Rely/Guarantee Reasoning for Multicopy Atomic Weak Memory Models

Nicholas Coughlin[1,2], Kirsten Winter[1,2], and Graeme Smith[1,2](✉)

[1] Defence Science and Technology Group, Brisbane, Australia
[2] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Australia
smith@itee.uq.edu.au

Abstract. Rely/guarantee reasoning provides a compositional approach to reasoning about concurrent programs. However, such reasoning traditionally assumes a sequentially consistent memory model and hence is unsound on modern hardware in the presence of data races. In this paper, we present a rely/guarantee-based approach for *multicopy atomic* weak memory models, i.e., where a thread's stores become observable to all other threads at the same time. Such memory models include those of the widely used x86-TSO and ARMv8 processor architectures, as well as the open-source RISC-V architecture. In this context, an operational semantics can be based on thread-local instruction reordering. We exploit this to provide an efficient compositional proof technique in which weak memory behaviour can be shown to preserve rely/guarantee reasoning on a sequentially consistent memory model. To achieve this, we introduce a side-condition, *reordering interference freedom*, reducing the complexity of weak memory to checks over pairs of reorderable instructions. To enable practical application, we also define a dataflow analysis capable of identifying a thread's reorderable instructions. All aspects of our approach have been encoded and proved sound in Isabelle/HOL.

1 Introduction

Reasoning about concurrent programs with interference over shared resources is a complex task. The interleaving of all thread behaviours leads to an exponential explosion in observable behaviour. Rely/guarantee reasoning [11] is one approach to reduce the complexity of the verification task. It enables reasoning about one thread at a time by considering an abstraction of the thread's environment given as a *rely condition* on shared resources. This abstraction is justified by proving that all other threads in the environment guarantee the assumed rely condition. The approach limits the interference between threads to the effects of the rely condition (specified as a relation over states) on a thread's state.

Xu et al. [34] show how rely/guarantee reasoning can be used to allow reasoning over individual threads in a concurrent program using Hoare logic [10]. We introduce a similar approach in [33] to allow thread-local reasoning, in the context of information flow security, using weakest precondition calculation [7].

M. Huisman et al. (Eds.): FM 2021, LNCS 13047, pp. 292–310, 2021.
https://doi.org/10.1007/978-3-030-90870-6_16

These approaches work equally well for concurrent programs executed on weak memory models under the implicit assumption that the code is data-race free. This is a reasonable assumption given that most programmers avoid data races due to them leading to unexpected behaviour when the code's execution is optimised under the weak memory model of the compiler [4,12] or underlying hardware [3,6,27]. However, data races may be introduced inadvertently by programmers, or programmers may introduce data races for efficiency reasons, as seen in non-blocking algorithms [19]. These algorithms appear regularly in the low-level code of operating systems, e.g., seqlock [5] is used routinely in the Linux kernel, and software libraries, e.g., the Michael-Scott queue [18] is used as the basis for Java's ConcurrentLinkedQueue in java.util.concurrent.

There are a number of approaches for verifying concurrent code under weak memory models [1,2,9,14–16,30,31], which are centred around relations between instructions in multiple threads, thereby precluding the benefits of thread-local reasoning. Notable amongst these is the work by Abdulla et al. [1,2] which aims at automated tool support via stateless model checking and is based on the axiomatic semantic model of [3]. Instead of thread-local reasoning the approaches deal with execution graphs which include not only the interleaving behaviour of concurrent threads but also "parallelisation" of sequential code resulting from weak memory behaviour. Techniques to combat the resulting state-space explosion and improve scalability include elaborate solutions to dynamic partial order reduction, context bounds for a bug-finding technique [1] and (for a sound approach) coarsening the semantic model of execution graphs through reads-from equivalences [2].

Closer to our approach is the proof system for concurrent programs under the C11 memory model developed by Lahav et al. [16]. This proof system is based on the notion of Owicki-Gries reasoning with interference assertions between each line of code to capture potential interleavings. However, to achieve a thread-local approach the authors present their logic in a "rely/guarantee style" in which interference assertions are collected in "rely sets" whose stability needs to be guaranteed by the current thread. This leads to a fine-grained consideration of interference between threads whereas in standard rely/guarantee reasoning the interference is abstracted into a rely condition which summarises the effects of the environment. Moreover, similarly to [1,2] the semantic model is based on (an abstraction of) the axiomatic model in [3] so that the interference between threads includes additionally weak memory effects thereby further complicating the analysis over each instruction. A somewhat-related approach to capture assertions on thread interference is presented in [15] which computes the reads-from relation between threads which is then taken into a account by the thread-local static analyser.

Approaches that propose a purely thread-local analysis for concurrent code under weak memory models include the work by Ridge [24] and Suzanne et al. [29]. Both capture the weak memory model of x86-TSO [26] by modelling the concept of store buffers. This limits their applicability to that of this relatively simple memory model and prohibits adaption to weaker memory models.

In contrast, this paper defines a proof system for rely/guarantee reasoning that is parameterised by the weak memory model under consideration. We restrict our focus to those memory models that are *multicopy atomic*, i.e., where a thread's stores become observable to all other threads at the same time. This includes the memory models of x86-TSO [26], ARMv8 [23] and RISC-V [32] processor architectures, but not POWER [25], older ARM [8] processors nor C11 [4]. As shown by Colvin and Smith [6], multicopy atomic memory models can be captured in terms of instruction reordering. That is, they can be characterised by a reordering relation over pairs of instructions in a thread's code, indicating when two instructions may execute out-of-order[1]. This has been validated against the same sets of litmus test used to validate the widely accepted weak memory semantics of Alglave et al. [3], establishing a high degree of confidence in the correctness of the reordering semantics.

Consequently, the implications of weak memory can be captured thread-locally, enabling compositional reasoning. However, thread-local reasoning under such a semantics is non-trivial. Instruction reordering introduces interference within a single thread, similar to the effects of interference between concurrent threads and equally hard to reason about. For instance, a thread with n reorderable instructions may have $n!$ behaviours due to possible reordering. To tackle such complexity, we exploit the fact that many of these instructions will not influence the behaviour of others. We reduce the verification burden to a standard rely/guarantee judgement [34], over a sequentially consistent memory model, and a consideration of the pair-wise interference between reorderable instruction in a thread, totalling $n(n-1)/2$ pairs given n reorderable instructions. The resulting proof technique has been automated and shown to be sound on both a simple while language and an abstraction of ARMv8 assembly code using Isabelle/HOL [21] (see https://bitbucket.org/wmmif/wmm-rg).

We begin the paper in Sect. 2 with a formalisation of a basic proof system for rely/guarantee reasoning introduced in [34]. In Sect. 3, we abstractly introduce reordering semantics for weak memory models and our notion of *reordering interference freedom* which suffices to account for the effects of the weak memory model. Moreover, we discuss the practical implications of our approach. In Sect. 4 we present the instantiation of the approach with a simple language and demonstrate reasoning with an example. Our work on a more elaborate instantiation of ARMv8 assembly and the verification of a work-stealing deque developed for ARM [17] is available in the Isabelle/HOL theories. We conclude in Sect. 5.

2 Preliminaries

The language for our framework is purposefully kept abstract so that it can be instantiated for different programming languages. It consists of individual

[1] For non-multicopy atomic processors such as POWER and older versions of ARM, the semantics of Colvin and Smith additionally requires a *storage subsystem* to capture each thread's view of the global memory, resulting in a more complex semantics that cannot be fully captured by reordering.

instructions α, whose executions are atomic, and *commands* (or programs) c which are composed of instructions using sequential composition, nondeterministic choice, iteration, and parallel composition. Commands also include the empty program ϵ denoting termination.

$$c ::= \epsilon \mid \alpha \mid c_1 \,;\, c_2 \mid c_1 \sqcap c_2 \mid c^* \mid c_1 \parallel c_2$$

Note that conditional instructions (like if-then-else and loops) and their evaluation are modelled via silent steps making a nondeterministic choice during the execution of a program (see Sect. 4).

A *configuration* of a program is a pair (c, σ), consisting of a command c to be executed and state σ (a mapping from variables to values) in which it executes. The behaviour of a component, or thread, in a concurrent program can be described via steps the program, including its environment, can perform during execution, each modelled as a relation between the configurations before and after the step. A *program step*, denoted as $(c, \sigma) \xrightarrow{ps} (c', \sigma')$, describes a single step of the component itself and changes the command (i.e., the remainder of the program). A program step may be an *action step* $(c, \sigma) \xrightarrow{as} (c', \sigma')$ which performs an instruction that also changes the state, or a *silent step*, $(c, \sigma) \rightsquigarrow (c', \sigma)$ which does not execute an instruction but makes a choice and thus changes the command only. Hence $\xrightarrow{ps} = (\xrightarrow{as} \cup \rightsquigarrow)$. An *environment step*, $(c, \sigma) \xrightarrow{es} (c, \sigma')$, describes a step of the environment (performed by any of the other concurrent components); it may alter the state but not the remainder of the program (of the component).

Program *execution* is defined via a small-step semantics over the command.

$$\begin{array}{l} \alpha \mapsto_\alpha \epsilon \\ c_1 \,;\, c_2 \mapsto_\alpha c_1' \,;\, c_2 \text{ if } c_1 \mapsto_\alpha c_1' \\ c_1 \parallel c_2 \mapsto_\alpha c_1' \parallel c_2 \text{ if } c_1 \mapsto_\alpha c_1' \text{ or } c_1 \parallel c_2 \mapsto_\alpha c_1 \parallel c_2' \text{ if } c_2 \mapsto_\alpha c_2' \end{array} \qquad (1)$$

The *semantics* of program steps is based on the evaluation of instructions. Each atomic instruction α has a relation over (pre- and post-) states $beh(\alpha)$, formalising its execution behaviour. A program step $(c, \sigma) \xrightarrow{as} (c', \sigma')$ requires an execution $c \mapsto_\alpha c'$ to occur such that the state is updated according to the executed instruction α, i.e.,

$$(c, \sigma) \xrightarrow{as} (c', \sigma') \Leftrightarrow \exists\, \alpha.\ c \mapsto_\alpha c' \wedge (\sigma, \sigma') \in beh(\alpha)\,. \qquad (2)$$

2.1 Rely/Guarantee Reasoning

A proof system for rely/guarantee reasoning in a Hoare logic style has been defined in [34]. Our approach largely follows its definitions, but includes a customisable verification condition, *vc*, with each instruction. This verification condition serves to capture the state an instruction must execute under to enforce properties such as the component's guarantee and can be considered part of the program's specification. For example, in an information flow security analysis (cf. [33]), it can be used to check that the value assigned to a publicly accessible variable is not classified. We define a Hoare triple as follows. For simplicity of presentation, we treat predicates as sets of states.

$$P\{\alpha\}Q \ \widehat{=}\ P \subseteq vc(\alpha) \cap \{\sigma \mid \forall \sigma'. (\sigma, \sigma') \in beh(\alpha) \Rightarrow \sigma' \in Q\} \tag{3}$$

The rely and guarantee conditions of a thread, denoted $\mathcal{R}$ and $\mathcal{G}$ respectively, are relations over (pre- and post-) states. The rely condition captures allowable environments steps and the guarantee constrains all program steps. A rely/guarantee pair ($\mathcal{R}$, $\mathcal{G}$) is wellformed when the rely condition is reflexive and transitive, and the guarantee condition is reflexive.

Given that $\mathcal{R}$ is transitive, stability of a predicate P under rely condition $\mathcal{R}$ is defined such that $\mathcal{R}$ maintains P.

$$stable_{\mathcal{R}}(P) \ \widehat{=}\ P \subseteq \{\sigma \mid \forall \sigma'. (\sigma, \sigma') \in \mathcal{R} \Rightarrow \sigma' \in P\} \tag{4}$$

The condition under which an instruction guarantees $\mathcal{G}$ is defined as

$$guar(\alpha, \mathcal{G}) \ \widehat{=}\ vc(\alpha) \subseteq \{\sigma \mid \forall \sigma'. (\sigma, \sigma') \in beh(\alpha) \Rightarrow (\sigma, \sigma') \in \mathcal{G}\}. \tag{5}$$

These ingredients allow us to introduce a rely/guarantee judgement. We do this on three levels: the instruction level $\vdash_{\mathsf{a}}$, the component level $\vdash_{\mathsf{c}}$, and the global level $\vdash$. On the instruction level the judgement requires that the pre- and post-condition are stable under $\mathcal{R}$. This ensures that these conditions, and hence the Hoare triple, hold despite any environmental interference. Additionally, the judgement requires that the instruction satisfies the guarantee $\mathcal{G}$.

$$\mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\alpha\}Q \ \widehat{=}\ stable_{\mathcal{R}}(P) \wedge stable_{\mathcal{R}}(Q) \wedge guar(\alpha, \mathcal{G}) \wedge P\{\alpha\}Q \tag{6}$$

A rely/guarantee proof system on the component and global levels follows straightforwardly and is given in Fig. 1. At the component level, note the necessity for the invariant of the [Iteration] rule to be stable (such that it continues to hold amid environmental interference). At the global level, the rule for parallel composition [Par] includes a compatibility check ensuring that the guarantee for each component implies the rely conditions of the other component. A standard [Conseq] rule over global satisfiability is supported by the proof system, but omitted in Fig. 1.

Such rules are standard to rely/guarantee reasoning [34]. Our modification can be seen in can be seen in [Comp], in which global satisfiability is deduced from component satisfiability $\vdash_{\mathsf{c}}$ plus an additional check on *reordering interference freedom*, $rif(\mathcal{R}, \mathcal{G}, c)$, which we introduce in Sect. 3.2. As a consequence, component-based reasoning in this proof system is based on standard rely/guarantee reasoning which can be conducted independently from the interference check.

Moreover, the proof system supports a notion of *auxiliary* variables, common to rely/guarantee reasoning [28,34]. These variables increase the expressiveness of the specification ($\mathcal{R}$, $\mathcal{G}$, P and Q) by representing properties of intermediate execution states. Auxiliary variables cannot influence program execution, as they are abstract, and their modification must be coupled with an instruction such that they are considered atomic.

$$[\mathsf{Atom}]\frac{\mathcal{R},\mathcal{G}\vdash_{\mathsf{a}} P\{\alpha\}Q}{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{\alpha\}Q} \qquad [\mathsf{Seq}]\frac{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c_1\}M \quad \mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} M\{c_2\}Q}{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c_1\,;c_2\}Q}$$

$$[\mathsf{Choice}]\frac{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c_1\}Q \quad \mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c_2\}Q}{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c_1\sqcap c_2\}Q} \qquad [\mathsf{Iteration}]\frac{\mathit{stable}_{\mathcal{R}}(P) \quad \mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c\}P}{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c^*\}P}$$

$$[\mathsf{Conseq}]\frac{P'\subseteq P \quad \mathcal{R}'\subseteq\mathcal{R} \quad \mathcal{G}\subseteq\mathcal{G}' \quad Q\subseteq Q' \qquad \mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c\}Q}{\mathcal{R}',\mathcal{G}'\vdash_{\mathsf{c}} P'\{c\}Q'}$$

$$[\mathsf{Comp}]\frac{\mathcal{R},\mathcal{G}\vdash_{\mathsf{c}} P\{c\}Q \quad \mathit{rif}(\mathcal{R},\mathcal{G},c)}{\mathcal{R},\mathcal{G}\vdash P\{c\}Q}$$

$$[\mathsf{Par}]\frac{\mathcal{R}_1,\mathcal{G}_1\vdash P_1\{c_1\}Q_1 \quad \mathcal{R}_2,\mathcal{G}_2\vdash P_2\{c_2\}Q_2 \quad \mathcal{G}_2\subseteq\mathcal{R}_1 \quad \mathcal{G}_1\subseteq\mathcal{R}_2}{\mathcal{R}_1\cap\mathcal{R}_2,\mathcal{G}_1\cup\mathcal{G}_2\vdash P_1\cap P_2\{c_1\parallel c_2\}Q_1\cap Q_2}$$

Fig. 1. Proof rules for rely/guarantee reasoning

3 Weak Memory Models

Weak memory models are commonly defined to maintain sequentially consistent behaviour given the absence of data races, thereby greatly simplifying reasoning for the majority of programs. However, as we are interested in the analysis of racy concurrent code, it is necessary to reason on a semantics that fully captures the behaviours these models may introduce.

Colvin and Smith [6] show that weak memory behaviour for multicopy atomic processors such as x86-TSO, ARMv8 and RISC-V can be captured in terms of instruction reordering. A memory model, in these cases, is characterised by a reordering relation over pairs of instructions indicating whether the two instructions can execute out-of-order when they appear in a component's code. This complicates reasoning significantly. For example, one needs to determine whether an instruction α that is reordered to execute earlier in a program can invalidate verification conditions that are satisfiable under normal executions (following the program order without reordering). In that sense, we are facing not only interference between concurrent components (which can be visualised as *horizontal* interference) but also interference between the instructions within one component (which can be pictured as *vertical* interference).

3.1 Reordering Semantics

The reordering relation, $\hookleftarrow$, of a component is syntactically derivable based on the rules of the specific memory model (see Sect. 3.3). In ARMv8, for example, two instructions which do not access (write or read) a common variable are deemed semantically independent and can change their execution order. Moreover, weak memory models support various memory barriers that prevent particular forms of reordering. For example, a full fence prevents all reordering, while a control fence prevents speculative execution (for a complete definition refer to [6]).

Matters are complicated by the concept of *forwarding*, where an instruction that reads from a variable written in an earlier instruction might replace the reading access with the written value, hence shedding the dependence to the variable in common. This allows it to execute earlier, anticipating the write before it happens. For example x := z; y := x, where z cannot be modified by another component, can execute as y := z; x := z. We denote the instruction α with the value written in an earlier instruction β forwarded to it as $\alpha_{\langle\beta\rangle}$. Note that $\alpha_{\langle\beta\rangle} = \alpha$ whenever β does not write to a variable that is read by α.

Forwarding can span a series of instructions and can continue arbitrarily, with later instructions allowed to replace variables introduced by earlier forwarding modifications. The term $\gamma \prec c \prec \alpha$ denotes reordering of the instruction α prior to the command c, with the cumulative forwarding effects producing γ. $\alpha_{\langle\!\langle c \rangle\!\rangle}$ denotes the cumulative forwarding effects of the instructions in command c on α. We define both terms recursively over c.

$$\begin{gathered}\alpha_{\langle\beta\rangle} \prec \beta \prec \alpha \mathrel{\widehat{=}} \beta \hookleftarrow \alpha_{\langle\beta\rangle} \\ \alpha_{\langle\!\langle c_1\,;\,c_2 \rangle\!\rangle} \prec c_1\,;\; c_2 \prec \alpha \mathrel{\widehat{=}} \alpha_{\langle\!\langle c_1\,;\,c_2 \rangle\!\rangle} \prec c_1 \prec \alpha_{\langle\!\langle c_2 \rangle\!\rangle} \;\wedge\; \alpha_{\langle\!\langle c_2 \rangle\!\rangle} \prec c_2 \prec \alpha\end{gathered} \tag{7}$$

To capture the effects of reordering, we extend the definition of executions (1) with an extra rule that captures out-of-order executions: A step can execute an instruction whose original form occurs later in the program if reordering and forwarding can bring it (in its new form γ) to the beginning of the program.

$$c_1\,;\; c_2 \mapsto_\gamma c_1\,;\; c_2' \text{ if } \gamma \prec c_1 \prec \alpha \wedge c_2 \mapsto_\alpha c_2' \tag{8}$$

3.2 Reordering Interference Freedom

Our aim is to eliminate the implications of this reordering behaviour and, therefore, enable standard rely/guarantee reasoning despite a weak memory context. To achieve this, we note that a valid reordering transformation will preserve the thread-local semantics and, hence, will only invalidate reasoning when observed by the environment. Such interactions are captured either as invalidation of the component's guarantee $\mathcal{G}$ or new environment behaviours, as allowed by its rely condition $\mathcal{R}$. Consequently, reorderings may be considered benign if the modified variables are not related by $\mathcal{G}$ or $\mathcal{R}$.

We capture such benign reorderings via reordering interference freedom. Two instructions are said to be *reordering interference free* (*rif*) if we can show that reasoning over the instructions in their original (program) order is sufficiently strong to also include reasoning over their reordered behaviour. Consider the program text $\beta\,;\, \alpha$, where α can be forwarded and executed before β, resulting in an execution equivalent to $\alpha_{\langle\beta\rangle}\,;\, \beta$. Reordering interference freedom between α and β under given rely/guarantee conditions is then formalised as follows.

$$\begin{aligned} \mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha) \;\mathrel{\widehat{=}}\; & \forall\, P, Q, M.\; \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\beta\}M \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} M\{\alpha\}Q \\ & \Rightarrow \exists\, M'.\; \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\alpha_{\langle\beta\rangle}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} M'\{\beta\}Q \end{aligned} \tag{9}$$

Importantly, $\mathit{rif}_{\mathsf{a}}$ is defined independently of the pre- and post-states of the given instructions, as can be seen by the universal quantification over P, M and Q in (9). This independence allows for the establishment of $\mathit{rif}_{\mathsf{a}}$ across a program via consideration of only pairs of reorderable instructions, rather than that of all execution traces under which they may be reordered. Such an approach dramatically reduces the complexity of reasoning in the presence of reordering, from one of $n!$ transformed programs for n reorderable instructions to $n(n-1)/2$ pairs. This can be seen in the case where all n instruction reorder, producing $n!$ permutations, whilst pairs need only consider the 2-combinations of n, equivalent to $n(n-1)/2$.

The definition of $\mathit{rif}_{\mathsf{a}}$ extends inductively over commands c with which α can reorder. Command c is *reordering interference free* from α under $\mathcal{R}$ and $\mathcal{G}$, if the reordering of α over each instructions of c is interference free, including those variants of α produced by forwarding.

$$\begin{aligned} \mathit{rif}_{\mathsf{c}}(\mathcal{R},\mathcal{G},\beta,\alpha) &= \mathit{rif}_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha) \\ \mathit{rif}_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_1\ ;\ c_2,\alpha) &= \mathit{rif}_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_1,\alpha_{\langle\!\langle c_2\rangle\!\rangle}) \wedge \mathit{rif}_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_2,\alpha) \end{aligned} \tag{10}$$

From the definition of executions including reordering behaviour given in (8) we have $c \mapsto_{\alpha_{\langle\!\langle r\rangle\!\rangle}} c' \Rightarrow r\ ;\ \alpha \in \mathit{prefix}(c) \wedge \alpha_{\langle\!\langle r\rangle\!\rangle} \prec r \prec \alpha$, where $\mathit{prefix}(c)$ refers to the set of prefixes of c. Program c is *reordering interference free* if and only if all possible reorderings of its instructions over the respective prefixes are reordering interference free.

$$\mathit{rif}(\mathcal{R},\mathcal{G},c)\ \mathrel{\widehat{=}}\ \forall\, \alpha, r, c'.\ c \mapsto_{\alpha_{\langle\!\langle r\rangle\!\rangle}} c' \Rightarrow \mathit{rif}_{\mathsf{c}}(\mathcal{R},\mathcal{G},r,\alpha) \wedge \mathit{rif}(\mathcal{R},\mathcal{G},c') \tag{11}$$

As can be seen from the definitions, checking $\mathit{rif}(\mathcal{R},\mathcal{G},c)$ amounts to checking $\mathit{rif}_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha)$ for all pairs of instructions β and α that can reorder in c, including those pairs for which α is a new instruction generated through forwarding. Therefore one can reason about a component's code as follows.

1. Compute all pairs of reorderable instructions, i.e., each pair of instructions (β,α) such that there exists an execution trace where α reorders before β according to the memory model under consideration.
2. Demonstrate reordering interference freedom for as many of these pairs as possible (using $\mathit{rif}_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha)$).
3. If $\mathit{rif}_{\mathsf{a}}$ cannot be shown for some pairs, introduce memory barriers to prevent their reordering or modify the verification problem such that their reordering can be considered benign.
4. Verify the component in isolation, using standard rely/guarantee reasoning with an assumed sequentially consistent memory model.

We detail steps 1–3 in the following sections and assume the use of any standard rely/guarantee reasoning approach for step 4.

3.3 Computing All Reorderable Instructions

Pairs of potentially reorderable instructions can be identified via a dataflow analysis [13], similar to dependence analysis commonly used in compiler optimisation. However, rather than attempting to establish an absence of dependence, we are interested in demonstrating its presence, such that instruction reordering is not possible during execution. This notion of dependence is derived from the language's reordering relation, such that α is dependent on β iff $\beta \not\hookleftarrow \alpha$. All pairs of instructions for which a dependence cannot be established are assumed reorderable.

The approach is constructed as a backwards analysis over a component's program text, incrementally determining the instructions a particular instruction is dependent on and, inversely, those it can reorder before. Therefore, the analysis can be viewed as a series of separate analyses, one from the perspective of each instruction in the program text.

We describe one instance of this analysis for some instruction α. The analysis records a notion of α's cumulative dependencies, which simply begins as all instructions γ for which $\gamma \not\hookleftarrow \alpha$. The analysis commences at the instruction immediately prior to α in the program text and progresses backwards. For each instruction β we first determine if α depends on β by consulting α's cumulative dependencies. Given a dependence exists, α's cumulative dependencies are extended to include β's dependencies via a process we refer to as *strengthening*, such that the analysis may subsequently identify those instructions α is dependent on due to its dependence on β. If a dependence on β cannot be shown, the instructions are considered reorderable, subsequently requiring $\mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha)$ to be shown. Moreover, a process of *weakening* is necessary to remove α's cumulative dependencies that β may resolve due to forwarding.

To illustrate the evolving nature of cumulative dependencies, consider the sequence $\beta\,;\ \gamma\,;\ \alpha$ where $\gamma \not\hookleftarrow \alpha$ and $\beta \not\hookleftarrow \gamma$ but $\beta \hookleftarrow \alpha$. The analysis from the perspective of α starts at γ and identifies a dependence, due to $\gamma \not\hookleftarrow \alpha$. Therefore, α gains γ's dependencies via strengthening. The analysis progresses to the next instruction, β, for which a dependence can be established due to α's cumulative dependencies including $\beta \not\hookleftarrow \gamma$. Consequently, despite no direct dependency between α and β, the sequence does not produce reordering pairs for α. Repeating this process for γ and β ultimately finds no reordering pairs over the entire sequence, resulting in no $\mathit{rif}_{\mathsf{a}}$ checks.

A realistic implementation of this analysis is highly dependent on the language's reordering relation. In most examples, this relation only considers the variables accessed by the instructions and special case behaviours for memory barriers, as illustrated by the instantiation in Sect. 4. Consequently, cumulative dependencies can be efficiently represented as sets of such information, for example capturing the variables read by α and those instructions it depends on. This representation lends itself to efficient set-based manipulations for strengthening and weakening.

The analysis has been implemented for both a simple while language and an abstraction of ARMv8 assembly, with optimisations to improve precision in each context. In particular, precision can be improved through special handling of the forwarding case as the effects of forwarding typically result in trivial $\mathit{rif}_{\mathsf{a}}$ checks. To illustrate, recall that forwarding will replace a load from a shared variable with a value written to the variable earlier in the program. As assembly instructions are typically limited to performing at most a single variable store or load at a time, forwarding will transform shared-variable loads into purely thread-local operations by eliminating the load. Consequently, the reordering of such an instruction with any further instructions is trivially free of interference, as the environment cannot observe or influence thread-local instructions.

Both the while language and ARMv8 implementations have been encoded and verified in Isabelle/HOL, along with proofs of termination (following the approach suggested in [20]).

Address Calculations. Dependence analysis is considerably more complex in the presence of address calculations. Under such conditions, it is not possible to syntactically identify whether two instructions access equivalent addresses, complicating an essential check to establishing dependence. Without sufficient aliasing information the analysis must over-approximate and consider the two addresses distinct, potentially introducing excess reordering pairs.

The precision of the analysis can be improved using an alias analysis to first identify equivalent address calculations, feeding such information into the dependency checks. Precision may also be improved by augmenting the interference check, $\mathit{rif}_{\mathsf{a}}$, with any calculations that have been assumed to be distinct. For example, consider $[x] := e;\ [y] := f$, where $[v] := e$ represents a write to the memory address computed by the expression v. If an alias analysis cannot establish $x = y$, it is necessary to consider their interference. As they are assumed to reorder, a proof demonstrating $\mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, [x] := e, [y] := f)$ can assume $x \neq y$. Such a property extends to any other comparisons with cumulative dependencies.

We have implemented such improvements in our analysis for ARMv8, relying on manual annotations to determine aliasing address calculations. These aliasing annotations are subsequently added to each instruction's verification condition to ensure they are sound.

3.4 Interference Checking

Given the set of reordering pairs, it is necessary to demonstrate $\mathit{rif}_{\mathsf{a}}$ on each to demonstrate freedom of reordering interference. Many $\mathit{rif}_{\mathsf{a}}$ properties can be shown trivially. For example, if one instruction does not access shared memory, $\mathit{rif}_{\mathsf{a}}$ can be immediately shown to hold as no interference via $\mathcal{R}$ could take place. Additionally, if the two instructions access distinct variables and these variables are not related by $\mathcal{R}$, then no interference would be observed.

If these shortcuts do not hold, then it is necessary to consider $\mathit{rif}_{\mathsf{a}}$ directly via manual verification. The property can be rephrased in terms of weakest precondition calculation [7], providing some automation.

3.5 Elimination of Reordering Interference

Step 3 of the process is intended to handle situations where $\mathit{rif}_{\mathsf{a}}$ cannot be shown for a particular pair of instructions. A variety of techniques can be applied in such conditions, depending on the overall verification goals. In some circumstances, a failure to establish $\mathit{rif}_{\mathsf{a}}$ indicates a problematic reordering such that the out-of-order execution of the instruction pair will violate any variation of the desired rely/guarantee reasoning. In such circumstances, it is necessary to prevent reordering through the introduction of a memory barrier.

As these barriers incur a performance penalty, this is not a suitable technique to correct all problematic pairs. Some reordering pairs can instead be resolved by demonstrating stronger properties during the standard rely/guarantee reasoning in step 4. We describe a series of techniques that can be employed to extract these stronger properties by modifying a program's verification conditions and/or abstracting over its behaviour. These techniques, while incomplete, are easily automated and cover the majority of cases.

Strengthening. Establishing $\mathit{rif}_{\mathsf{a}}$ may fail in cases where an instruction in a reordering pair modifies the other's verification condition. In such circumstances, it is possible to *strengthen* verification conditions such that the interference becomes benign by capturing both the in-order and out-of-order execution behaviours. Given a reordering pair $(\beta,\, \alpha)$, this is achieved by first determining the weakest P that solves $P\{\alpha_{\langle\beta\rangle};\ \beta\}(\mathit{true})$, representing the implications of each instruction's verification conditions when executed out-of-order. β's verification condition is then modified by conjoining this P to it, such that the constraints of the out-of-order execution are established during standard reasoning.

For example, consider the component $(y = 0)\{z := z + 1;\ x := y\}(\mathit{true})$ where, due to a specialised analysis, the assignment to x has the verification condition $z = 1 \vee y = 0$ (and that for the assignment to z is true). Assume that $\mathcal{R}$ is the identity relation, i.e., no variables are changed by environment steps, and $\mathcal{G}$ is true. This component may be trivially verified when ignoring weak memory effects, as the verification condition for $x := y$ is transformed by $z := z + 1$ into $z = 0 \vee y = 0$, clearly implied by the specified precondition $y = 0$.

However, assuming the two assignments may be reordered, it is necessary to establish $\mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, z := z + 1, x := y)$. Unfortunately, such a property does not hold. Recall that $\mathit{rif}_{\mathsf{a}}$ requires an out-of-order judgement for all valid in-order judgements under all possible pre- and postconditions. Therefore, we need only identify conditions that are valid for an in-order execution but invalid for the out-of-order to disprove $\mathit{rif}_{\mathsf{a}}$. This can be seen with the precondition $z = 0$ and postcondition true, as we can establish an in-order judgement of $(z = 0)\{z := z + 1;\ x := y\}(\mathit{true})$ via the same reasoning as above. This does not hold for the out-of-order case of $(z = 0)\{x := y;\ z := z + 1\}(\mathit{true})$, as $z = 0$ does not imply $z = 1 \vee y = 0$.

Applying the strengthening approach, we compute P for the out-of-order execution as $z = 1 \vee y = 0$. The verification condition for $z := z + 1$ is then conjoined with this P to derive its new condition. As it was originally true, this trivial results in the verification condition becoming $z = 1 \vee y = 0$. With this new verification condition, we establish $\mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, z := z + 1, x := y)$, by invalidating the in-order judgement.

With *rif* established, the standard rely/guarantee reasoning in step 4 must demonstrate $(y = 0)\{z := z + 1;\ x := y\}(\mathit{true})$, with the strengthened verification condition for $z := z + 1$. This obviously holds given $y = 0$ initially.

Ignored Reads. An additional issue when correcting for $\mathit{rif}_{\mathsf{a}}$ derives from the quantification of the pre- and post-states. This quantification reduces the proof burden, such that only pairs of reorderable instruction must be considered, but can introduce additional proof effort where the precise pre- and post-states are well known and limited reordering takes place. For instance, consider the simple component $(\mathit{true})\{x := 1;\ z := y\}(x = 1)$ with a rely specification that will preserve the values of x and z always and the value of y given $x = 1$. The rely/guarantee reasoning to establish this judgement is trivial. However, the component will fail to demonstrate $\mathit{rif}_{\mathsf{a}}$ when considering the reordering of $x := 1$ and $z := y$, as their program order execution may establish the stronger $(\mathit{true})\{x := 1;\ z := y\}(x = 1 \wedge z = y)$, whereas the reordered cannot.

We employ two techniques to amend such situations. The most trivial is a weakening of the component's $\mathcal{R}$ specification to remove the relationship between y and x, as it is unnecessary for the component's verification. Otherwise, if this is not possible, the component can be abstracted to $(\mathit{true})\{x := 1;\ \mathsf{chaos}\ z\}(x = 1)$, where $\mathsf{chaos}\ v$ encodes a write of any value to the variable v. Consequently, the read of y is ignored. Both standard rely/guarantee reasoning and *rif* can be established for this modified component, subsequently enabling verification of the original via a refinement argument.

We propose the automatic detection of those reads that do not impact reasoning and, therefore, can be ignored when establishing *rif*. In general, such situations are rare as the analysis targets assembly code produced via compilation. Consequently, such unnecessary reads are eliminated via optimisation. Moreover, the $\mathcal{R}$ specification infrequently over-specifies constraints on the environment.

3.6 Soundness

Soundness of the proof system has been proven in Isabelle/HOL and is available in the accompanying theories at https://bitbucket.org/wmmif/wmm-rg.

3.7 Completeness

The proof system is incomplete due to the over-approximations required to reduce reasoning to pairs of reorderable instructions. This is by design, as the approach benefits significantly from such simplifications and the problematic cases appear rare, particularly when the techniques suggested in Sect. 3.5 are applied. As an illustration of these problematic cases, consider

$(P)\{x := v_1;\ y := v_2\}(true)$, where P is some precondition, the rely condition preserves the values of x and y, and the guarantee is $true$. Moreover, assume the verification condition for $y := v_2$ requires $x \neq y$ and the instructions can reorder.

When considering both possible execution orderings a sufficient precondition P would be $x \neq y \wedge v_1 \neq y$, as this captures the constraints imposed by the single verification condition. However, the *rif* approach will introduce an additional, unnecessary condition to establish $rif_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, x := v_1, y := v_2)$. First, observe that $x := v_1$ modifies the verification condition for $y := v_2$. Therefore, the verification condition for $x := v_1$ must be strengthened to $x \neq y$, following the same approach as the example in Sect. 3.5. However, the resulting instructions are still not interference free, as $y := v_2$ can now modify the new verification condition for $x := v_1$. This can be resolved through an additional application of strengthening, extending the verification condition for $x := v_1$ to $x \neq y \wedge x \neq v_2$. Consequently, the approach requires a precondition P stronger than $x \neq y \wedge v_1 \neq y \wedge x \neq v_2$, over-approximating the true requirements.

This failure can be attributed to the lack of delineation between the original components of a verification condition and those added due to strengthening, as interference checks on the latter are not necessary. We leave an appropriate encoding of such differences to future work.

4 Instantiating the Proof System

In this section, we illustrate instantiating the proof system with a simple while language. The Isabelle/HOL theories accompanying this work also include an instantiation for ARMv8 assembly which has been used to verify an implementation of the Chase-Lev work-stealing deque developed for ARM [17].

We distinguish three different types of state variables: global variables *Glb* and local variables *Loc*, which are program variables, and global auxiliary variables *Aux*. Local variables are unique to each thread and cannot be accessed by others.

Atomic instructions in our language comprise skips, assignments, guards, two kinds of fences, and coupling of an instruction with an auxiliary variable assignment and/or with a specific verification condition (similar to an assertion)

$$inst ::= \mathsf{nop} \mid v := e \mid \mathsf{guard}\ p \mid \mathsf{fence} \mid \mathsf{cfence} \mid \langle inst, a := e_a \rangle \mid \{\!|\, p_a \,|\!\} inst$$

where v is a program variable, e an expression over program variables, p a boolean expression over program variables, a an auxiliary variable, e_a an expression over program and auxiliary variables, p_a a boolean expression over program and auxiliary variables, and $\langle inst, a := e_a \rangle$ denotes the atomic execution of *inst* followed by $a := e_a$.

Commands are defined over atomic instructions and their combinations

$$cmd ::= inst \mid cmd\,;\ cmd \mid \mathsf{if}\ p\ \mathsf{then}\ cmd\ \mathsf{else}\ cmd \mid \mathsf{do}\ cmd\ \mathsf{while}(p, Inv)$$

where *Inv* denotes a loop invariant. Instructions instantiate individual instructions (i.e., α) in our abstract language. Sequential composition directly instantiates its abstract counterpart. Conditionals and loops are defined via the choice and iteration operator, i.e., $\mathsf{if}\ p\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is defined as $(\mathsf{guard}\ p)\ ;\ c_1 \sqcap (\mathsf{guard}\ \neg p)\ ;\ c_2$, and $\mathsf{do}\ c\ \mathsf{while}(p, \mathit{Inv})$ as $(c\ ;\ (\mathsf{guard}\ p))^*\ ;\ \ c\ ;\ (\mathsf{guard}\ \neg p)$, where the invariant *Inv* holds at the start of c's execution.

A reordering relation $\overset{inst}{\hookleftarrow}$ (and its inverse $\overset{inst}{\not\hookleftarrow}$) is defined over atomic instructions based on syntactic independence of reorderable instruction [6]. For all instructions α and β

$$\mathsf{fence} \overset{inst}{\not\hookleftarrow} \alpha,\quad \alpha \overset{inst}{\not\hookleftarrow} \mathsf{fence},\quad \mathsf{guard}\ p \overset{inst}{\not\hookleftarrow} \mathsf{cfence},$$
$$\mathsf{cfence} \overset{inst}{\not\hookleftarrow} \alpha \text{ if } rd(\alpha) \nsubseteq Loc,$$
$$\mathsf{guard}\ p \overset{inst}{\not\hookleftarrow} \alpha \text{ if } wr(\alpha) \in Glb \vee wr(\alpha) \in rd(\mathsf{guard}\ p) \vee rd(\mathsf{guard}\ p) \cap rd(\alpha) \nsubseteq Loc,$$

and for all other cases,

$$\beta \overset{inst}{\hookleftarrow} \alpha \text{ if } wr(\beta) \neq wr(\alpha) \wedge wr(\alpha) \notin rd(\beta) \wedge rd(\beta) \cap rd(\alpha) \subseteq Loc.$$

where $wr(\alpha)$ is the program variable written by α and $rd(\alpha)$ the program variables read by α. Note that a cfence is used to prevent speculative reads of global variables when placed prior to the reading instruction and after a guard [6].

Forwarding a value to an assignment instruction in our language is defined as $(v_\alpha := e_\alpha[v_\beta \backslash e_\beta]) \prec (v_\beta := e_\beta) \prec (v_\alpha := e_\alpha)$ and to a guard as $(\mathsf{guard}\ p[v_\alpha \backslash e_\alpha]) \prec (v_\alpha := e_\alpha) \prec (\mathsf{guard}\ p)$ where $e[v \backslash e']$ replaces every occurrence of v in e by e'. The instruction after forwarding carries the same verification condition as the original instruction, i.e., $vc(\alpha_{\langle\beta\rangle}) = vc(\alpha)$.

Note that auxiliary variable updates and verification conditions do not influence the reordering relation, as they will not constrain execution behaviour. Both of these annotations remain linked to their respective instructions during reordering and forwarding.

4.1 Peterson's Mutual Exclusion Algorithm

We use Peterson's mutual exclusion algorithm [22] to demonstrate the workings of the instantiated proof system. The program (shown in Fig. 2) consists of two threads, each of which aims to get exclusive access to the shared variables when they are modified in the thread's critical section (which is represented by a placeholder in the figure). Fences have been added where required for the instantiated proof system.

In order to demonstrate our rely/guarantee reasoning, we define a rely condition for each thread that is reflected by the other thread's guarantee condition. These conditions refer to an auxiliary variable $a : Boolean$, which captures which thread can be in its critical section: when a is false, the left thread t_0 cannot be in its critical section, and when a is true, the right thread t_1 cannot be in its critical section. The rely/guarantee conditions can then be phrased as follows:

```
{ $flag0 := true$;                                        { $flag1 := true$;
  fence;                                                    fence;
  $\langle turn := true,\ a := false\rangle$                $\langle turn := false,\ a := true\rangle$
  fence;                                                    fence;
  do                                                        do
     $\langle r0 := flag1, a := a \vee \neg flag1\rangle$;       ||      $\langle r3 := flag0, a := a \wedge flag0\rangle$;
     $r1 := turn$;                                              $r4 := \neg turn$;
  while($r0 \wedge r1, flag0 \wedge (a \vee turn)$);        while($r3 \wedge r4, flag1 \wedge (\neg turn \vee \neg a)$);
  $\{\!| a |\!\}$ cfence;                                   $\{\!| \neg a |\!\}$ cfence;
        $critical\_section$;                                      $critical\_section$;
  $\{\!| a |\!\}$ fence;                                    $\{\!| \neg a |\!\}$ fence;
  $flag0 := false$;                                         $flag1 := false$;
}                                                         }
```

Fig. 2. Peterson's algorithm with fences to guarantee correctness under weak memory

$$\begin{aligned}\mathcal{R}_0 = \mathcal{G}_1 = \ & \mathit{flag}0 = \mathit{flag}0' \wedge ((\mathit{flag}0 \wedge a) \Rightarrow a') \wedge \\ & (\mathit{turn} = \mathit{turn}' \vee (\mathit{turn} \wedge \neg \mathit{turn}' \wedge (\mathit{flag}0 \Rightarrow a'))) \\ \mathcal{R}_1 = \mathcal{G}_0 = \ & \mathit{flag}1 = \mathit{flag}1' \wedge ((\mathit{flag}1 \wedge a') \Rightarrow a) \wedge \\ & (\mathit{turn} = \mathit{turn}' \vee (\neg \mathit{turn} \wedge \mathit{turn}' \wedge (\mathit{flag}1 \Rightarrow \neg a')))\end{aligned}$$

That is, $\mathcal{R}_0$ specifies that (i) the right thread t_1 does not modify $\mathit{flag}0$, (ii) if t_0 is in the critical section, which is the case when $(\mathit{flag}0 \wedge a)$, t_1 cannot change a, and (iii) either turn remains unchanged or it is set to false in which case a cannot be falsified if t_0 has not exited its critical section, specified by $(\mathit{flag}0 \Rightarrow a')$. $\mathcal{R}_1$ can be explained similarly.

Reasoning over t_0's code c_0 requires showing that it is reordering interference free, $\mathit{rif}(\mathcal{R}, \mathcal{G}, c_0)$, which holds if $\mathit{rif}_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha)$ for all instructions α, β in c_0 such that $\beta \hookleftarrow \alpha$ (see (9)–(11)). As an example we discuss reordering interference freedom on the first two instructions of the original code (without fence instructions) $\mathit{flag}0 := \mathit{true}\,;\ \langle \mathit{turn} := \mathit{true}, a := \mathit{false}\rangle$ which are syntactically independent and, following the definition of $\overset{inst}{\hookleftarrow}$, can execute out-of-order. To show $\mathit{rif}_{\mathsf{a}}$ requires proving that $\forall\, P, Q, M$.

$\mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\mathit{flag}0 := \mathit{true}\}M \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} M\{\langle \mathit{turn} := \mathit{true}, a := \mathit{false}\rangle\}Q \Rightarrow$
$\exists\, M'.\, \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\langle \mathit{turn} := \mathit{true}, a := \mathit{false}\rangle\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} M'\{\mathit{flag}0 := \mathit{true}\}Q$.

Recall from (6) that the pre- and post-conditions of a Hoare triple are stable and that the instruction satisfies the component's guarantee. The latter holds for $\mathit{flag}0 := \mathit{true}$ since $\mathcal{G}_0$ does not constrain $\mathit{flag}0$, and for $\langle \mathit{turn} := \mathit{true}, a := \mathit{false}\rangle$ since $\mathcal{G}_0$ always allows turn to stay true when it is true and change to true from false when a changes to false (since this makes $\mathit{flag}1 \Rightarrow \neg\, a'$ true).

Let Q be the predicate $\mathit{flag}0 \wedge (a \vee \mathit{turn})$ which is stable under $\mathcal{R}_0$ ($\mathit{flag}0$ cannot be changed, a cannot become false when $\mathit{flag}0$ is true, and given that $\mathit{flag}0$ is true turn can only become false when a becomes true). Then, via standard weakest precondition calculation, M could be $\mathit{flag}0$ and P could be true each of which are also stable under $\mathcal{R}_0$. Hence, the antecedent of the implication holds. However, weakest precondition calculation for the same Q in the consequent requires that $M' \Rightarrow (a \vee \mathit{turn})$. This can only be made stable by including $\mathit{flag}0$ in M'. Hence, via weakest precondition calculation again, we require that $P \Rightarrow \mathit{flag}0$. That is, the program would need to be initialised with $\mathit{flag}0$ which is not suitable in the context of Peterson's algorithm (as *flag*0 should only be set when t_0 wants to enter the critical section [22]). Therefore, the analysis suggests including a fence instruction to prevent the reordering of the first and second instruction.

For all other reordering pairs in t_0 we follow similar reasoning (and similarly for the pairs of t_1), resulting in additional fence instructions being required, as shown in Fig. 2, to ensure the correct working of the algorithm. Note that fences are not required between the instructions in the loop bodies. While these instructions are reorderable under $\overset{inst}{\hookleftarrow}$, they can be proven to be reordering interference free. This is demonstrated in our Isabelle/HOL theories.

5 Conclusion

This paper presents a truly thread-local approach to reasoning about concurrent code on a range of weak memory models. It employs standard rely/guarantee reasoning to handle interference between threads, and a separate check of *reordering interference freedom* to handle interference within a thread due to weak memory behaviour.

Reordering interference freedom provides evidence that the weak memory model under consideration will not invalidate properties shown via standard rely/guarantee reasoning. It is a novel concept that hinges on a thread-local reordering semantics which can be defined for any *multicopy atomic* weak memory model, i.e., where a thread's stores become observable to all other threads at the same time. Such memory models include the widely used x86-TSO and ARMv8 processor architectures, and the open-source RISC-V architecture.

Importantly, our approach reduces the check of reordering interference to only pairs of instructions, thereby significantly reducing its complexity. Moreover, the computation of these pairs has been automated along with the validation of trivially benign reordering pairs. Consequently, the only additional manual burden is the establishment of freedom of reordering interference between instruction pairs exhibiting intricate interactions via rely/guarantee conditions. In situations where freedom of reordering interference cannot be shown, our approach includes methods to amend the program, to prohibit reordering behaviour, or modify its verification conditions, such that stronger arguments for reordering interference freedom may be shown.

The paper exemplifies an instantiation of the approach for a simple while language and memory model, and uses it to verify the mutual exclusion property of Peterson's algorithm. The approach is also instantiated to a more realistic assembly language, verifying a work-stealing deque developed specifically for ARM processors. These results, along with a soundness proof for our approach, have been encoded in Isabelle/HOL.

References

1. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 56–74. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_4
2. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 150:1–150:29 (2019)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1-7:74 (2014)
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 55–66. ACM (2011)
5. Boehm, H.: Can seqlocks get along with programming language memory models? In: Zhang, L., Mutlu, O. (eds.) Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI 2012, pp. 12–20. ACM (2012)
6. Colvin, R.J., Smith, G.: A wide-spectrum language for verification of programs on weak memory models. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 240–257. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_14
7. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990). https://doi.org/10.1007/978-1-4612-3228-5
8. Flur, S., et al.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 608–621. ACM (2016)
9. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 355–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_19
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
11. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
12. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 175–189. ACM (2017)
13. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of POPL, pp. 194–206. ACM (1973)
14. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: persistency semantics for verification under ext4. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
15. Kusano, M., Wang, C.: Thread-modular static analysis for relaxed memory models. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 337–348. ACM (2017)

16. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
17. Lê, N., Pop, A., Cohen, A., Zappa Nardelli, F.: Correct and efficient work-stealing for weak memory models. In: PPoPP 2013, pp. 69–80. ACM (2013)
18. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds.) Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275. ACM (1996)
19. Moir, M., Shavit, N.: Concurrent data structures. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications. Chapman and Hall/CRC (2004)
20. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10542-0
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
22. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
23. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. PACMPL **2**(POPL), 19:1–19:29 (2018)
24. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_4
25. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 175–186. ACM (2011)
26. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
27. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2011)
28. Stølen, K.: A method for the development of totally correct shared-state parallel programs. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 510–525. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54430-5_110
29. Suzanne, T., Miné, A.: Relational thread-modular abstract interpretation under relaxed memory models. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 109–128. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_6
30. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, pp. 691–707. ACM (2014)
31. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, pp. 867–884. ACM (2013)

32. Waterman, A., Lee, Y., Patterson, D.A., Asanovi, K.: The RISC-V instruction set manual. Volume 1: User-level ISA, version 2.2. Technical report EECS-2016-118, Department of Electrical Engineering and Computer Science, University of California, Berkeley (2016)
33. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: IEEE Computer Security Foundations Symposium (CSF 2021). IEEE Computer Society (2021)
34. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997)