_x.w_z.r_z.r_y$ of Example 1 is given in Figure 2. Note that any action w^{out} is allowed at any

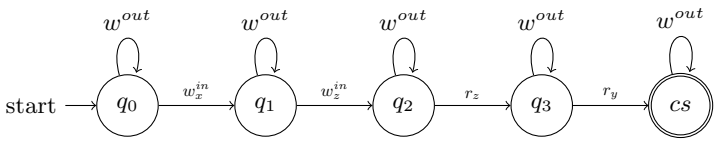

Fig. 2. The finite automaton for $p_1 = w_x.w_z.r_z.r_y$ in Example 1a

time independently of the writes that entered the buffer. The correctness of the buffer is ensured by the TS of the system.

We introduce a sequentially consistent system that has the same set of configurations Γ but does not use write buffers. We achieve this by ensuring that every action w^{in} entering the write buffer leaves it in the next transition w^{out}. We recall the transition system of a total store ordering is $Sys_{TSO} = (\Gamma, \gamma_0, \rightarrow)$. We define the sequentially consistent system with the restriction that a state with a nonempty buffer contains only outgoing transitions labelled with a write leaving the buffer w^{out}.

Definition 3 *The transition system modeling a sequentially consistent environment is $Sys_{SC} := (\Gamma, \gamma_0, \rightarrow')$ with $\rightarrow' \subset \rightarrow$ s.th for all $(\gamma, a, \gamma') \in \rightarrow$ holds $(\gamma, a, \gamma') \in \rightarrow'$ iff $\gamma = (Mem, W[\epsilon])$ for some memory state $Mem : V \rightarrow D$ or $a = w^{out}$ for some $w \in Write$.*

This system allows only sequentially consistent computations.

Example 1b (Dekker) Examine the program of Example 1: $P = \{p_1, p_2\}$ with $p_1 = w_x.w_z.r_z.r_y.$ and $p_2 = \mathbf{w_z.w_y.r_x}$
The TSO computation is $\tau = r_y.\mathbf{w_z.w_y}, \mathbf{r_x}.w_x.w_z$ Let the SC computation σ be

the order in which the actions were fired by D.
$\sigma = w_x.w_z.r_z.r_y.\mathbf{w_z}.\mathbf{w_y}, \mathbf{r_x}$
The SC computation gives us the incoming writes and reads and the TSO computation shows when a write leaves the buffer. This produces the following controller input:

$$w_x^{in}.w_z^{in}.r_z.r_y.\mathbf{w_z^{in}}.\mathbf{w_y^{in}}.\mathbf{w_z^{out}}.\mathbf{w_y^{out}}.\mathbf{r_x}.w_x^{out}.w_z^{out}$$

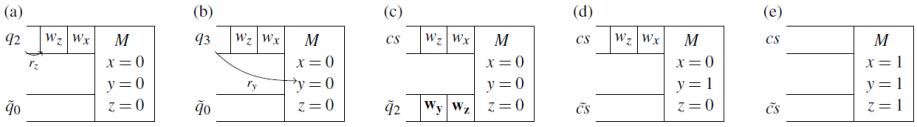

Fig. 3. An illustration of the inconsistent computation of Example 1 and 1b (mutex). In (a) w_x and w_z are added to the buffer of p_1 and r_z is an early read ($w_x^{in}.w_z^{in}.r_z$). In (b), r_y is read on the shared memory by p_1. Then $w_y.w_z$ are added to the buffer of p_2 and r_x is read on the shared memory ($\mathbf{w_z^{in}}.\mathbf{w_y^{in}}$) in (c). In (d), the buffer of p_2 is emptied ($\mathbf{r_x}.w_x^{out}.w_z^{out}$) and then $\mathbf{r_x}$ is read on the shared memory. The buffer of p_1 is emptied in (e) ($w_x^{out}.w_z^{out}$). The computation is $\tau = r_y.\mathbf{w_z}.\mathbf{w_y}.\mathbf{r_x}.w_x.w_z$. The input sequence is $w_x^{in}.w_z^{in}.r_z.r_y.\mathbf{w_z^{in}}.\mathbf{w_y^{in}}.\mathbf{w_z^{out}}.\mathbf{w_y^{out}}.\mathbf{r_x}.w_x^{out}.w_z^{out}$.

3 Consistency and Traces

We have informally referred to consistency of computations as the notion that computations exhibit the same behaviour. The most obvious interpretation is that a set of computations are consistent if no computation is feasible or each computation is feasible and ends in the same state of the system, meaning the values of the variables in the shared memory have to coincide after the executions. This was introduced by Owens as resultSC [Owe10]

Definition 4 *Let Sys_A and Sys_B be the transition systems of system architectures A and B. A computation σ on a system Sys_A is* state-consistent *with a computation τ on Sys_B iff they are both either not feasible or they end in configurations γ and γ' such that the shared memory of γ is in the same state as γ'.*

Definition 5 *A computation σ of a program P with Sys_A is* consistent *with Sys_B iff there is a computation σ' of P using Sys_B that is consistent with σ.*
A computation σ of a program P with Sys_A is sequentially consistent *iff it is consistent with Sys_{SC}.*

Definition 6 *A Program P is* robust *under Sys_A iff every computation of P with Sys_A is sequentially consistent.*

In the static analysis of a program under TSO, the robustness-problem needs to be solved. This was shown to be PSPACE-complete [BMM11]. We use controllers in order to avoid solving robustness and concentrate on sequential consistency instead. State-consistency is the weakest definition of consistency, it allows every computation that produces the correct variable assignments. However, it is unsuited for controllers, as we will see in Section 4. We will use traces, a stricter definition which does not allow some computations that end in the correct memory state. The concept of traces was introduced by Shasha and Snir [SS88]. To formalize traces, we define the following relations on the actions of a computation σ [SS88].

Definition 7 *The* program order relation *of a component* $\xrightarrow{po}_p$ *gives the order in which actions of the component p have been fired by the program. This is the order in which they were before rewriting. For a component p, it holds* $a \xrightarrow{po}_p b \Leftrightarrow p = \alpha.a.b.\beta$. *The po-relation is* $\xrightarrow{po} := \bigcup_{p \in P} \xrightarrow{po}_p$.

Definition 8 *The* store relation *of a computation* σ *and a variable x gives the order in which the actions of* σ *write on x. It holds* $w_x \xrightarrow{st}_x \tilde{w}_x$ *iff* $\sigma = \alpha.w_x.\beta.\tilde{w}_x.\gamma$ *and* β *contains no write on x.* $\xrightarrow{st} := \bigcup_{x \in V} \xrightarrow{st}_x$.

If $w \xrightarrow{st} w'$ holds, we say w' overwrites w.

Definition 9 *The* source relation *of a computation* σ *defines for each write* w_x *which read action* r_x *has read the value of x that was assigned by* w_x. *For any* $r_x \in \sigma$ *holds* $w_x \xrightarrow{src} r_x$ *iff* $\sigma = \alpha w_x.\beta.r_x.\gamma$ *where* β *contains no write on x. For any early read* $r_x \notin \sigma$ *with* $r_x \in p$ *holds* $w_x \xrightarrow{src} r_x$ *iff* $p = \alpha w_x.\beta.r_x.\gamma$ *where* β *contains no write on x.*

If $w \xrightarrow{src} r$ holds, we say r reads w or r reads the value of w.

The following descriptions of the relations between the actions use the buffer input In created by a computation σ. We can see that they are equivalent to the previous definitions using σ.

For an action a, we define $a' := a$, if it is a read and $a' := a^{in}$, if it is a write. The projection of an input sequence In' to its actions of component p is $In' \downarrow_p$.

Lemma 1 *For a given Input In, the relations between the actions are equivalently characterised by:*
$a \xrightarrow{po}_p b$, *iff there is a sequence* β *containing no* w^{in} *or r of p such that*

$$a', b' \in In \downarrow_p \wedge In = \alpha.a'.\beta.b'.\gamma$$

$w_x \xrightarrow{st}_x \tilde{w}_x$ *iff there is a sequence* β *without outgoing writes on x such that*

$$In = \alpha.w_x^{out}.\beta.\tilde{w}_x^{out}.\gamma$$

To characterize the source relation, we describe two cases. In the first, we assume there is no $\tilde{w}_x \in p$ such that

$$In = \alpha'.\tilde{w}_x^{in}.\beta'.r_x.\gamma'.\tilde{w}_x^{out}\delta'$$

$w_x \xrightarrow{src}_x r_x$, iff there is a sequence β containing no outgoing write on x such that

$$In = \alpha.w_x^{out}.\beta.r_x.\gamma \text{ (reorder)}$$

In the second case, we assume there is a component p such that $w_x, r_x \in p$ holds. $w_x \xrightarrow{src}_x r_x$, iff there is a sequence β containing no incoming write $\tilde{w}_x^{in}$ of p such that

$$In = \alpha.w_x^{in}.\beta.r_x.\gamma.w_x^{out}\delta \text{ (early read)}$$

Let $Actions(P)$ be the set of actions occurring in a program P.

Definition 10 *A* trace *$T = T(\sigma)$ of a computation σ of a program P consists of the actions of P and the relations between them.*

$$T := (Actions(P), \xrightarrow{po}, \xrightarrow{st}, \xrightarrow{src})$$

The same holds for a trace $T = T(In)$ of a buffer input In of a program P.

We can use the relation descriptions that are based on the input sequence to construct a trace from a sequentially read input. We add an action to the trace when it first occurs. Recall, that every write w occurs twice in the input: w^{in} and w^{out}.

When a new input-symbol is processed, the trace is updated with the following *trace updates* :

1. We add a program order relation for any action r or w^{in}, the start node of which is always the last action in the *po*-relation of the component.
2. We add a store relation $\tilde{w} \xrightarrow{st} w$, when a write action w leaves the buffer. The start node of this relation is the former last write action $\tilde{w}$ on the variable that accessed the shared memory.
3. We add a source relation $w \xrightarrow{src} r$, when a read action r occurs. The start node w of this source relation is either in the last write action on the variable in the memory or the components buffer if it is an early read.
4. We add a conflict relation $r \xrightarrow{cf} w$, when a write action w leaving the buffer is the goal of a store relation with a start node $\tilde{w}$, that is also the start node of a source relation $\tilde{w} \xrightarrow{src} r$.

The trace updates are directly derived from the definition of the relations given an input In and thus they are correct.

Definition 11 *A computation σ on a system Sys_A is* trace-consistent *with a computation τ on Sys_B iff $T(\sigma) = T(\tau)$*

We will use this model of trace-consistency to construct controllers and we will refer to trace-consistency simply as *consistency*. This notion of consistency has a property that is easier to check for violations than state-consistency. However, it requires that we extend our definition of traces. We add a new *conflict relation* cf. A read r_x is in cf-relation with a write w_x if there is a another write $\tilde{w}_x$ such that r_x reads the value written by $\tilde{w}_x$ and w_x overwrites it or if w_x is the first write on x and r_x reads the initial value of x.

$$r_x \xrightarrow{cf} w_x \Leftrightarrow \exists \tilde{w}_x(\tilde{w}_x \xrightarrow{src} r_x \wedge \tilde{w}_x \xrightarrow{st} w_x) \vee (\nexists \tilde{w}_x(\tilde{w}_x \xrightarrow{st} w_x) \wedge \nexists w'(w'_x \xrightarrow{src} r_x))$$

Alternatively, we say r_x is in cf-relation to w_x if w_x is the first action to write on x after r_x was executed and r_x did not read the value written by w_x (which may happen if r_x is an early read).

Since we only use the existence of st and src-relations and we already have definitions for these relations using the buffer input, it is not necessary to give a second characterization of the cf-relation using the input. These relations together form the *happens before relation*.

$$\xrightarrow{hb} := \xrightarrow{po} \cup \xrightarrow{st} \cup \xrightarrow{src} \cup \xrightarrow{cf}$$

An *extended trace* is $T := (Actions(P), \xrightarrow{po}, \xrightarrow{st}, \xrightarrow{src}, \xrightarrow{cf})$. Note the extended definition of the trace does not contain more information than the previous one.

Theorem 1 *A computation σ of a program P with Sys_A is sequentially consistent iff the extended trace contains no cycle.[SS88]*

We now have a notion of consistency that can be checked efficiently. It is trivial, to construct a computation σ of a program P such that $T(\sigma)$ is a cyclic extended trace and all actions of the program read and write only the initial value of the variables in the shared memory. After any computation of the program P, the memory remains unchanged and so every computation of the program is state-consistent. We see that the consistency model using traces is stricter than state-consistency.

From now on, we will only work with extended traces and thus, we will refer to them simply as traces. When not further specified, consistency refers to sequential consistency.

Let $r \xrightarrow{cf} w$ be a relation in a trace such that r and w belong to the same component p. By definition of the conflict relation, w has not left the buffer nor was it in the buffer when r was executed. This means w was fired by the program after r and thus $r \xrightarrow{po}{}^*_p w$ holds. Since we are only interested in the reachability properties of a trace, we can omit the relation $r \xrightarrow{cf} w$ from a trace.

From the input, we can obtain the TSO rewritings and thus the trace. See Figure 4.

Note that τ is not only obtained from σ through TSO rewriting, but the buffer also provides additional shuffling. This means, it is not sufficient to check only if τ is consistent with σ. We have to check, whether τ is consistent with any SC computation $\sigma' \in L(p_1) \sqcup\!\sqcup L(p_2)$.

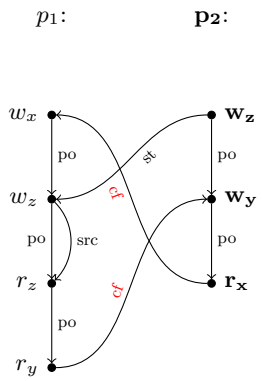

Fig. 4. The trace of τ in Example 1. The trace contains a circle so the computation is inconsistent

In our model, the input of a controller does not contain fence actions since they only influence the behaviour of the buffer and thus the shape of the input. In a trace, they are only connected with the program order relation and do not change traces with respect to acyclicity.

4 Controllers

Using the trace-consistency, we now introduce controllers. We research a controller that sequentially processes the buffer input created by a program execution and ensures consistency of the computation by delaying some actions. Before a write enters or leaves the buffer or a read is executed, the system requests to perform the action from the controller. The controller allows it, if it does not destroy sequential consistency. In order to achieve this, the controller has to obtain and interpret all the partial knowledge of the trace that is available from the processed input.

First, we look at a basic controller that does not necessarily process all available information, but eventually detects every inconsistency. We define a controller by the language of inputs it accepts for any given program using TSO architecture. A controller has to accepts all inputs of a program that have a consistent input of the same program in a sequential environment.

Definition 12 *A transition system C is a controller iff it holds*

$$\forall P \in \mathcal{P}\, \forall In \in L(A_P) \cap L(Sys_{TSO})$$

$$(In \in L(C) \Leftrightarrow \exists In' \in L(A_P) \cap L(Sys_{CS})\, (In \text{ is consistent with } In'))$$

In Section 6, we will show how a controller can be improved in order to find inconsistencies immediately.

We see that the notion of state-consistency is not useful for our purposes. The controller for the buffer is supposed to read the input sequentially and detect inconsistencies during the computation, state-consistency of a feasible computation depends solely on the state at the end of the computation. Any feasible computation can be extended with a suffix of write actions that change all variable assignments of the shared memory to any state. Since we do not now the program until the input has been processed, we cannot check an input for consistency until it has been processed. This defeats the purpose of an online analysis of the input.

Applying our knowledge of traces [SS88] gives us alternative definitions for a controller.

Lemma 2 *A TS C is a controller iff*

$$\forall P \in \mathcal{P}\, \forall In \in L(A_P) \cap L(Sys_{TSO})$$

$$(In \in L(C) \Leftrightarrow \exists In' \in L(A_P) \cap L(Sys_{CS})\, (T(In) = T(In')))$$

Lemma 3 *A TS C is a controller iff*

$$\forall P \in \mathcal{P}\, \forall In \in L(A_P) \cap L(Sys_{TSO})\, (In \in L(C) \Leftrightarrow T(In) \textit{ is cycle-free})$$

Lemma 3 gives us the basic idea behind the controllers we will introduce: we build a partial trace and check if it contains a cycle. In Section 5, we show under which assumptions a finite automaton can be used as a controller and what its space complexity is.

5 Finite Automata as Controllers

We now investigate different restrictions of a system and determine whether we can find a controller that is a finite automaton. We consider the size of the set of variables, their values and the maximal buffer size. We furthermore determine which of these parameters have to be finitely bounded in order to be able to construct a finite automaton as a controller. We will see that the following restrictions hold: the sets of variables has to be finite and the maximal length of the buffer content has to be finitely bounded.

We then show that such an automaton exists by giving a constructive proof in which we build an automaton, that stores finitely bounded variations of traces in its states.

Theorem 2 *Given a limited buffer and an unlimited number of boolean variables, no finite automaton is able to test any computation for consistency.*

Proof. Assume there is such a finite automaton A. There is an $n \in \mathbf{N}$ such that A has less than 2^n states. Given variables $X = \{x_1, ..., x_n\}$, we construct for every subset $X' = \{x_{i_1}, ...x_{i_k}\} \subseteq X$, $k \leq n$ the input In:

$$w_z := (w, z, 1, p_3), r_y := (r, y, 0, p_3),$$
$$w_y := (w, y, 1, p_1), w_y := (w, y, 1, p_1),$$
$$\forall 1 \leq l \leq k : w_{i_l} := (w, x_{i_l}, 1, p_l),$$
$$w_{i_k} := (w, x_{i_k}, 1, p_1), w_{i_k} := (w, x_{i_k}, 1, p_1)$$

$$In := w_z^{in}.r_y.w_y^{in}.w_y^{out}.w_{i_1}^{in}.w_{i_1}^{out}.w_{i_2}^{in}.w_{i_2}^{out}.\ldots.w_{i_k}^{in}.w_{i_k}^{out}$$

We define $r_j := (r, x_j, 1, p_2), \mathbf{w_z} := (w, z, 0, p_2), \mathbf{w_z} := (w, z, 0, p_2), w_z := (w, z, 1, p_3)$ and consider the following continuations In' of the input:

$$In' := r_j.\mathbf{w_z^{in}}.\mathbf{w_z^{out}}.w_z^{out}$$

The trace of an input $In.In'$ contains a cycle iff $x_j \in X'$, as shown in Figure 5. Since there are 2^n different possible sets X', A has at least 2^n states. This is in contradiction to the assumption that A has less then 2^n states. □

From the proof follows immediately that the size of an automaton is at least exponential in the number of variables.

Theorem 3 *Given an unlimited buffer and a limited number of variables, no finite automaton is able to test any computation for consistency.*

Proof. Assume there is such a finite automaton A. Let $V = \{x_1, x_2, \ldots, x_m, y\}$. There is an $n \in \mathbf{N}$ such that A has less than n^m states. We construct n sequences of reads $s_1, \ldots, s_n$. For every variable x_i, we assign a read r_{x_i} to one of the sequences. Within a sequence, the order of the reads are given by their indices: $i < j \Rightarrow r_{x_i} < r_{x_j}$. Given sequences $s_1, \ldots s_n$, we construct the input sequence

$$In(s_1, \ldots s_n) := w_y^{in}.s_1.w_y^{in}.s_2.w_y^{in}\ldots.w_y^{in}.s_n.w_y^{in}$$

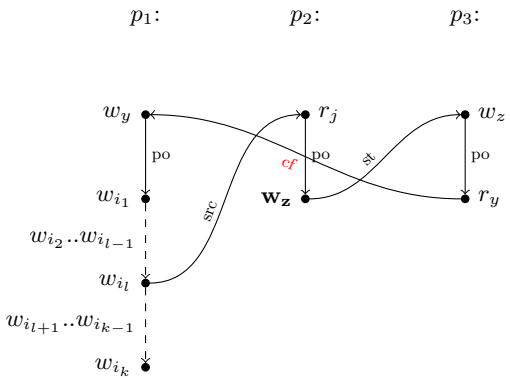

Fig. 5. The trace of $In.In'$ with $x_{i_l} = x_j \in X'$. The resulting *src*-relation completes the cycle and thus leads to inconsistency

There are n^m possibilities to assign the m reads to n positions in the buffer. Since A has less than n^m states, there must be two sets of sequences $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_n$ such that A is in the same state after processing $In(s_1, \ldots s_n)$ and $In(s'_1, \ldots s'_n)$.

Let $i \leq n$ be the smallest number that satisfies $s_i \neq s'_i$. Assume without loss of generality, that there is a read r_{x_j} such that $r_{x_j} \in s_i$ and $r_{x_j} \notin s'_i$. It follows that in the trace of $In(s'_1, \ldots s'_n)$, r_{x_j} occurs later in the *po*-relation. We give an input continuation

$$In'(r_{x_j}, i) := (w_y^{out})^i.\mathbf{w^{in}_{x_j}}.\mathbf{w^{out}_{x_j}}.\mathbf{w^{in}_{y}}.\mathbf{w^{out}_{y}}.(w_y^{out})^{n-i+1}$$

Both inputs contain a sequence of relations

$$r_{x_j} \xrightarrow{cf} \mathbf{w_{x_j}} \xrightarrow{po} \mathbf{w_y} \xrightarrow{st} w_y$$

But only the trace $T(In(s'_1, \ldots s'_n).In'(r_{x_j}, i))$ contains the relation $w_y \xrightarrow{po} r_{x_j}$ that completes the cycle. $In(s_1, \ldots s_n)$ is constructed such that any write w_y on y that precedes r_{x_j} in the *po*-relation already left the buffer before $(w_y^{out})^i$ has been processed. The write w_y that is reached by $\mathbf{w_y} \xrightarrow{st} w_y$ is contained in the sequence $(w_y^{out})^{n-i+1}$. Since $In(s_1, \ldots s_n).In'(r_{x_j}, i)$ is sequentially consistent while $T(In(s'_1, \ldots s'_n).In'(r_{x_j}, i))$ contains a cycle, A has to be in different states after processing $In(s_1, \ldots s_n)$ and $In(s'_1, \ldots s'_n)$. This is a contradiction to the assumption that A is in the same state. □

This proof shows that the size of a controller is at least polynomially dependent on the size of the buffer.

Lemma 4 *If a trace contains more than one conflict relation from a program component p to a write action w, then all except the last can be omitted and the trace remains cyclic (respectively cycle-free).*

Proof. If a program contains two reads $r_1 \xrightarrow{po}{}^* r_2$ and there is a w with $r_1 \xrightarrow{cf} w$ and $r_2 \xrightarrow{cf} w$ and there is a cycle containing $r_1 \xrightarrow{cf} w$, then we can replace this relation with $r_1 \xrightarrow{po}{}^* r_2 \xrightarrow{cf} w$. We have found another cycle containing $r_2 \xrightarrow{cf} w$ and not $r_1 \xrightarrow{cf} w$. So the relation $r_1 \xrightarrow{cf} w$ does not effect whether the trace is cyclic. □

Lemma 5 *If a trace contains an early read r_1 and po-relation $r_1 \xrightarrow{po} r_2$ and we exchange r_1 and r_2 in the po-relation using the following modifications*

$$a \xrightarrow{po} r_1 \Rightarrow a \xrightarrow{po} r_2$$
$$r_1 \xrightarrow{po} r_2 \Rightarrow r_2 \xrightarrow{po} r_1$$
$$r_2 \xrightarrow{po} a \Rightarrow r_1 \xrightarrow{po} a$$

then the trace remains cyclic (cycle-free).

We show that any cycle remains:

Proof. If the trace contains a cycle

$$a \xrightarrow{po} r_1 \xrightarrow{po} r_2 \xrightarrow{po} b \xrightarrow{hb}{}^* a$$

then the modified trace contains a cycle $a \xrightarrow{po} r_2 \xrightarrow{po} r_1 \xrightarrow{po} b \xrightarrow{hb}{}^* a$.

If the trace contains a cycle

$$a \xrightarrow{po} r_1 \xrightarrow{po} r_2 \xrightarrow{hb} b \xrightarrow{hb}{}^* a, \text{ with } r_2 \xrightarrow{hb} b \notin \xrightarrow{po}$$

then the modified trace contains a cycle $a \xrightarrow{po} r_2 \xrightarrow{hb} b \xrightarrow{hb}{}^* a$.

If the trace contains a cycle

$$a \xrightarrow{hb} r_2 \xrightarrow{po} b \xrightarrow{hb}{}^* a, \text{ with } a \xrightarrow{hb} r_1 \notin \xrightarrow{po}$$

then the modified trace contains a cycle $a \xrightarrow{hb} r_2 \xrightarrow{po} r_1 \xrightarrow{po} b \xrightarrow{hb}{}^* a$.

If the trace contains a cycle

$$a \xrightarrow{hb} r_1 \xrightarrow{po} r_2 \xrightarrow{hb}{}^* a \text{ with } a \xrightarrow{hb} r_1 \notin \xrightarrow{po}$$

then it holds $a \xrightarrow{src} r_1$ and since r_1 is an early read, $a \xrightarrow{po}{}^* r_1$ holds. This is a contradiction.

We show that the modification does not add any cycles. The first 3 cases are analogue to the proof that no cycles are added. If a cycle

$$a \xrightarrow{hb} r_2 \xrightarrow{po} r_1 \xrightarrow{hb} b \xrightarrow{hb}{}^* a$$

is created in the modified trace then the original trace must contain a cycle $a \xrightarrow{po}{}^* r_1 \xrightarrow{po} r_2 \xrightarrow{hb}{}^* a$. This is the case because either $a \xrightarrow{po} r_1$ holds or $a \xrightarrow{src} r_1$ does. Since r_1 is an early read, this implies $a \xrightarrow{po}{}^* r_1$. □

We call the modification of the lemma an *exchange modification*. This technical lemma shows that it is sufficient for a controller to store which early reads of a component p are processed between the the time a write w enters the buffer of p and its direct successor enters the buffer. We do not need to know their order or where they occur in the program order related to other reads of p in the same interval. This can be simplified further using the following lemma.

Lemma 6 *If a trace contains two early reads r_1 and r_2 that read the same write w ($w \xrightarrow{src} r_1 \wedge w \xrightarrow{src} r_2$) and occur back-to-back ($r_1 \xrightarrow{po} r_2$), we modify the trace by removing r_2 using the following modification: $r_2 \xrightarrow{hb} a \Rightarrow r_1 \xrightarrow{hb} a$*
The trace remains cyclic (cycle-free).

Proof. Assume a cycle is removed. This cycle has the form $a \xrightarrow{hb} r_2 \xrightarrow{hb}{}^{*} a$. Since r_2 is an early read and either $a \xrightarrow{po} r_2$ or $a \xrightarrow{src} r_2$ holds, the cycle has the form $a \xrightarrow{hb}{}^{*} r_1 \xrightarrow{po} r_2 \xrightarrow{hb}{}^{*} a$. The modified trace contains a cycle $a \xrightarrow{hb}{}^{*} r_1 \xrightarrow{hb}{}^{*} a$. Obviously, no cycle is added. □

We call this modification a *removal modification*.

We now know, that instead of storing every early read that occurs between two writes of a component p, which is an unbounded quantity, it is sufficient to store only one early read for every write in the buffer that was read by an early read occurring between the two writes. This is bounded by the buffer length.

Theorem 4 *Given a finite buffer and limited number of variables and values, a finite automaton exists that checks for consistency.*

Proof. We construct an automaton such that the states represent the possible traces. If the trace contains a cycle, it is a non-accepting state. Since the number of possible traces is infinite, we need to reduce the trace to a finitely bounded version with limited size. This is called a *compact trace*.

The controller needs to construct a trace from a sequentially read input. We recall the trace updates:

1. We add a program order relation for any action r or w^{in}, the start node of which is always the last action in the *po*-relation of the component.
2. We add a store relation $\tilde{w} \xrightarrow{st} w$, when a write action w leaves the buffer. The start node of this relation is the former last write action $\tilde{w}$ on the variable that accessed the shared memory.
3. We add a source relation $w \xrightarrow{src} r$, when a read action r occurs. The start node w of this source relation is either in the last write action on the variable in the memory or the components buffer if it is an early read.
4. We add a conflict relation $r \xrightarrow{cf} w$, when a write action w leaving the buffer is the goal of a store relation with a start node $\tilde{w}$, that is also the start node of a source relation $\tilde{w} \xrightarrow{src} r$.

Except for the *src*-relations to early reads, these updates disregards store actions until they leave the buffer. This means that inconsistencies are not detected as early as possible. In Example 1, it only forms a cycle when the buffer is emptied. This is sufficient for now. In Chapter 6, we will discuss how to modify a controller in order to detect inconsistencies immediately.

For this method, it is not necessary to know the complete trace of the processed input to check for consistency. In order to check for cycles in the trace, we only need to know the reachability between the vertices to which new relations are added. When a new relation $a \to b$ is added, a cycle is formed iff a is reachable from b. If this is the case, the controller enters an non-accepting state.

Note that in every update, we only need the last writes that accessed a variable in memory and the buffer content to add *st*- and *src*-relations. For the conflict relation, we need a read r that may be at any point in the trace. Let $r_{i,latest}$ be the latest read that accesses the latest write w on the shared memory on variable x. The read r satisfies $w \xrightarrow{src} r$ with r in p_i. Per definition, $r_{i,latest}$ occurs after r and since it is not an early read and w is the latest write on x, $r_{i,latest}$ reads the value of w. It holds w also satisfies $w \xrightarrow{src} r_{i,latest}$ with. Using Lemma 4, we only need to check the latest read on every variable for every component.

If an outgoing cf-relation is added to an early read, then the write action w with $w \xrightarrow{src} r$ has left the buffer and was overwritten by another write $\tilde{w}$ such that $w \xrightarrow{st} \tilde{w}$. This means at the time a cf-relation can be added, the early read is already a read on a write action that has accessed the shared memory.

For each component, the last action sent by the program is the only action from which an outgoing *po*-relation of this component can be added. So we know the only actions that a newly added relation can start from are the following:

- The last writes accessing the memory on each variable
- The last reads accessing one of the last writes for each variable and each component
- The writes in the buffers
- The last action fired by the program

These are called the *relevant actions*. They are illustrated in Figure 6.

When a buffer element w_x is accessed by an early read r_x, we first store the read implicitly by adding a source relation from w_x to the last element in the buffer at the execution time of r_x. When w leaves the buffer, we store r explicitly as the last read on x of the component. Then we add it to the *po*-relation directly before the buffer element following the one we marked earlier with the *src*-relation.

The compact trace is stored in a state of the finite automaton as a directed graph and the vertices are the relevant actions to which outgoing relations can still be added as illustrated in Figure 6. The reachability property consists of the happens before relation of a trace as well as its transitive closure. We store the relations as arcs labelled with the names of the relations (src, st, po_i, cf)

where po_i is the po-relation between actions of component p_i. We also add the transitive closure of this modified happens before relation. When we use the term

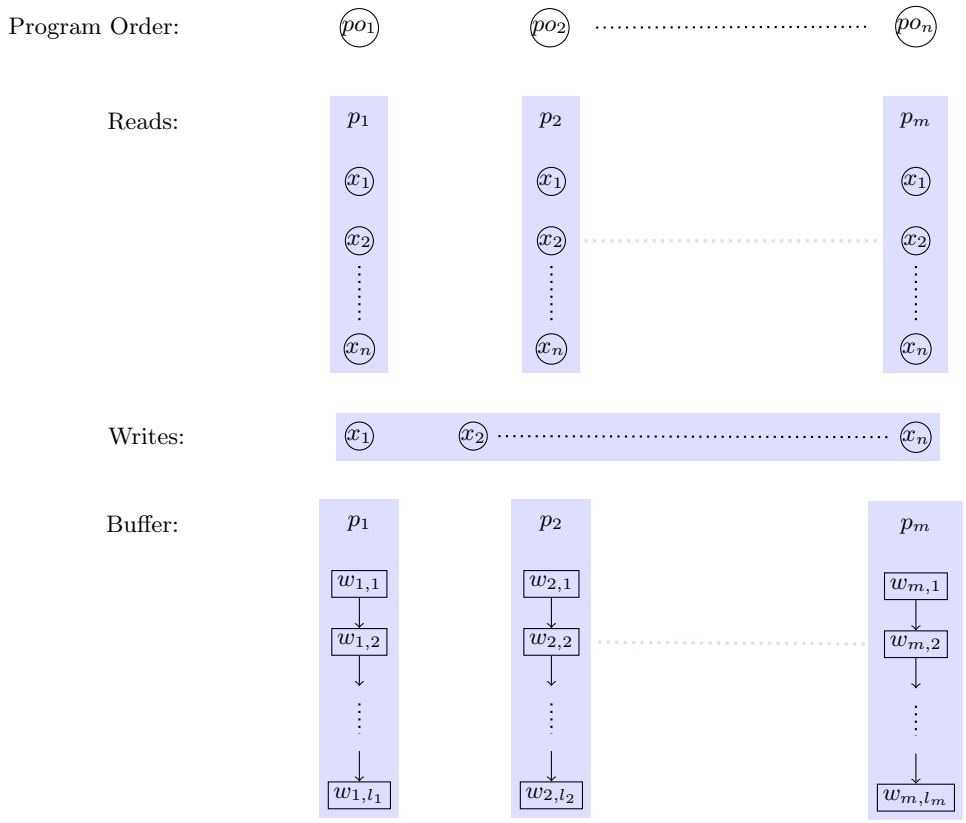

Fig. 6. The compact trace is stored as a graph. Let $P = \{p_1, \dots p_m\}$ and $V = \{x_1, \dots x_n\}$, $i \leq n$, $j \leq m$. For every component p_j there is a a vertex x_i for the last read on x_i executed by p_j and a vertex w for every buffer element of p_j. For every variable x_j, there is a vertex x_j for the last write performed globally on x_j. For every component p_i, there is an a vertex po_i containing the last action in the components po-relation. Let $l(i)$ the size of the buffer content of component p_i and $(w_{i,1}, \dots w_{i,l(i)})$ its buffer content.

moving an action or replacing another vertex, it means we remove all incoming and outgoing relations from the target vertex and replace them with those of the moving action. Then we remove the previous vertex of the moved action.

The compact trace is updated using the following *compact trace rules*:

Rule 1 $a = (w, x, i, p_l)^{in}$**:** Add a to the buffer, then add a po_l-relation from the element of p_l which has no outgoing po_l-relation to a and update the tran-

sitive closure.

Rule 2 $a = (w, x, i, p_l)^{out}$**:** Add store relation from the last write to x in the memory $a' = (w, x, j, p_k)$ to a in the buffer.
If a' has an outgoing source relation $a' \xrightarrow{src} r$, then add $r \xrightarrow{cf} a$.
Update the transitive closure.
If the compact trace is cycle-free, remove a from the buffer and replace a' with it (including its relations). If a' is the last element reached by po_k in the trace, then we move a' in the vertex po_k. If there is already an action occupying the vertex (i.e. if it is connected to some vertex), then we replace it.
If there is a relation $a \xrightarrow{src} w$ with w a buffer element (indicating an early read on a), then remove the relation. Create a new read r_x. If w is not the last bufferelement, there is a write w' directly after w in the buffer. Replace the relation $b \xrightarrow{po} w'$ with $b \xrightarrow{po} r_x \xrightarrow{po} w'$.
If w is the last buffer element, add a po_l-relation from the element with incoming but no outgoing po_l-relations to r_x. Then add $a \xrightarrow{src} r_x$ to the compact trace.
Replace the node of last read p on x with r_x and update the transitive closure.

Rule 3 $a = (r, x, i, p_l)$**:** If there is no write to x in the buffer of p_l, add a po_l-relation from the element of p_l which has no outgoing po_l-relation to a. Then add the source relation from the last write to x in the memory to a and update transitive closure. Remove the last read of x from component p_l and replace it with a.
If there is a write w_x such that w_x is the last write on x in the buffer of p_l, then add a source relation from w_x to the last buffer element.

It makes no difference, whether we keep the (unconnected) vertices of empty places in the buffer in the compact trace or not. The compact trace of the initial state consists of the graph with an empty set of edges.

It remains to be proven, whether the cyclic property of the compact trace created by an input In is the same as the cyclic property of the non-compact trace of In. We show that the reachability property of the compact trace T' constructed from In is the same as the reachability of the relevant actions of a non-compact trace T that was modified from $T(In)$ using the exchange and removal modifications given in Lemma 5 and 6. We say the compact trace T' is *equivalent* to T if this holds. Since the trace of an empty computation is obviously equivalent to the initial compact trace containing no relations, it is sufficient to prove, that the application of the compact trace rules is equivalent to the application of the trace updates in combination with the exchange and removal modifications.

By definition of the transitive closure, the following holds for the relations of the compact trace:

$$(a \xrightarrow{hb}{}^{*}\ b \xrightarrow{hb}{}^{*}\ c \Rightarrow a \xrightarrow{hb}{}^{*}\ c) \wedge (a \overset{hb}{\not\to}{}^{*}\ b \Rightarrow \nexists b \in Act : a \to^{*} b \to^{*} c)$$

It follows that once the transitive closure has been updated, we can delete actions without affecting reachability of the remaining actions.

We show, that every relation added by a compact-rule is equivalent to some modifications and applications of trace updates.

- Compact rule 1 obviously adds relations consistent with the first trace update which is the only one that applies to a write entering the buffer.
- Compact rule 2 adds cf-relations and st-relations the same as traces updates 3 and 4. If a' is no longer necessary to add new relations, we can safely remove the old write a' without destroying any paths in the trace containing a' since we store the transitive closure instead of just the hb-relation. However, if there is an incoming po_k-relation on a' but no outgoing one, then this will be added when the next input on p_i is processed and we cannot remove a'. Since a' is the last action in the po-relation of p_k, this does not hold for the previous action of vertex po_k and we can now safely remove and replace it.
  If there is a src-relation to another buffer element, then there was a early read r s.th. $a \xrightarrow{src} r$ holds and the src-relation is added consistent with the third trace rule. The po-relations added for r are equivalent to the po-relations added by the first trace rule and then modified by repeatedly applying the exchange modification to r.
- If the processed read a is not an early read, then compact rule 3 is analogue to the first and third rule. When a is an early read, then the src-relation to the last buffer element w is added and rule 2 correctly processes the read later.

We now show that for every relation added by a trace update, changes are made to the compact trace that are equivalent to the relation added by the trace update and some modifications.

- The first trace update construct the po-relation of the trace. In compact rules 1 and 3 a po-relation on the relevant cases is constructed that leaves out any early reads. Let $r1, \cdots, r_n$ be early reads and a and b are not early reads. The relation $a \xrightarrow{po}_p r_1 \xrightarrow{po}_p r_2 \ \ldots \xrightarrow{po}_p r_n \xrightarrow{po}_p b$ in the trace is only partly implemented . Compact rules 1 and 3 create the relation $a \xrightarrow{po}_p b$. Afterwards, some reads are inserted in a different place of the po-relation by compact rule 2. We have already shown, that a compact rule 2 adds early reads consistently with the application of trace rule 1 and modifications.
  If there is an early read r that is not added by compact rule 2, then another early read $\tilde{r}$ on the same write was already added. This is consistent with applying the exchange modification to read $\tilde{r}$, until $\tilde{r} \xrightarrow{po}_p r$ holds and then the removal modification on r.
- Trace update 2 is equivalent to the st-relation added in rule 2.
- Trace update 3 is equivalent to compact rule 3 if the read is not an early read.
  If it is an early read, either the src-relation is added in compact rule 2 equivalent to the application of some exchange modifications or the read is not added, which means the removal modification can be applied.

– Trace update 4 is equivalent to compact rule 2 if the read r is not an early read and it is a latest read.
  We have shown that if r is not a latest read, it is sufficient to add the conflict relation from the latest read. This is done in compact rule 2
  If r is an early read on a write w and r is added to the compact trace, then r is added to the last reads of the component at the execution time of w^{out}. For every write $\tilde{w}$ with $r \xrightarrow{cf} \tilde{w}$ holds that $\tilde{w}^o ut$ was executed after w^{out}, since $w \xrightarrow{st} \tilde{w}$ holds. If follows that the relation $r \xrightarrow{cf} \tilde{w}$ is added by compact rule 2.

For a given input, the compact rules affect the cyclic property of the compact trace the same way the trace updates affect the trace. From the correctness of the compact rules follows, that the given finite automaton is a controller. □

The following example gives us an application of the 3 different updates that occur on the compact trace.

Example 2 (Compact Trace Updates) These are the updates occurring in Figure 7:[1]

Update 1: In component p_2, input w^{in} is processed by adding w to the buffer and adding the *po*-relation from the last executed input on p_2, in this case this is the last buffer element $w_{2,l(2)}$.

Update 2: In p_1, $w_{x_2}^{out}$ is processed by removing the first buffer element w_{x_2} from the buffer, adding the *st*-relation from the last write (write-vertex x_2) and then replacing it.

Update 3: In p_n, a read action r_{x_1} is added by replacing the last read action of p_n on x_1 and adding a *po*-relation from the last vertex reached by program order relation. In this case it is $w_{m,l(m)}$.

Note that in updates 2 and 3 the last actions reachable by the components program order are the last buffer elements. This means the vertices po_2 and po_3 are not necessary in this case.
There are no early reads in this example.

Example 2a (A Compact Trace Using a po-vertex) We examine the compact trace after processing the following input.

$$w_x^{in}, w_x^{out}, \mathbf{w_x^{in}}, \mathbf{w_x^{out}}, r_y$$

The last write on x came from p_2 and replaces the previous w_x on the vertex which is moved to po_1. When r_y is fired, we add a po_1-relation from the vertex po_1 to the read vertex x of p_1.

These theorems show that a finite controller C exists if and only if the set of variables and the maximal length of a buffer are finitely bounded.

[1] The colours refer to the respective parts of the figure.

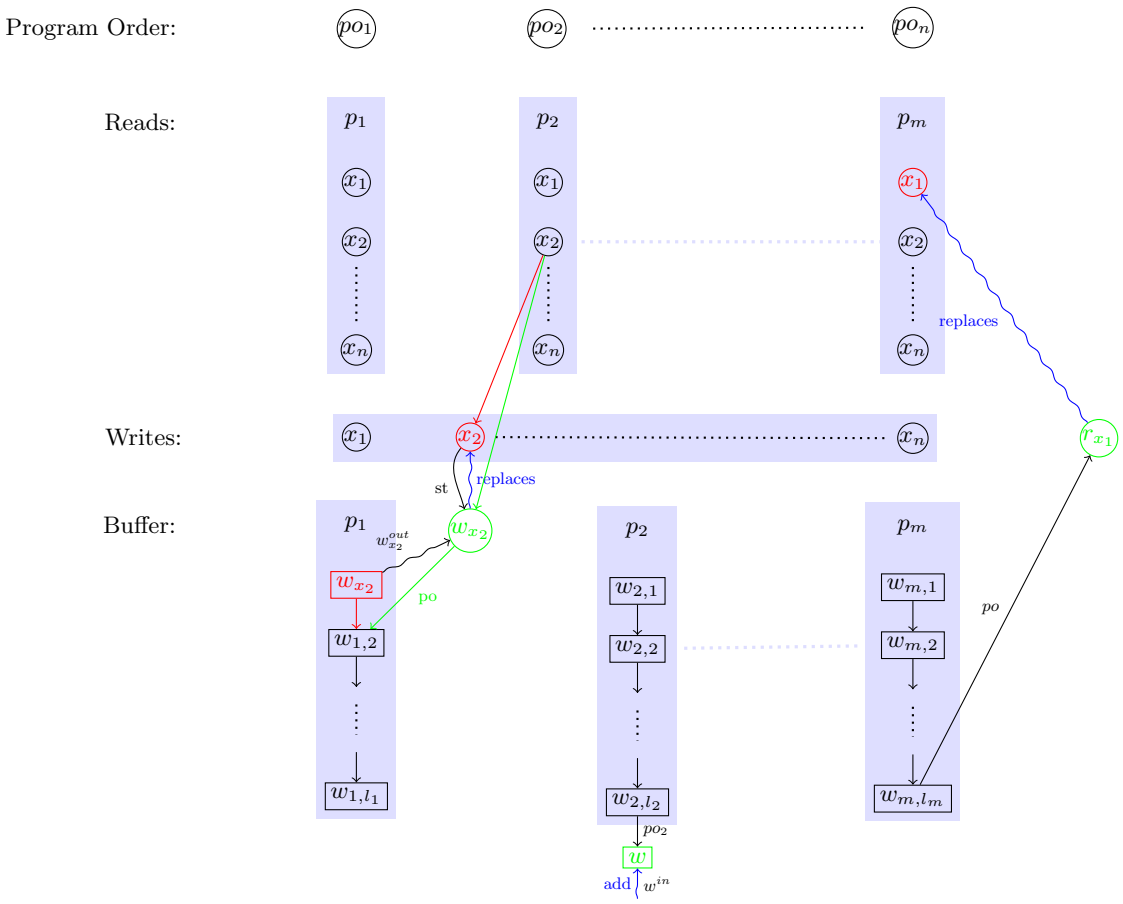

Fig. 7. Illustration of Example 2: The 3 updates of of the compact trace when an action is executed. Update 1 (w^{in}) is illustrated on component p_2, Update 2 ($w^{out}_{x_2}$) occurs in p_1 and Update 3 (r_{x_1}) in p_n.

Theorem 5 *A controller C that is a finite automaton exists under a system Sys_{TSO} iff for Sys_{TSO} holds*

$$\exists i, j \in \mathbb{N} : \ |V| < i \wedge \ \forall l \leq n : |b_l| < j$$

Proof. We examine the size of the finite controller we constructed in order to obtain an upper bound for the size of a finite controller with a minimal set of states. Let B be the maximal length of a buffer.

The set of vertices consists of $|P|$ nodes for the last actions in the *po*-relation, $|P| \cdot |V|$ for last reads, $|V|$ for the last writes and maximal $B \cdot |P|$ for the buffer content. Between this vertices are directed edges without loops (start vertex and end vertex of a edge are different). There are

$$|P| + |V| \cdot |P| + |V| + B \cdot |P|$$

many vertices and

$$(|P| + |V| \cdot |P| + |V| + B \cdot |P|) \cdot (|P + |V| \cdot |P| + |V| + B \cdot |P| - 1)$$

many possible edges with with $3 + |P|$ different labels. Additionally, each place in the buffer is assigned a write or it is empty, which gives us $(|V|+1)^{B \cdot |P|}$ many configurations of the write buffer. The number of states $|Q|$ of the constructed automaton consists of the number of possible relations between the vertices and the buffer content. It has the following upper bound.

$$|Q| \leq (3 + |P|)^{(|P|+|V|\cdot|P|+|V|+B\cdot|P|)\cdot(|P+|V|\cdot|P|+|V|+B\cdot|P|-1)} \cdot (|V| + 1)^{B \cdot |P|}$$

□

Since the term of the proof in the previous theorem grows exponentially in every parameter. it is not a very useful as an upper bound. We will find a better estimation in Section 9.

The proofs of the previous Theorems 3 and 2 show that the size of a minimal finite controller is at least exponential in the number of variables and linear in the length of the buffer.

6 Delayed Reaction to Inconsistencies

A controller does not necessarily find inconsistencies at the time they occur. The controller that is introduced in the proof of Theorem 4 processes write actions when they leave the buffer, whereas an improved controller might analyse the properties of possible traces that will occur depending on the order in which the buffer is emptied. In Example 1, we see that both components have entered the critical sections. Thus the computation is inconsistent. This is the case although their actions have not left the buffer yet and the trace constructed from the partial input does not yet contain a cycle.

We say a transition system reaches an *error state* if it cannot process an input symbol. A transition system processing input a in state q reaches an error state iff there is no transition $q \xrightarrow{a} q'$.

A controller A is called a *fastest controller* if there is no controller A' that enters an error state after reading an input In while A has not reached an error state after processing In. A fastest controller detects inconsistency at the earliest possible moment. This property is necessary for a controller that causes the system to actively avoid inconsistent computations by analysing the input elements before they are executed.

Lemma 7 *Given controllers A and A', A' is a fastest controller iff the following holds:*
After processing an input in, A' enters an error state iff A either enters an error state or a state s from which every continuation In' such that In' forms a complete computation *(i.e. ends in an empty buffer) enters an error state.*

Proof. Let A' be a fastest controller. Assume A' enters an error state after processing In, but A enters a state s that leads to an accepting state after processing a rest input In'. Then $In.In'$ is a consistent input and A' is in an accepting state after processing it. Since A' enters an error state after processing In and an automaton cannot leave an error state, A' does not accept $In.In'$. This is a contradiction to $In.In'$ being consistent.

Assume A' does not enter an error state after reading In, but A does. Since A is a controller, In is an inconsistent input and A' does not yet detect the inconsistency. This is a contradiction to A' being a fastest controller.

Assume A' does not enter an error state after reading In and A enters a state that enters an error state after processing any In' that forms a complete computation. Since A' is a fastest controller, In is a consistent input. There is a continuation In' that completes the computation and leads to an accepting state of A'. Since $In.In'$ is a complete computation and A' accepts it, it is consistent and A also accepts it. This is a contradiction to the fact that there is no continuation of In leading A to an accepting state.

Let A' be a controller, but not a fastest. There is an inconsistent input In, such that A' accepts In. Since A is a correct controller, it either reaches an error state after processing In or a state that cannot reach an accepting state after processing a rest input In' that forms a complete computation. □

Let B be the maximal length of a write buffer and n the number of components. We use a simple automaton $Count$ that identifies states with empty buffers by counting the processed writes leaving the buffer w^{out} and subtracting the incoming writes w^{in}. We define $Count = (Q = \{q_0, \ldots, q_{B \cdot n}\}, q_0, \rightarrow, Q_F = \{q_0\})$, where the following holds.

$$\forall i < B \cdot n(q_i \xrightarrow{w^{in}} q_{i+1} \wedge q_{i+1} \xrightarrow{w^{out}} q_i) \wedge \forall q \in Q : q \xrightarrow{r} q$$

If the buffer is unbounded, a similar transition system exists with an infinite number of states.

For two automata $A = (Q, q_0, \rightarrow, Q_F), A' = (Q', q'_0, \rightarrow', Q'_F)$ we define the parallel composition that accepts the intersection of the languages of A and A': $A||A' := (Q \times Q', (q_o, q'_0), \rightarrow'', Q_F \times Q'_F)$ with $(q, q') \rightarrow'' (q_1, q'_1) \Leftrightarrow [q \rightarrow q_1 \wedge q' \rightarrow' q'_1]$. It holds $L(A||A') = L(A) \cap L(A')$.

We now use *Count* to enhance a controller in such a way that the states reachable by complete computations are identified and every such state is only reachable by complete computation. There is no state that is reachable by a complete computation and an incomplete computation. We intersect the controller and *Count* and get a new controller where the complete states are the states marked by *Count* with q_0 as states where the buffer is empty.

Lemma 8 *Given a controller A, a state (s, q) of $A||Count$ is reached via complete computations iff $q = q_0$ holds.*

Proof. According to the definition, a computation is complete, if the buffer is empty. This is the case iff *Count* is in state q_0. So a state disables outgoing actions iff the paths leading to it represent complete computations. □

A state of a controller that satisfies the condition in Lemma 8 is called a *complete state*. We use Lemma 7 and 8 to construct Algorithm 6.1 that creates a fastest controller from a controller by analysing if a state can reach accepting complete states.

Algorithm 6.1 Create Fastest Controller

Require: Controller A
Construct $A||Count$
Identify complete states of $A||Count$
Create reachability matrix of $A||Count$
Turn every state that does not reach an accepting complete state into an error state
return Resulting fastest Controller A'

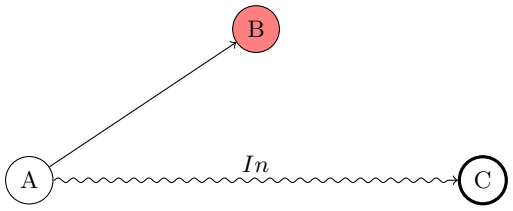

Fig. 8. Algorithm 6.1 modifies $A||Count$. State A reaches accepting complete state $C = (q, q_0)$ via input In. State B does not reach an accepting complete state.It is removed from the accepting states.

The algorithm is computable according to Lemma 8. Since the algorithm only adds error states, any input that is not accepted by A is not accepted by A'. If an input In is accepted by A, then the run of A forms a path to an accepting state. Since In ends in a state with empty buffer by definition, in is accepted by $Count$ and so it is also accepted by the parallel composition of A and $Count$ Every state on that path reaches an accepting complete state and is not changed by the algorithm. The same path also forms an accepting run of A' and the input is accepted by A'.

Let the state q be such that it does not reach an accepting complete state. Every continuation In' that completes the input enters an error state from q. See Figure 8 The last step of the algorithm removes every such state. This means there is now an input continuation for every state that leads to an accepting complete state. Thus the input controller A and the output controller A' satisfy the condition of Lemma 7.

7 Preprocessing

In order to reduce the complexity of a controller, we explore the notion of transferring some tasks to a preprocessing step. We perform a static analysis on a program and use it to provide a controller with additional information about the program that is being processed. This might increase efficiency and reduce delays. Since by definition, a controller has to work correctly for every program, the information provided in addition to the input sequence is optional. The controller retains full functionality without it. It might also be useful to examine controllers that require some information about the program in order to function properly. We give a relaxed definition of controllers, that does not require the controller to detect every inconsistency.

Incomplete Controller

We introduce a variation of controllers, that allows some inconsistent computations.

Definition 13 *Let C' be a controller. We call C an* incomplete controller *iff*

$$L(C') \cap L(Sys_{TSO}) \subset L(C) \cap L(Sys_{TSO}) \subset L(Sys_{TSO})$$

One possible way to construct an incomplete controller is to construct a controller that examines only some of the components. We allow those actions of the input that belong to a real subset of the concurrent programs.

Definition 14 *Let $In|_I, I \subset \{1 \dots n\}$ be the projection of the input In on actions belonging to $p_i, i \in I$. Let C be a controller, $C[I]$ is a* reduced controller *iff*

$$In \in L(C[I]) \cap L(Sys_{TSO}) \Leftrightarrow In|_I \in L(C) \cap L(Sys_{TSO})$$

Reduced controllers are of importance when studying distributed controllers (Section 8), since they they can be used a a basis for the construction of agents.

Another intuitive incomplete controller is one that only examines the actions of some variables:

Definition 15 *Let $In|_X, X \subset V$ be the projection of the input In on the actions on variables in X. Let C be a controller, $C[X]$ is a* variable controller *iff*

$$In \in L(C[X]) \cap L(Sys_{TSO}) \Leftrightarrow In|_X \in L(C) \cap L(Sys_{TSO})$$

Combining Preprocessing and Controllers

We can ensure sequential consistency of the computations of a program by combining a static analysis with an incomplete controller.

We enhance a reduced controller such that it performs as a controller for a given program $P \in \mathcal{P}$, by providing it with the set of component that is needed to function correctly on P. In the preprocessing, we determine which set of components, represented by their indices $I(P)$, is not necessary to test a computation for inconsistencies. The set $I(P)$ is a set of indices such that every computation obtained by reducing an inconsistent computation of P to actions of the components p_i with $i \in P(I)$ is also inconsistent. Trivially, this is the set of all component indices. We require $I(P)$ to be minimal, although any proper subset of the components which are non-empty would increase efficiency. We use a dynamic version of a reduced controller that is provided with a parameter I and reduces the input to the corresponding components. We provide a reduced controller with the set $I(P)$ before execution of P and ensure the consistency of the computation of P:

$$\forall P \in \mathcal{P}\, \forall In \in L(A_P) \cap L(Sys_{TSO})$$

$$(In \in L(C[I(P)]) \Leftrightarrow \exists In' \in L(A_P) \cap L(Sys_{CS})\, (In \text{ is consistent with } In'))$$

This combination of preprocessing and a reduced controller has the following advantage. Since a component p_i, such that $i \in I(P)$ is not being processed by the reduced controller, there is no delay on that component.

Preprocessing can also be used to identify the set of variables $X(P)$ that occur in multiple components of program P. We provide a dynamic variable controller with the set $X(P)$ before the execution of P and ensure consistency of its execution:

$$\forall P \in \mathcal{P}\, \forall In \in L(A_P) \cap L(Sys_{TSO})$$

$$(In \in L(C[X(P)]) \Leftrightarrow \exists In' \in L(A_P) \cap L(Sys_{CS})\, (In \text{ is consistent with } In'))$$

Since an action on a variable $x \notin X$ is not processed, the controller only has to test if $x \in X$ holds. Depending on the size of X and its internal representation, this is a short delay. Depending on the controller, we can expect the processing of actions on variables in X to be faster than on a similar controller that is not limited to a subset of variables. The size of the set of provided variables X can

be reduced further by limiting it to the smallest set of variables X' that satisfies the following condition: Every input sequence In of P is inconsistent iff $In \downarrow_X$ is inconsistent. This is obviously some subset of the variables occurring in multiple components.

The preprocessing does not have to be limited to providing information, it can also modify the program. We can greatly improve the efficiency of a variable controller by setting a constant limit $k \in \mathbb{N}$ to the size of the parameter X. If a program has more than k variables that occur in multiple components, we need additional restrictions on the possible computations. We add fences to the program such that every remaining inconsistent computation is identified by $C[X]$. We choose a set X, such that the number of introduced fences is minimal. This ensures that every computation of the modified program with the variable controller is consistent.

If a program contains more than k variables occurring in multiple components, this method decreases the delay caused by the controller. Since the number of actions that are processed is reduced, the processing time for such actions and the time required to test if an action needs to be processed is also decreased. However we have now added additional delays caused by the fences. This is a trade-off between the gained efficiency of the controller and the additional delays caused by fences. This method is advantageous for a program, where a small number of variables occur very often in multiple components and the other variables occur either in only one component or very rarely.

These methods of parametrized controllers can be improved by allowing for the parameters to change during the computation. It is certainly possible to combine these methods.

Note, that the location based memory fences[LMLV11] discussed in Section 1 could be interpreted as an incomplete controller combined with a preprocessing that uses *l-mfences* to mark positions of the program and passes that information to the controller during the computation.

8 Distributed Controllers

Until now, we understood a controller as a single automaton that has instant access to actions performed on any buffer. However, in practice not all buffers are instantly accessible. In a multiple processor system, the controller has to compare buffers that are not part of the same hardware component. This means the controller is distributed and consists of multiple transition systems with access to different components that have to communicate with each other. Such transition systems are called agents. It takes longer to send a message than to process an action. The agents of the controller each have access to a different subset of components p_i. They communicate by sending messages containing an action with additional information to all components. If an agent needs to send a message, it sets a flag, such that there are no delays if no message is send.

A *message* $m = (a, inf)$ consists of an action $a \in Act$ and some additional information $inf \in Data$. The set of possible messages is $M = Act \times Data$.

We modify the transition system modelling the system $Sys_{TSO} = (\Gamma, \gamma_0, \rightarrow)$ such that it allows messages to be sent by the agents at any point:

$$Sys_{dis} := (\Gamma, \gamma_0, \rightarrow')$$

$$\rightarrow' := \rightarrow \cup \{(\gamma, m, \gamma) \mid \gamma \in \Gamma, m \in M\}$$

The transition system modelling the program $A_P = (Q, q_0, \rightarrow)$ is modified the same way:

$$A_{P,dis} := (Q, q_0, \rightarrow \cup \{(q, m, q) \mid q \in Q, m \in M\})$$

A *distributed controller* consists of a set of transition systems, we call agents, that each have access to a subset of the components. They communicate using messages in such a way that they function together as a controller.

Definition 16 *Let $I \subseteq \{1, \ldots, n\}$ a set of component indices. The TS $C_I = (\Gamma, \gamma_0, \rightarrow)$ with $\rightarrow \subseteq \Gamma \times (Act \cup M) \times \Gamma$ is an* agent *iff*

$$\gamma \xrightarrow{a} \gamma' \wedge a = (\ldots, p_i) \in Act \wedge i \notin I \Rightarrow \gamma = \gamma'$$

Let C be a controller, $C_{dis} := C_{I_1} || \ldots || C_{I_m}$ such that $I_1 \cup \ldots \cup I_m = \{1, \ldots, n\} \wedge \forall i \leq m : I_i \neq \emptyset$ is a distributed controller iff all $C_{I_j}, j \leq m$ are agents and

$$In \in L(C) \cap L(Sys_{TSO}) \cap L(A_P) \Leftrightarrow$$

$$\exists In' \in In \sqcup M^* : In' \in L(C_{dis}) \cap L(Sys_{dis}) \cap L(A_{P,dis})$$

Trivially, a distributed controller containing only one agent is just a controller. So we only examine distributed controllers with multiple components.

When constructing a distributed controller, we aim to minimize the number of messages sent when analysing a computation. We are interested in the proportion of sent messages to processed input elements.

In the best case, a distributed controller sends no message. In the next section, we will introduce the Cycle-Algorithm, that sends no messages if it processes an input without reads.

For a given controller, let m_{In} be the number of messages sent when processing the input In. In order to evaluate a distributed controller on the number of messages it creates, we find a *worst case input sequence* In, such that the ratio of sent messages to input length $\frac{m_{In}}{|In|}$ is maximal. We approximate some upper bound for the term $\frac{m_{In}}{|In|}$ for all controllers and input sequences and compare it to the approximation of the worst case input for our controller.

Theorem 6 *For every distributed controller C, there is a consistent input sequence In such that*

$$\frac{m_{In}}{|In|} \geq \frac{1}{3}$$

Proof. Let C be a distributed controller with components p_1 and $\mathbf{p_2}$ belonging to agents A and B. We examine a smallest input that produces a cycle in its trace.

$$In := w_x^{in}, r_y, \mathbf{w_y^{in}}, \mathbf{r_x}$$

As we see in Figure 9, this input produces a cycle. It is easy to see that no input of length 3 can produce a cycle in its trace.

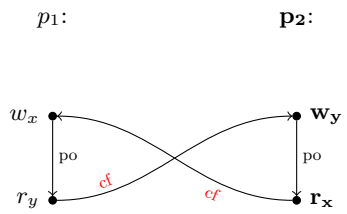

Fig. 9. The cyclic trace of $In := w_x^{in}, r_y, \mathbf{w_y^{in}}, \mathbf{r_x}$.

Now, we count how many messages a controller has to send in order to detect this cycle. The *po*-relations are detected by the components agent without the use of messages. The cf-relations however have to occurs between actions of different agents. To find the relation $\mathbf{r_x} \xrightarrow{cf} w_x$, at some point one agent must be made aware that the components have processed an incoming write w_x^{in} and a read r_x, both on the same variable x. If agent A finds the relation, then agent B must have sent a message that it has processed a read on x. If agent B finds it, then A has sent a message informing B about w_x The same holds for the other cf-relation $r_y \xrightarrow{cf} \mathbf{w_y}$. So there must be at least 2 messages sent for this input. We modify the input, so that it remains consistent and ends in an empty buffer.

$$In := w_x^{in}, r_y, \mathbf{w_y^{in}}, \mathbf{r_z}, w_x^{out}, \mathbf{w_y^{out}}$$

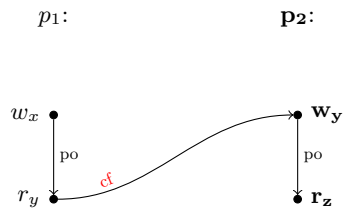

Fig. 10. The trace of the complete input $In = w_x^{in}, r_y, \mathbf{w_y^{in}}, \mathbf{r_z}, w_x^{out}, \mathbf{w_y^{out}}$ produces no cycle.

We know, that the remaining cf-relation requires one message. There is no cf-relation $\mathbf{r_z} \xrightarrow{cf} w_x$ (and thus no cycle). Obtaining this information requires a comparison of the variables z and x at one of the agents. One of the actions needs to be broadcasted. This leaves us with two necessary messages for an input of lengths six. The ratio is $\frac{1}{3}$. □

9 The Cycle-Algorithm

We now construct an algorithm for distributed controllers. We assume that every agent operates on only one component. When constructing a distributed algorithm, it is often sufficient to make this assumption. We can generalize from this scenario by merging the agents for an index-set I_j into one agent C_{I_j}. This might lead to unnecessary messages being sent in order to convey information between components that have been merged to the same agent. In order to minimize the number of messages, we need to recognize the internal messages that do not influence the behavior of other agents and substitute them with an ϵ-transition in the agent. We will see, that every message the algorithm sends must arrive in every component in order to ensure correct behaviour of the Cycle-Algorithm. This means the generalization to multiple components per agent is trivial for the Cycle-Algorithm.

When looking at the structure of the compact traces stored in the states, we see that an element in the buffer w is connected by the *po*-relation to the action that occured before it and the one after it. Those are either the buffer elements before and after w or the last read action that overtook some part of the buffer before w^{in} and the first read after. All other connections to and from w are a result of the transitivity rule. We use this knowledge to reduce the size of the state by applying the following Lemma. We use the concept of the automaton from Theorem 4 and refine it. Let r be an early read on a write w_i in a buffer of the form

$$(w_1, \ldots, w_i, w_{i+1}, \ldots, w_n)$$

We say the early read overtakes the part $(w_{i+1}, \ldots, w_n)$ of the buffer.

We have shown in Lemma 5 and 6 and Theorem 4, that an early read r can be ignored, until the write action w that the read was performed on ($w \xrightarrow{src} r$)

leaves the buffer. At this point, we can safely treat r like a read that accesses the value of w in the shared memory. The only difference is, the read may not be the last element in the *po*-relation anymore. We need to place r behind a write w', such that w' was the last element in the buffer, when r occured in the input. The situation is depicted in part (b) of Figure 11.

Since an early read r such that $w \xrightarrow{src} r$ holds only needs to be observed from the time w leaves the buffer, we will rever to to this as its execution time. We define an ordering $<$ on the actions based on their execution times: The execution time of a write action refers to the time it leaves the buffer. The execution time of a regular read is the time it occurs. The execution time of an early read r that is performed on some write w in the buffer is directly after w^{out}. It holds $w^{out} < r$. If there is an action a such that $w^{out} < a < r$ holds, than a is an early read on w that occurs before r in the input sequence.

Lemma 9 *If a trace contains a cycle C, then C contains a cf-relation $r \xrightarrow{cf} w$ such that r is in a component p different from the one w belongs to and the buffer of p is nonempty at the execution time of r. Furthermore, r and all read actions directly following r in the program order are the earliest actions in the cycle.*

Proof. Let C be a cycle in a trace and a be the earliest action in c. The cycle must contain a relation a' such that $a' \rightarrow a$ and $a < a'$. We examine the possible kinds of relations between a' and a.

Case 1($w = a' \rightarrow_{src} r = a$)**:** By definition of the *src*-relation, r reads the value of w and thus $w^{out} < r$

Case 2($w \rightarrow_{st} w'$)**:** Since w enters the memory before w' $w < w'$

Case 3($r \rightarrow_{cf} w$)**:** There is a w' such that $w' \xrightarrow{st} w \vee w' \xrightarrow{src} r \Rightarrow w \not< r$ since w' is the last write action on the variable before r and $w' < w$.

Case 4($a' \rightarrow_{po} a$)**:** If a and a' are both write or read actions then $a' < a$. If a' is a write and a a read then $a' < a$. If a' is a write and a is a read, then a can overtake a' iff a' is in the buffer at the execution time of a.

$\Rightarrow$ the earliest action is a read r.

Only the *po*- and the cf-relation can start from a read. Assuming it is a *po*-relation $r \xrightarrow{po} w$ reaching a write action w, then a', which was shown in Case 4 to be a write, has already left the buffer at the execution time of w and is not reachable any longer. This contradicts a cycle. Any number of read actions reached by program order may follow, but a write action can not be reached with program order without violating the cycle property. So eventually a write action has to be reached with a cf-relation. □

The idea behind the Cycle-Algorithm is, that in order to find a cycle in a trace, we begin by looking at a read action r overtaking some buffer content and treat every such read action r as the beginning of a potential cycle. The previous lemma shows that this is sufficient. We track the actions reachable from r by $r \xrightarrow{cf} w \rightarrow^*$ and broadcast the relevant information. We stop this process iff

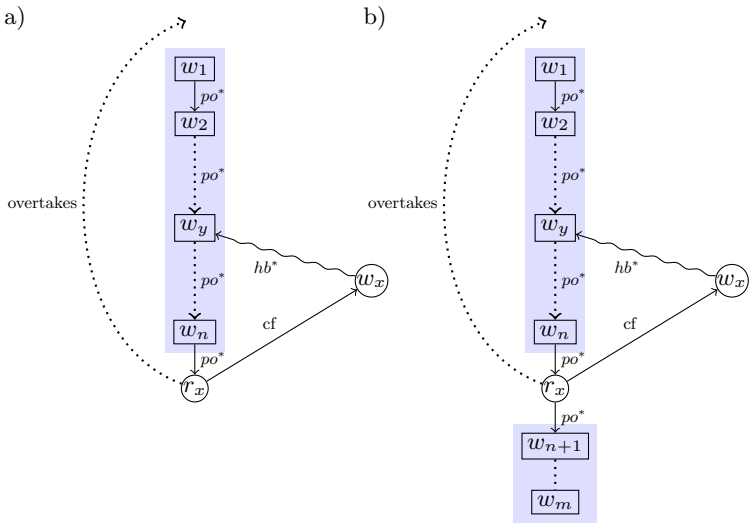

Fig. 11. The read r_x starts a cycle. The figure illustrates the buffer content at its execution time. In part (a), r_x is not early, w_n is the last action executed in the cycle and was also the last element in the buffer at the execution of r. The loop is found, when an action on y is executed while there is still a write w_y in the buffer that was overtaken by r_x. In (b), the read is early and at the time of its execution, additional writes have entered the buffer and the read only overtakes a part of the buffer.

either r is reachable from one of these actions, indicating a cycle in the trace or all the actions that were overtaken by r have left the components buffer.

We will refer to a potential cycle in the trace simply as a *cycle* and we will call a closed cycle signalling sequential inconsistency a *loop*. It is possible that a minimal cycle contains more than one conflict relation. In this case, one read r, which starts a target cycle, reaches an action w_x such that w_x reaches another cycle-starting r' with the *po*-relation. In this case we use "incorporation of cycles". We send a message, stating that all elements of the cycle started by r' are reachable from r and thus they are added to the target cycle.

We detail the algorithm as a set of `read` methods with different parameters that are called by the input elements and messages that are read by the agent. If a read r_x is processed in component p_i, the method `read`(r_x) given by Algorithm 9.1 is called. For w^{out}, `read`(w^{out}) is called (see Algorithm 9.2). However there is no such method executed when an incoming write w^{in} is processed. The write is merely added to the internal representation of the buffer content. An agent sends a message using the send command. A message contains a variable, the type of the message and a set of cycle identifiers. If an agent sends a message, then all agents, including the sender, read the message and execute the corresponding method. Each component stores each potential cycle as two sets of variables: one for the variables on which write actions have been executed $w(c)$ and one for read actions $r(c)$. This is sufficient in order to check if some action is reached with a store, conflict or read relation. A read r_x on x is reached by the cycle with the *src*-relation, if $x \in w(c)$ holds. A write w_x is reached by the *st*-relation if $x \in w(c)$ holds and it is reached by the cf-relation if $x \in r(c)$ holds. We say an action is added to a cycle c if its variable is added to $w(c)$ or $r(c)$.

Additionally, we mark for every component the state of a cycle as inactive, active or activated by some buffer element.

If $c \in \mathit{ActiveCycles}$, then the cycle c is active. This means that an executed action of this component was reached by the read r that started the cycle and thus every further action executed on this component is reached with the program order relation and added to the cycle.

If $c \in \mathit{ActivatesCycles}(w)$ holds, c it is activated by a buffer element w and the buffer elements preceding the activating element are not added to the cycle, only the ones succeeding it and any further read action. This situation occurs, when a read action overtaking a buffer content, that ends in w, is reached with the source relation.

If c is inactive, then no action of the component has been added to the cycle yet. This is illustrated in Figure 11.

If an early read on a buffer element w occurs, this is marked by a link from w to the last element $\tilde{w}$ of the buffer. This is similarly to the behaviour of the controller given in Theorem 4. The links are stored in a set $\mathit{Suspended}(w)$, so when r occurs $\mathit{Suspended}(w) := \mathit{Suspended}(w) \cup \{r\}$ is performed. After w has left the buffer, the algorithm processes the read r similar to Algorithm 9.1. The differences are as follows: we know that the read is performed on w, so we do

not check if $x \in buffer$ holds. The closing element is not the last element in the buffer, but the write $\tilde{w}$, so every use of $tail(buffer)$ is replaced by $\tilde{w}$. The read is not added to all cycles activated in the component, but only to those whose activations occur until and including $\tilde{w}$.

Since we have already shown in Theorem 4, that we can treat an early read as if it would occur later and only succeeds a part of the buffer in the program order, we know that this is correct. Any further analysis of this behaviour does not lead to new insights. In order to keep the algorithm manageable and to keep the proofs from becoming too convoluted, we will not include the handling of early reads in our presentation of the Cycle-Algorithm.

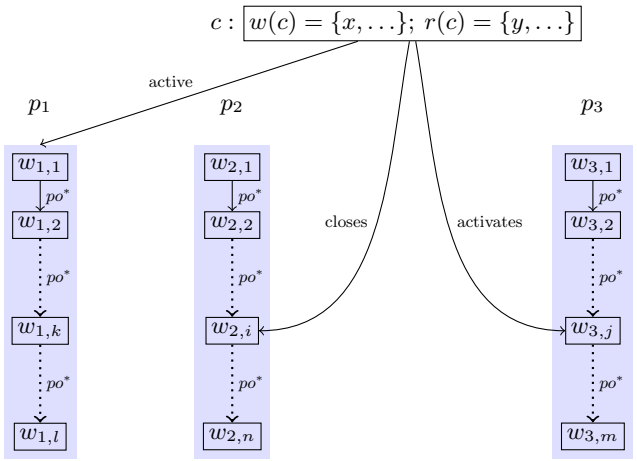

Fig. 12. The algorithms representation of a cycle c. The cycle was started by a read in component p_2 after w_i^{in}. The buffer segment $(w_{2,1}, \dots w_{2,i})$ is the end of c. It is active in p_1 and activated in p_2. The cycle contains writes on x and reads on y.

Algorithm 9.1 When a read action r_x overtakes some buffer content, the other components are alerted to the fact that a new potential cycle gets started. The last overtaken buffer element is marked as the one that closes the cycle since it will be the last action performed that precedes r_x in the program order. This means r_x can no longer be reached by the potential cycle once the closing element leaves the buffer and the potential cycle can be discarded.

If there is no write w or read action r on the same variable in an active cycle, r_x is added. It is sufficient to add only one read r_x per variable x since any w' reachable with $r_x \rightarrow_{cf} w'$ would still be detected as reachable from a later read r'_x on x when r_x is omitted. Every element reachable from r_x is also reachable from a write w on x with the store relation.

If there is a potential cycle that contains a write action on the same variable x as r_x, then r_x can be reached with the source relation from this cycle and all

Algorithm 9.1 read(r_x)

```
 1: if w_x ∉ buffer∧buffer.length≠ 0 then
 2:    if ClosesCycle(tail(buffer))= NULL then
 3:       send message(id := newId, r_x)
 4:       ClosesCycle(tail(buffer)) := id
 5:    else
 6:       if x ∉ r(ClosesCycle(tail(buffer))) then
 7:          send message(r_x, ClosesCycle(tail(buffer)))
 8:       end if
 9:    end if
10: end if
11: if C := {c ∈ ActiveCycles∪ActivatesCycles(buffer) | x ∉ r(c)∪w(c)} ≠ ∅ then
12:    send message(C, r_x)
13: end if
14: for all c ∈ Cycles do
15:    if x ∈ w(c) then
16:       if buffer.length ≠ 0 then
17:          ActivatesCycles(tail(buffer)) := ActivatesCycles(tail(buffer)) ∪ {c}
18:       else
19:          ActiveCycles := ActiveCycles ∪ {c}
20:       end if
21:    end if
22: end for
```

actions following r_x in the program order are also reachable from this cycle and so we mark the last element in the buffer as the one activating the cycle.

If there is an element in the buffer that activates a cycle or if the cycle is active, then this cycle reaches an action that precedes r_x in the *po*-relation and r_x is added.

Algorithm 9.2 When a write action w_x on a variable x leaves the buffer, a message is send if w_x closes a cycle. If there is a cycle c containing a write action on x then w_x is reachable from c with a store relation and c is activated. If w_x is reached by a cycle c (c is active) and no write action on x was added to the cycle ($x \notin w(c)$), then a message is send to inform other components that c now contains a write on x.

If there is an active cycle that was created by a read action r of this component then there is a element that closes the cycle in the buffer. This means w_x precedes r in the program order relation and the potential cycle c' created by r is reachable from every potential cycle that is active in the moment. We incorporate the cycle c' started by r in all active cycle.

Algorithm 9.3 Incorporation occurs if there is some buffer element w that was overtaken by a read r starting a cycle c', such that w is reachable by some other cycle c. Since r is reachable from w with the *po*-relation, any action in c' is reachable from the starting element of c and thus c' is added to c.

Algorithm 9.2 read(w_x^{out})

for all $c \in Cycles \backslash ActiveCycles$ **do**
  if $x \in w(c) \cup r(c)$ **then**
    $ActiveCycles := ActiveCycles \cup \{c\}$
  end if
end for
for all $w \in buffer \cup \{w_x\} : ClosesCycle(w) \neq NULL$ **do**
  for all $c' \in ActiveCycles : c'.incorporates(ClosesCycle(w)) = false$ **do**
    send inc($ClosesCycle$(w), $ActiveCycles$)
    $c'.incorporates(ClosesCycle(w)) := true$
  end for
end for
if $ActiveCycles \neq \emptyset$ **then**
  send message($ActiveCycles \cap \{c \in Cycles : x \notin w(c)\}, w_x$)
end if
if $ClosesCyle(w_x) \neq NULL$ **then**
  send CloseCycle($ClosesCycle(w_x)$)
end if
if $ActivatesCycles(w_x) \neq \emptyset$ **then**
  $ActiveCycles := ActiveCycles \cup \{ActivatesCycles(w_x)\}$
end if

Algorithm 9.3 read(inc(sources,targets))

for all $c \in targets$ **do**
  if $\exists w \in buffer : c = ClosesCycle(w)$ **then**
    if $ActiveCycles \cap sources \neq \emptyset \vee$
    $\exists w' \in buffer[w' < w \wedge ActivatesCycles(w') \cap sources \neq \emptyset] \vee$
    $\exists w_x \in buffer[w_x \leq w \wedge x \in r(sources) \cup w(sources)]$ **then**
      return "Loop found"
    end if
  end if
  for all $c' \in sources$ **do**
    $w(c) := w(c) \cup w(c')$
    $r(c) := r(c) \cup r(c')$
  end for
  if $ActiveCycles \cap sources \neq \emptyset$ **then**
    $ActiveCycles := ActiveCycles \cup targets$
  end if
  for all $w_x \in buffer$ **do**
    if $c \in ActivatesCycles(w_x)$ **then**
      $ActivatesCycles(w_x) := ActivatesCycles(w_x) \cup sources$
    end if
  end for
end for

Let a cycle c be such that it is incorporated in a cycle c'. Whenever c is active or activated, then c' has to be as well. The variables of c are added to c'. If an incorporated cycle reaches the closing element of a target cycle, then a cycle closes and we return "Loop found".

Algorithm 9.4 read(message($cycles, r_x$))

1: **for all** $c \in cycles$ **do**
2: $r(c) := r(c) \cup \{x\}$
3: **end for**
4: **if** $cycles \subseteq Cycles$ **then**
5: **if** $\exists w'_x \in buffer$ earliest action in buffer using x **then**
6: **if** $\exists w''_y \in buffer : w'_x \leq w''_y \wedge ClosesCycle(w''_y) \in cycles$ **then**
7: return "Loop found"
8: **end if**
9: **end if**
10: **else**
11: $Cycles := Cycles \cup cycles$
12: **end if**
13: **if** $\exists w'_x \in buffer$ earliest action in buffer using x **then**
14: **if** $\exists w''_y \in buffer : w'_x \leq w''_y \wedge ClosesCycle(w''_y) \neq NULL$ **then**
15: send inc($ClosesCycle(w''_y), cycles$)
16: **end if**
17: **end if**

Algorithm 9.4 A read action r_x may either start a new cycle or be added to an existing cycle. If there is a write action on the same variable w'_x in the buffer that reaches a read r starting another cycle c' with the *po*-relation ($w'_x \xrightarrow{po}{}^{*} r$), then c' is incorporated into the cycles r_x was added to.

Algorithm 9.5 read(ClosesCycles(cls))

1: $ActiveCycles := ActiveCycles \backslash cls$
2: $Cycles := Cycles \backslash cls$
3: **for all** $w \in buffer$ **do**
4: $ActivatesCycles(w) := ActivatesCycles(w) \backslash cls$
5: **end for**

Algorithm 9.5 If a cycle is closed, it becomes irrelevant and is deleted everywhere.

Algorithm 9.6 If a write action is broadcasted, it gets added to the respective cycles by every component. It is tested, whether it reaches a buffer element w

Algorithm 9.6 read(message($cycles, w_x$))

1: **for all** $c \in cycles$ **do**
2: $w(c) := w(c) \cup \{x\}$
3: **end for**
4: **if** $\exists w'_x \in buffer$ earliest action in buffer using x **then**
5: **if** $\exists w''_y \in buffer : w''_y \geq w'_x$ such that $ClosesCycle(w''_y) \in cycles$ **then**
6: return "Loop found"
7: **end if**
8: **end if**
9: **if** $\exists w'_x \in buffer$ earliest action in buffer using x **then**
10: **if** $\exists w''_y \in buffer : w''_y \geq w'_x$ such that $ClosesCycle(w''_y) \neq NULL$ **then**
11: send inc($ClosesCycle(w''_y), cycles$)
12: **end if**
13: **end if**

such that w precedes a closing element of one of the cycles that w_x belongs to. If this is the case, it completes a cycle. If there is a write w'_x on the same variable in a buffer that precedes a closing element of another cycle c', then the read that started c' is reachable in the *po*-relation ($w'_x \xrightarrow{po}{}^* r$) Every such cycle c' is incorporated into the cycles r_x was added to.

Lemma 10 *Let c be a cycle started by r_y. Let a be the next executed action of a component p_i. For any variable x, let r_x be the next read and w_x the next write executed on x.*
If c is active in component p_i, then $r_y \rightarrow^* a$ holds.
If $x \in w(c)$, then $r_y \rightarrow^* w_x$ and $r_y \rightarrow^* r_x$ holds.
If $x \in r(c)$, then $r_y \rightarrow^* w_x$ holds.
If a is a read and an element w in the buffer satisfies $c \in ActivatesCycles(w)$, then $r_y \rightarrow^* a$ holds.

Proof. Proof by induction over the construction process of a cycle.

Induction Basis: When c is created in Algorithm 9.1 line 3, it contains only r_y and is active in no component. $\Rightarrow r_y \xrightarrow{cf} w_y$

Induction Step: The algorithm adds new variables to cycles and activates them and adds them to *ActivatesCycles* in a number of ways:

Case 1: c becomes active in component p_i in Algorithm 9.1 line 19: According to line 15 holds

$$x \in w(c) \Rightarrow \exists w'_x : r_y \rightarrow^* w'_x \xrightarrow{st}{}^* w''_x \xrightarrow{src} r_x \xrightarrow{po} a$$

Case 2: c becomes active in component p_i in Algorithm 9.2 line 19: The cycle was added to ActivatesCycles either in Algorithm 9.1 line 17 or Algorithm 9.3 line 16.

Case 2a: Algorithm 9.1 line 17: Analog to Case 1. For the next action a on p_i holds $r_x \xrightarrow{po} a$ since the buffer content at the execution time of r_x has been processed.

$$x \in w(c) \Rightarrow \exists w'_x : r_y \rightarrow^* w'_x \xrightarrow{st}{}^* w''_x \xrightarrow{src} r_x \xrightarrow{po} a$$

Case 2b: Algorithm 9.3 line 16: Since this requires ActivatesCycle(w_x) to be not empty, there must be a cycle c_1 such that c_1 that was added to ActivatesCycle(w_x) by Algorithm 9.1 line 17 and then followed by a finite sequence of incorporations

$$inc(sources_1, targets_1) \dots inc(sources_k, targets_k)$$

ending in $c_k := c$ such that the following holds:

$$c_1 \in sources_1 \wedge c_k \in targets_k$$

$$\wedge \forall i \leq k-1 \; \exists c_i \in cycles(c_i \in targets_i \cap sources_{i+1})$$

The induction hypothesis holds for c_1. Let c_i be started by $r_i \Rightarrow r_1 \rightarrow^* a$. If a cycle c_i is incorporated in a cycle c_{i+1} in Algorithm 9.2 line 8, then c_{i+1} is active in the component that started cycle c_i. Furthermore, it reaches the write w which is leaving the buffer and preceding r_i in the program order. This holds since the closing element of c_i is still in the buffer ($\Rightarrow r_{i+1} \rightarrow^* r_i$). If $r_i \rightarrow^* a$ holds, $r_{i+1} \rightarrow^* a$ holds as well.
If the cycle was incorporated in Algorithm 9.4 or Algorithm 9.6, there is a write w_x on a variable $x \in w(c_{i+1}) \cup r(c_{i+1})$ that reaches r_i in the program order since the closing element of c_i does not precede w' in the buffer. Let w'_x be the next write on x. Since $x \in w(c_{i+1}) \cup r(c_{i+1})$ holds, the induction hypothesis applies for c_{i+1} and if $r_i \rightarrow^* a$ holds, then it follows

$$r_{i+1} \rightarrow^* w'_x \xrightarrow{st}{}^* w_x \xrightarrow{po}{}^* r_i \rightarrow^* a$$

Applying this argument on the sequence gives us $r_k = r_y \rightarrow^* a$.

Case 3: c becomes active in component p_i in Algorithm 9.2 line 3: We know that $x \in w(c) \cup r(c)$ holds and according to the induction hypothesis $r_y \rightarrow^* w_x$ holds, where w_x is the write processed in Algorithm 9.2. Since a is the next action on the component following w_x, it holds that $w_x \xrightarrow{po} a$ and thus $r_y \rightarrow^* a$.

Case 4: c becomes active in component p_i in Algorithm 9.3 line 12: Let c be an active cycle in sources starting with r_y. For c holds the induction hypothesis $r_y \rightarrow^* a$. According to the conditions for $inc()$ to be called (see Case 2b), for any c' starting at r' in targets holds $r' \rightarrow^* r_y$ and thus $r' \rightarrow^* a$

Case 5: A variable is added to $w(c)$ in Algorithm 9.3 line 8: Let $x \in w(c')$ be an element added to $w(c)$. If c' starts at r', then $r' \rightarrow^* w_x \wedge r' \xrightarrow{hb} r_x$ holds (since ind. hypothesis. holds for c'). According to the conditions for $inc()$ to be called (see Case 2b), $r \xrightarrow{hb} r' \rightarrow^* w_x$ resp. $r \xrightarrow{hb} r' \rightarrow^* r_x$ holds,

Case 6: A variable is added to $r(c)$ in Algorithm 9.3 line 9: Let $x \in r(c')$ be an element added to $r(c)$. If c' starts at r', then $r' \rightarrow^* w_x$ holds (since the induction hypothesis holds for c'). According to the conditions for $inc()$ to be called, $r \xrightarrow{hb} r' \rightarrow^* w_x$ holds,

Case 7: A variable is added to $r(c)$ in Algorithm 9.4 line 2: The message containing a read r_x is send by a component p_i in Algorithm 9.1 line 3 ($\curvearrowright$induction basis), line 7 or 12. If it is line 7, r_x is in the same component as the read r_y that started the cycle and thus $r_y \xrightarrow{po}{}^* r_x$ holds. If it was send in line 12, then c was active in p_i when r_x was executed or an earlier read belonging to c reaches r_x with the po-relation $\Rightarrow r_y \rightarrow^* r_x$.

If w_x is the first write on x following r_x, then $r_y \rightarrow^* r_x \xrightarrow{cf} w_x$ holds. If an earlier write w'_x is the first write on x after r_x, then $r_y \rightarrow^* r_x \xrightarrow{cf} w'_x \xrightarrow{st}{}^* w_x$ holds.

Case 8: A variable x is added to $w(c)$ in Algorithm 9.6 line 2: The cycle c is active in the component sending the message containing a write w'_x on x in Algorithm 9.2 line 13 at the execution time of $w'_x \Rightarrow r_y \rightarrow^* w'_x$ Let w_x be the next write and r_x the next read on $x \Rightarrow r_y \rightarrow^* w'_x \xrightarrow{st} w_x \wedge r_y \rightarrow^* w_x \xrightarrow{src} r_x$

□

We show that the Cycle-Algorithm correctly processes a sequence of action that is executed in the order of their relations. Such a sequence does not require incorporation of cycles.

Lemma 11 *Let $a_0 = w_y$ be a write on y and $s = r_y \xrightarrow{cf} a_0 \xrightarrow{hb} a_1 \xrightarrow{hb} \ldots \xrightarrow{hb} a_n$ a sequence starting at a read r_y on y that overtakes some buffer content ending in w_x (the closing element) such that $r_y < a_0 < a_1 < \ldots < a_n < w_x$. If after processing s, the Cycle-Algorithm has not returned "Loop found", the following statement is valid. Every cycle c such that $y \in r(c) \cup w(c)$ after the execution of r_y has the following properties: after processing any action $a_i, i \leq n$ on a variable x, if a_i is a write, $x \in w(c)$ holds and c is active in its component. If it is a read, $x \in r(c) \cup w(c)$ and either c is active in its component or there is an element $w \in buffer$ such that $c \in ActivatesCycles(w)$. There is at least one such cycle c.*

Proof. By induction over n.

Induction basis ($n = 0$)**:** Since r_y overtakes some buffer content, it either starts a new cycle c containing y in $r(c)$ (Algorithm 9.1, line 3) with the last element in the buffer w_x its closing element or adds it to a cycle that has w_x as its closing element (Algorithm 9.1, line 7)

Induction step ($n - 1 \xrightarrow{hb} n$)**:** We distinguish between the possible properties of the next relation $a_{n-1} \xrightarrow{hb} a_n$.

Case 1 ($r_x = a_{n-1} \xrightarrow{cf} w_x = a_n$)**:** When Component p reads w_x, the induction hypothesis holds for c and $x \in r(c) \cup w(c)$. Algorithm 9.2 activates c in p (line 19) and adds x to $w(c)$ if necessary (line 13).

Case 2 ($w_x = a_{n-1} \xrightarrow{st} w'_x = a_n$)**:** According to the induction hypothesis, $w(c)$ contains x at the execution time of w'_x and thus Algorithm 9.2 activates c (line 19).

Case 3 ($w_x = a_{n-1} \xrightarrow{po} a_n$)**:** According to the induction hypothesis, c is active in p after processing a_{n-1} in this component. If remains active after processing a_n and a_n is added to c if necessary (Algorithm 9.2, line 19 or Algorithm 9.1, depending on whether a_n is a write or read action).

Case 4 ($r_x = a_{n-1} \xrightarrow{po} a_n = r_z$)**:** Either c is active in p after processing a_{n-1} in this component (Algorithm 9.1 line 17) or $\exists w \in buffer(c \in ActivatesCycles(w))$. If it is active, it remains active after processing a_n and a_n is added to c if necessary by Algorithm 9.1. If there is a write $w \in buffer$ such that $c \in ActivatesCycles(w)$ after processing a_{n-1}, then either w has left the buffer and activated c (Algorithm 9.2) before the execution time of a_n, in which case the lemma holds for n, or a_n is executed before the element has left the buffer. This means w is still in the buffer and if $z \in w(c) \cup r(c)$ does not yet hold, the condition in line 11 is fulfilled and r_z is added to c. Thus, the claim holds for n.

Case 5 ($r_x = a_{n-1} \xrightarrow{po} a_n = w_z$) If c is active after processing r_x, it remains active. If not, there is a $w \in buffer : c \in ActivatesCycles(w)$. Since w_z is after r_x in the program order, the buffer content at the execution time of r_x must leave the buffer before w_z can be executed. That means w leaves the buffer and c becomes active in Algorithm 9.2 line 19. Either way, c is active add the execution time of w_z and thus w_z is added to c by Algorithm 9.2 line 19 if necessary.

Case 6 ($w_x = a_{n-1} \xrightarrow{src} r_x = a_i$)**:** According to the induction hypothesis, $x \in w(c)$ holds after a_{n-1} is executed and thus either the cycle c becomes in the component executing r_x or c is added to $ActivatesCycles(w)$ for some buffer element w.

□

We now examine sequences, where the execution times of the actions are not monotonically increasing. This may cause the algorithm to incorporate cycles.

Lemma 12 *Let $s = r_y \xrightarrow{cf} w_y = a_0 \xrightarrow{hb} a_1 \xrightarrow{hb} \ldots \xrightarrow{hb} a_n$ be a sequence starting at a read r_y that overtakes a buffer content with w_x its latest element (closing element) such that $\forall i \leq n : r_y < a_i < w_x$. If after processing s, the algorithm has not returned "Loop found", then there is a cycle c such that after processing all actions $a_i, i \leq n$ on a variable x, the following holds: If a_i is a write, then x is contained in $w(c)$ and c is active in its component. If it is a read, $x \in r(c) \cup w(c)$ holds and either c is active in its component or there is an element $w \in buffer$ such that $c \in ActivatesCycles(w)$.*

Proof. Induction over number of relations $a_{i-1} \xrightarrow{hb} a_i$ with decreasing execution times $a_{i-1} > a_i$.

Induction basis ($k = 0$): Holds according to Lemma 11.

Induction step($k \xrightarrow{hb} k+1$): Let $a_{i-1} \xrightarrow{hb} a_i$ be the last step in the sequence such that $a_{i-1} > a_i$. Then there is a variable x such that $a_{i-1} =: w_x \xrightarrow{po} r_x := a_i$ and w_x is the last write that r_x overtakes. According to the induction hypothesis, there is a c such that c fulfils the condition for $a_1 \rightarrow^* a_{i-1}$. Let c' be the cycle fulfilling the condition for $a_i \rightarrow^* a_n$ where $a_i < \ldots < a_n$ holds.
If $c.incorporates(c') = FALSE$ holds at the time w_x is processed, then c' is incorporated into c. This holds because w_x is the closing argument of c'.
If $c.incorporates(c') = TRUE$ then the cycle c' has been incorporated earlier. According to Lemma 11 the cycle c' fulfils the condition for the already executed prefix.
At the time of incorporation, c' fulfils the condition for some prefix $a_i \rightarrow^* a_j$; $i \leq j \leq n$, where a_j is the last action in the sequence $a_i \rightarrow^* a_n$ before the incorporation. Since incorporation copies all *ActiveCycles* and *ActivatesCycle* entries of the source and adds all its variables to the target cycle, we know that after incorporation, c fulfils the condition for the prefix $a_i \rightarrow^* a_j$. Since $a_j < \ldots < a_n$ holds after the execution of the remaining suffix of the sequence, we know that according to Lemma 11, c fulfils the condition for the whole sequence.

□

Theorem 7 *The algorithm is sound.*

Proof. The algorithm returns "Loop found" in Algorithm 9.3 or 9.4 or 9.6.

Assume Algorithm 9.3 returns "Loop found" in component p_i, then there is an active cycle c' started by r'_y that is incorporated in a cycle c started by r_y with a closing element w. We see in Algorithm 9.2 calling inc(), that $r_y \rightarrow^* r'_y$ holds and the condition in line 3 is fulfilled. There are two possible cases: If w is in the buffer and either c' is active in p_i or a buffer element a that precedes w activates c', then either $r'_y \rightarrow^* w$ or $r'_y \rightarrow^* a \xrightarrow{po}{}^* w$ holds (Lemma 10). By definition of the closing element, $w_x \xrightarrow{po}{}^* r_y$ holds. We have found a cycle:

$$r_y \rightarrow^* r'_y \rightarrow^* (a \xrightarrow{po}{}^*) w \xrightarrow{po}{}^* r_y$$

If $x \in w(c) \cup r(c)$ and there is a write w_x on x in the buffer not later than the closing element of c, then depending on whether $x \in w(c)$ or $x \in r(c)$, we conclude:

$$x \in w(c') \Rightarrow r_y \rightarrow^* r'_y \rightarrow^* a \xrightarrow{st}{}^* w_x \xrightarrow{po}{}^* r_y$$

or

$$x \in r(c') \Rightarrow r_y \rightarrow^* r'_y \rightarrow^* a \xrightarrow{cf} (w'_x \xrightarrow{st}{}^*) w_x \xrightarrow{po}{}^* r_y$$

Assume Algorithm 9.4 returns "Loop found". Let w_x^1 be next write action after r_x on x and $c \in ClosesCycles(w''_y) \cap cycles$ such that it was started by a read r_y. Because $c \in cycles$, c contains r_x and according to the previous lemma

$r_y \rightarrow^* w_x^1$ holds. Write w_x'' follows w_x' in the buffer and thus in the program order and w_x'' is the closing element of c

$$\Rightarrow r_y \rightarrow^* w_x^1 \xrightarrow{st}^* w_x' \xrightarrow{po}^* w_x'' \xrightarrow{po} r_y$$

Assume Alg 9.6 returns "Loop found". Analogue to the previous case regarding Algorithm 9.4, we get

$$\Rightarrow r_y \rightarrow^* w_x^1 \xrightarrow{st}^* w_x' \xrightarrow{po}^* w_x'' \xrightarrow{po} r_y$$

□

Theorem 8 *The algorithm is complete.*

Proof. Any cycle c starts at a read r overtaking a buffer content ending in a write w. Let $a_0 := r, a_n := w$ and i be the minimal value such that c has the following form.

$$c = a_0 \xrightarrow{hb} a_1 \rightarrow^* a_i \xrightarrow{po}^* a_n \xrightarrow{po} r$$

We assume that, after processing $a_1 \rightarrow^* a_{i-1}$, the algorithm has not returned "Loop found". If a_i is a read on x, then $a_{i-1} \xrightarrow{src} a_i$ and a_{i-1} is a write. According to Lemma 12, there is a cycle c such that $w(c)$ contains x when $a_0 \rightarrow^* a_{i-1}$ is executed.

If a_i is a write on x, then either $a_{i-1} \xrightarrow{st} a_i$ or $a_{i-1} \xrightarrow{cf} a_i$.

If $a_{i-1} \xrightarrow{st} a_i$, then a_{i-1} is a write and x was added to $w(c)$.

If $a_{i-1} \xrightarrow{cf} a_i$, then a_{i-1} is a read and thus either $x \in w(c)$ or $x \in r(c)$. Assume $x \in w(c)$.

In these cases, x was added to $w(c)$ either by Algorithm 9.3 or 6. Assume x was added by Algorithm 6. When the prefix of the sequence is executed, a_i is still in the buffer and either a_i or a later element closes the cycle, either way the conditions for returning "Loop found" in Algorithm 6 are fulfilled. If x was added by Algorithm 9.3, then a_i is in the buffer not later then the closing element and the third condition for "Loop found" in Algorithm 9.3 is fulfilled.

Assume now $x \in r(c)$. In this case x was either added in Algorithm 9.3 (analog to $x \in w(c)$ was added in Algorithm 9.3) or Algorithm 9.4. If it was added in Algorithm 9.4 then a_i is the earliest element using x in the buffer and a_i or a later write is the closing element of c and thus the condition for "Loop found" in Algorithm 9.4 is fulfilled. □

Now that we have proven soundness and completeness, we know the algorithm is correct.

Theorem 9 *The algorithm returns "Loop found" iff the computation has a cycle.*

It remains to prove, that the algorithm is a fastest controller. We require the some technical lemmas for this task.

Definition 17 *Let c and c' be cycles. The cycle c precedes c' iff there are buffer elements w and w' of the same component such that it holds*

$$ClosesCycle(w) = c \vee ClosesCycle(w') = c' \vee w < w'$$

Lemma 13 *Let c, c' be cycles such that c precedes c'. Any cycle incorporated by c is also incorporated by c'.*

Proof. At the creation of c, the corresponding read action is added to c'. This means c is not active or contained in an *ActivatesCycles* list and all variables in $w(c)$ and $r(c)$ are also in c'. It is easy to see that at every future time, c' contains all variables of c and if c is active in a component so is c'. If c is in a list $ActivatesCycles(w)$, then c' is either active in the component or it is in some $ActivatesCycles(w')$ such that w' is in the same component and $w' \leq w$. Hence whenever c satisfies a condition for incorporation, c' satisfies it as well. □

Lemma 14 *In any state of the algorithm, for every cycle c holds that it is not active in its own component and the buffer content up to its closing element does not activate c or contains a variable in $w(c) \cup r(c)$.*

Proof. Assume there is a reachable state that does not satisfy that condition.

If there is a variable of the buffer content in $w(c) \cup r(c)$ that was not added by incorporation, then the condition for "Loop found" in either algorithm 4 or 6 would have been satisfied.

If c is active in the component or activated before its closing element and this did not happen through incorporation, then at some earlier point there was an element of the buffer content up to the closing element in $w(c) \cup r(c)$. If one of those conditions were satisfied by incorporation of some c', then Algorithm 3 would have returned "Loop found" at the time of incorporation. □

Theorem 10 *The Cycle-Algorithm is a fastest controller.*

Proof. Idea: We prove that the controller does not return "Loop found" too early by showing that every input the controller processes without returning "Loop found" can be continued to a final state with empty buffers without a cycle.

Assume the Cycle-Algorithm is not a fastest controller, then there is an input In such that In cannot be continued to a final state and the algorithm does not return "Loop found" when processing In. Let C be the configuration after processing In, containing a set of cycles and the buffer content. We perform an induction over the number of cycles $k := |cycles|$:

Induction basis: $(k = 0)$ If $k = 0$ holds, then the buffers can be emptied by a sequence of writes leaving the buffer without the algorithm finding a loop, since this requires a cycle and they can only be created by a read action.

Induction Step: $(k > 0)$ Assume every cycle incorporates another cycle. Since incorporations are transitive and the number of cycles is finite, there must be a cyclic sequence of incorporations. So at some point there was a function

call $inc(sources, targets)$ such that $sources \cap targets \neq \emptyset$. The algorithm would have returned "Loop found" at this time. This is a contradiction.
There is at least one cycle c' such that c' is not incorporated by another cycle. According to the previous lemma, there is also a cycle c such that c is not incorporated by another cycle and c is not preceded by another cycle. Let $w_1, \dots w_k$ be the buffer content up to the closing element w_k of c. Since c is not preceded by another cycle, its closing element is the first closing element in the buffer. Since it is not incorporated by another cycle, the following holds: There is no cycle active in its component and there is no buffer element w preceding the closing element such that w activates a cycle or has a variable occurring in another cycle.
Using Lemma 14, this means we can continue the input with $w_1^{out}, \dots w_k^{out}$ without entering an error state. This does not create new cycles and it deletes c so the induction hypothesis holds for the new state.

After In has been processed, we analyse the potential cycles in the algorithm and determine a continuation of the input such that the trace contains no loop. This is a contradiction to the assumption, that In cannot lead to a complete state of a controller. □

The Finite Cycle-Algorithm

The Cycle-Algorithm can easily be implemented as a single finite automaton without message broadcasting. Every component stores its own information about cycle activation. When a variable is added to a cycle, the information is immediately broadcasted, which means that the relevant information is stored consistently in all components. Note that for a cycle c, whenever a variable occurs in $w(c)$, it does not matter if it also occurs in $r(c)$, since the algorithm never tests for $r(c)$, only for $w(c) \cup r(c)$. It is therefore sufficient to store one set of variables by using one bit to indicate whether a variable is in $w(c)$ or $r(c)$. The algorithm might produce multiple activating elements of the same cycle in the same component. It is sufficient to store only the earliest one for each component, since any succeeding activations obviously do not influence the behaviour. For the suspended reads of a cycle, we store the positions in the components buffer.

Theorem 11 *Given n program components, a finite buffer size B and a finite set of variables $|V|$, it is possible to implement the algorithm as a finite automaton with the following number of states.*

$$|V|^{n \cdot B} \cdot n \cdot B \cdot 3^{|V|} \cdot (B+2)^{n-1} \cdot 2^B$$

Proof. The buffers can store $B \cdot n$ many writes. For each write we only need to store its variable. This amounts to $|V|^{n \cdot B}$ possible buffer configurations.

Every position in the buffer can be the closing element of one cycle. There are $B \cdot n$ possible cycles. For each cycle c, a variable can have three states: it can either be in $w(c)$ or $r(c)$ or not in the cycle. In every component except

its starting component, a cycle can be either active or inactive or activated by a buffer element. There are $(B+2)^{n-1}$ possible activations for a cycle and 2^B many possibilities to mark a set of buffer elements as suspended reads. □

Note that the term of Theorem 11 is not a least upper bound for the size of the Cycle-Algorithm. It contains configurations not reachable by the algorithm and also contains configurations that exhibit the same behaviour as each other.

This upper bound is exponential in every parameter, however the number of components in a system n is usually rather small. In Section 7, we have shown how to use preprocessing to reduce the number of variables the controller processes and other methods to reduce the complexity of a controller.

Furthermore, if an implementation of a controller can be constructed that has access to the buffer, it does not need to store the buffer configuration internally and its complexity is reduced by a factor of $|V|^{n \cdot B}$.

10 Time Complexity of Controllers

Our motivation for introducing controllers was to increase the efficiency of program executions by removing the delays created by fences. In order to prevent inconsistent computations, the controller has to interact with the system in such a way that an action is send to the controller before it is executed. The controller then either allows the action to be executed or it delays it, if it would cause an inconsistency. We say the system *requests* an action. It has to be a fastest controller in order to be able to perform that task. The processing of the input actions by the controller adds a delay to every action in the input sequence. In contrast, the fence actions only add delays to some some input elements. In order to increase efficiency by using controllers instead of fences, a controller needs to be optimized with respect processing time.

Note that a controller has to perform two tasks in order to process an input action: it has to decide if the action leads to an inconsistent computation and it has to perform an update on its internal state.

When the system requests to execute an action, the controller checks if it would cause the computation to become inconsistent. In this case, the controller denies the execution and it does not need to update its state. If the controller allows the execution, the system can already start to execute the action while the controller computes its state update.

To examine the Cycle-Algorithm(see Section 9), we recall, that a loop can only be found if the variable of the input element (which has to be a read or outgoing write) occurs in the ends of certain cycles. So the Cycle-Algorithm needs only to check some buffers for a given variable when a read or an outgoing write is requested. The state update however is more complex.

If the controller has not finished its state update by the time the next action is requested to be performed, an additional delay is caused. However, in some situations, it is possible to decide, whether an action is allowed before the state update is completed. A simple example is the Cycle-Algorithm. An action w^{in}

causes no test for loops in the algorithm and thus it is always allowed by the controller independent of its state.

Another method to enable an earlier start of the processing is the following. We identify those parts of the internal storage of the controller that are not affected by the state update before beginning the update process. When a new action is requested by the system, the controller can start those operations that only use the marked part of the internal storage of the controller.

When an action is requested of a distributed controller, the controller might need to send messages. In order to avoid such a delay, the distributed controller has to be constructed such that the agents send messages with information that might be requested in advance. In the Cycle-Algorithm, the starting component of a cycle c sends out messages containing the variables in the end of the cycle. Another agent can successively test if an action creates a loop in c without sending the action to the starting component of c. These messages do not need to delay a computation, they could be performed in the background since they do not effect the functionality of the controller.

Further techniques that can increase efficiency were introduced in Section 7.

11 Conclusion and Outlook

In this thesis, we have given a formal representation of the interactions of the write buffer in a total store ordering system. Based on the definition of TSO computations using rewrites, we have developed a technique to construct a trace of a computation by sequentially reading the buffer input. A controller was defined that observes the memory actions as they access the buffer and the shared memory so that consistency is ensured.

In the following some results of this thesis are summarized. It is not possible to construct such a controller for a TSO system as a finite automaton if the number of variables or the maximal buffer size is unbounded. However, there exists a finite automaton that is a controller for a TSO model where the number of variables and the maximal size of the write buffer is bounded. Such a controller can construct a compact version of a trace and check the reachability of the actions in order to detect cycles. The size of a finite controller with a minimal number of states is exponential in the number of variables and at least polynomial in the size of the buffer.

Any controller that observes inconsistencies too late can be transformed into a fastest controller that detects every inconsistency at the first input action it occurs.

We defined the Cycle-Algorithm as an efficient distributed controller that stores potential cycles starting at some read that overtakes some buffer content.

Any implementation of a controller needs to be very time and space efficient in order to justify its usage. We have given an overview of multiple ways to increase the efficiency of a controller. A promising approach is to perform a static analysis on the program and use it to modify the program or provide additional information to the controller.

It would be optimal to test these algorithms in a real hardware environment, but of course this is beyond the scope of this thesis. A software simulation might be used to perform benchmark tests on common mutex algorithms. This might provide a notion of the performance advantage of an implementation of a controller against the use of fences.

Future research should examine how the concept of a controller for the buffers of the shared memory could be utilized for other relaxed memory models such as TSOR or PSO.

Index

References

[AA93] S. V. Adve and J. K. Aggarwal. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, June 1993.

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. *SIGPLAN Not.*, 45(1):7–18, January 2010.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[AM11] Jade Alglave and Luc Maranget. Stability in weak memory models. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2011.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 107–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[BMM11] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 428–440, Berlin, Heidelberg, 2011. Springer-Verlag.

[Dij65] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[DPN93] David L Dill, Seungjoon Park, and Andreas G. Nowatzyk. Formal specification of abstract memory models. In *Proceedings of the 1993 symposium on Research on integrated systems*, pages 38–52, Cambridge, MA, USA, 1993. MIT Press.

[Int07] Intel. Intel 64 architecture memory ordering white paper, 2007.

[LMLV11] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 75–84, New York, NY, USA, 2011. ACM.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag.

[PD95] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 34–41, New York, NY, USA, 1995. ACM.

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.