Relational Thread-Modular Abstract Interpretation Under Relaxed Memory Models *

Thibault Suzanne[1,2,3] and Antoine Miné[3]

[1] Département d'informatique de l'ENS, École Normale Supérieure, CNRS, PSL Research University, 75005 Paris, France
thibault.suzanne@ens.fr
[2] Inria
[3] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France
Antoine.Mine@lip6.fr

Abstract We address the verification problem of numeric properties in many-threaded concurrent programs under weakly consistent memory models, especially TSO. We build on previous work that proposed an abstract interpretation method to analyse these programs with relational domains. This method was not sufficient to analyse more than two threads in a decent time. Our contribution here is to rely on a rely-guarantee framework with automatic inference of thread interferences to design an analysis with a thread-modular approach and describe relational abstractions of both thread states and interferences. We show how to adapt the usual computing procedure of interferences to the additional issues raised by weakly consistent memories. We demonstrate the precision and the performance of our method on a few examples, operating a prototype analyser that verifies safety properties like mutual exclusion. We discuss how weak memory models affect the scalability results compared to a sequentially consistent environment.

1 Introduction

Multicore programming is both a timely and challenging task. Parallel architectures are ubiquitous and have significant advantages related to cost effectiveness and performance, yet they exhibit a programming paradigm that makes reasoning about the correctness of the code harder than within sequential systems. Weakly consistent memory models, used to describe the behaviour of distributed systems and multicore CPUs, amplify this fact: by allowing more optimisations, they enable programs to run even faster; however this comes at the cost of counter-intuitive semantic traits that further complicate the understanding of these programs, let alone their proof of correctness. These difficulties coupled

* This work is supported in part by the ITEA 3 project 14014 (ASSUME) and in part by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

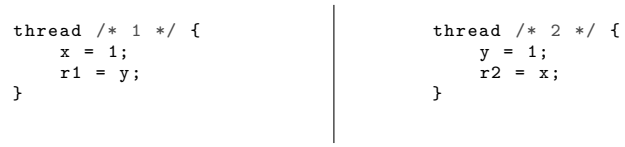

Figure 1: A simple program with counter-intuitive possible results on x86.

with the use of such architectures in critical domains call for automatic reasoning methods to ensure correctness properties on concurrent executions.

In a previous work [20], we proposed an abstract interpretation method to verify such programs. However, this method worked by building a global control graph representing all possible interleavings of the threads of the target program. The size of this graph grows exponentially with the number of threads, which makes this method unable to scale. This paper describes a thread-modular analysis that circumvents this problem by analysing each thread independently, propagating through these thread analyses their effect on the execution of other threads. We target in particular the *Total Store Ordering* (TSO) and *Partial Store Ordering* (PSO) memory models.

1.1 Weak Memory Models

A widespread paradigm of concurrent programming is that of shared memory. In this paradigm, the intuitive semantics conforms to sequential consistency (SC) [12]. In SC, the allowed executions of a concurrent program are the interleavings of the instructions of its threads. However, modern multicore architectures and concurrent programming languages do not respect this property: rather, for optimisation reasons, they specify a *weakly consistent memory model* that relaxes sequential consistency and allows some additional behaviors.

We mainly target the TSO (Total Store Ordering) memory model, which is amongst others known for being the base model of `x86` CPUs [19]. In this model, a thread cannot immediatly read a store from another thread: they write through a totally ordered *store buffer*. Each thread has its own buffer. Non-deterministically, the oldest entry of a store buffer can be flushed into the memory, writing the store value to the corresponding shared variable. When attempting to read the value of some variable, a thread begins by looking at the most recent entry for this variable in its store buffer. If there is none, it reads from the shared memory.

The program of Figure 1 exhibits a non-intuitive behaviour. In SC, after its execution from a zeroed memory, either `r1` or `r2` must be equal to `1`. However, when executed on `x86`, one can observe `r1 = 0 && r2 = 0` at the end. This happens when Thread 1 (respectively Thread 2) reads the value of `x` (respectively `y`) whereas Thread 2 (respectively Thread 1) has not flushed its store from its buffer yet.

Another related model, PSO (Partial Store Ordering), is strictly more relaxed than TSO, in that its buffers are only partially ordered: stores to a same

variable keep their order, but stores to different variables can be flushed in any order into the memory. Another way of expressing it consists in having a totally ordered buffer for each thread and each variable, with no order between different buffers. Both models define a `mfence` instruction that flushes the buffer(s) of the thread that executes it. A systematic insertion of `mfence` allows to get back to sequential consistency, but has a performance cost, thus one should avoid using this instruction when it is not needed for correctness.

As we stated earlier, our main target is TSO, as most previous abstract interpretation works. It acts as a not too complex but real-life model, and fills a sweet spot where relaxed behaviours actually happen but do not always need to be forbidden for programs to be correct. However, to design a computable analysis that stays sound, we were forced to drop completeness by losing some (controlled) precision: this is the foundation of abstract interpretation [5]. Our abstraction ignores the write order between two different variables, to only remember sequences of values written into each variable independently. This design choice makes our analysis sound not only under TSO, but also incidentally under PSO. Therefore we will present it as a PSO analysis since it will simplify the presentation, although the reader should have in mind that it stays sound w.r.t. TSO. The loss of precision, in practice, incurred by a PSO analysis on a TSO program will be discussed in Section 4.

We believe our analysis can be extended to more relaxed models such as POWER/ARM by adding "read buffers". This extension could pave the way for the C and Java models, which share some concepts, but we did not have the time to properly study them yet. However we rely on a very operational model: more complex ones are axiomatically defined, so one will need to provide a sound operational overapproximation before doing abstraction.

1.2 Abstraction of Relaxed Memory

To analyse concurrent programs running under PSO, we focus on abstract interpretation [5]. The additional difficulty to design abstractions when considering store-buffer-based memory models lies in buffers: they are unbounded and their size changes dynamically and non-deterministically. This work builds on our previous work [20] that proposed an efficient abstraction for representing buffers.

Our implementation (cf. Section 4) targets small algorithms implementable in assembly. Hence the core language of the programs we aim to analyse is a minimal imperative language, whose syntax is defined in Figure 3, Section 2. The program is divided in a fixed number of threads, and they all run simultaneously. Individual instructions run atomically (one can always decompose a non-atomic instruction into atomic ones). We believe that additional features of a realistic programming language, such as data structures and dynamic allocation, are orthogonal to this work on weakly consistent memory: we focus on numerical programs, yet one can combine our abstractions with domains targetting these features to build a more complete analysis.

The domain proposed in our previous paper [20] relies on a summarisation technique initially proposed by Gopan et al. [6] to abstract arrays, which they

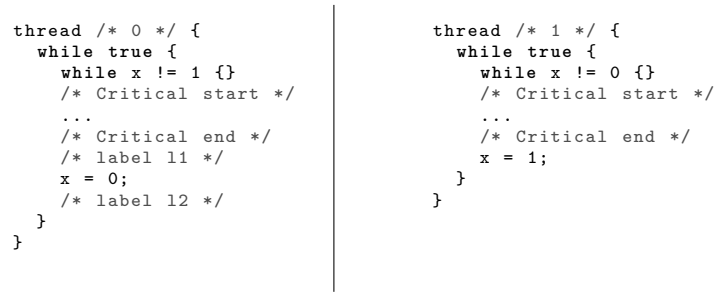

Figure 2: Round-robin: a concurrent program example.

adapt to abstract unbounded FIFO queues. Summarisation consists in grouping together several variables $x_1, ..., x_n$ in a numerical domain into a single summarised variable x_{sum}, which retains each possible value of every x_i. For instance, let us consider two possible states over three variables: $(x, y, z) \in \{(1, 2, 3); (4, 5, 6)\}$. If we regroup x and y into a summarised variable v_{xy}, the possible resulting states are $(v_{xy}, z) \in \{(1, 3); (2, 3); (4, 6); (5, 6)\}$. Note that, due to summarisation, these concrete states of (x, y, z) are also described by that abstract element: $(1, 1, 3)$, $(2, 2, 3)$, $(2, 1, 3)$, $(4, 4, 6)$, $(5, 5, 6)$, $(5, 4, 6)$.

We use this technique to summarise the content of each buffer, excluding the most recent entry that plays a special role when reading from the memory. Once summarisation is done, we obtain states with bounded dimensions that can be abstracted with classic numerical domains. This abstraction is described at length in our previous paper [20].

1.3 Interferences: Thread-Modular Abstract Interpretation

The immediate way of performing abstract interpretation over a concurrent program is to build the global control graph, product of the control graph of each thread, that represents each possible interleaving. This graph has a size which is exponential in the number of threads and linear in each thread size: it does not scale up. Thread-modular analyses have been designed to alleviate this combinatorial explosion [8, 10, 15–17]. Amongst them, we use the formal system of interferences, that has been proposed by Miné [15] to analyse each thread in isolation, generating the effects it can have on the execution of other threads, and taking into account the effects generated by these other threads. Thread-modular analysis scales up better because the analysis is linear in the sum of thread sizes (instead of their product), times the number of iterations needed to stabilise the interferences (which is low in practice, and can always be accelerated by widening [15]).

The effects generated by this analysis are named interferences. Consider the program in Figure 2, running in sequential consistency from a zeroed memory. This program is a standard round-robin algorithm, whose purpose is to alternate the presence of its threads in the critical section. To analyse it, we first consider

Thread 0 and analyse it separately as if it were a sequential program. It cannot enter the critical section since `x` is initially equal to `0`, so the analysis ends here. Then we analyse Thread 1, that immediately exits its inner loop and then enters the critical section, after which it sets `x` to `1`. We then generate the *simple interference* $T1 : x \mapsto 1$, that means that Thread 1 can put 1 in x. Every read from x by a thread can now return 1 instead of the value this thread stored last, in a flow insensitive way. Afterwards, Thread 1 analysis ends: it cannot enter back its critical section, since `x` is still equal to `1` when it tries again. We go back to Thread 0. The new analysis will take into account the interference from Thread 1 to know that `x` can now be equal to `1`, and thus that Thread 0 can enter its critical section. It will generate the interference $T0 : x \mapsto 0$, and notice that the critical section can be entered several times when applying the interference from Thread 1. Then the second analysis of Thread 1 will also determine that Thread 1 can enter its critical section more than once. No more interference is generated, and the global analysis has ended. It is thread-modular in the sense that it analyses each thread code in isolation from other thread code.

This *simple interference* analysis is provably sound: in particular, it has managed to compute that both threads can indeed enter their critical section. However, it did not succeed in proving the program correct. In general, simple interferences associate to each variable (an abstraction of) the set of its values at each program point. They are non-relational (in particular, there is no relation between the old value of a variable and its new value in an interference) and flow insensitive. To alleviate this problem, previous works [15, 16] introduced relational interferences, that model sets of possible state transitions caused by thread instructions between pairs of program points, i.e., they model the effect of the thread in a fully relational and flow-sensitive way, which is more precise and more costly, while still being amenable to classic abstraction techniques. For instance, in the program of Figure 2, one such interference would be "When `x` is equal to `1`, and Thread 1 is not in its critical section, Thread 0 can write `0` in `x`; and by doing so it will go from `label l1` to `label l2`". The relational interference framework is complete for reachability properties thus not computable, but Monat and Miné [16] developed precise abstractions of interferences in SC that allow proving this kind of programs in a decidable way.

In this paper, we will combine such abstractions with the domains for weakly consistent memory to get a computable, precise and thread-modular abstract interpretation based analysis under TSO. We implemented this analysis and provided some results on a few examples. We mostly aim to prove relational numerical properties on small albeit complex low-level programs. These programs are regarded as difficult to check — for instance, because they implement a synchronisation model and are thus dependent on some precise thread interaction scenario. We show that our analysis can retain the precision needed to verify their correctness, while taking advantage of the performances of a modular analysis to be able to efficiently analyse programs with more than 2 threads, which is out of reach of most non-modular techniques.

⟨*prog*⟩ ::= ⟨*thread*⟩ *

⟨*thread*⟩ ::= `thread` `{` ⟨*stmt*⟩ `}`

⟨*stmt*⟩ ::=
| ⟨*var*⟩ `=` ⟨*expr*⟩
| `if` ⟨*expr*⟩ `{` ⟨*stmt*⟩ `}` [`else` `{` ⟨*stmt*⟩ `}`]
| `while` ⟨*expr*⟩ `{` [⟨*stmt*⟩] `}`
| ⟨*stmt*⟩ `;` ⟨*stmt*⟩

⟨*expr*⟩ ::=
| ⟨*var*⟩
| $n \in \mathbb{Z}$
| ⟨*expr*⟩ † ⟨*expr*⟩
| ⋆⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`

⟨*var*⟩ ::= `x`, `y`, `z`...

⟨†⟩ ::= `*` | `/` | `+` | `-` | `=` | `<` | `>` | `<=` | `>=` | `&&` | `||`

⟨⋆⟩ ::= `not` | `-`

Figure 3: Program syntax

Section 2 describes the monolithic and modular concrete semantics of concurrent programs running under the chosen memory model. Section 3 defines a computable modular abstraction for these programs. Section 4 presents experimental results on a few programs using a test implementation of our abstract domains and discusses scaling when considering weakly consistent memories. We present a comparison with related works in Section 5. Section 6 concludes.

The monolithic semantics of Section 2 has been dealt with in our previous work [20]. Our contribution is composed of the modular semantics of Section 2, Section 3 and Section 4.

2 Concrete Semantics

2.1 Interleaving Concrete Semantics

Figure 3 defines the syntax of our programs. We specify in Figure 4 the domain used in the concrete semantics. We consider our program to run under the PSO memory model. Although TSO is our main target, PSO is strictly more relaxed, therefore our PSO semantics stays sound w.r.t. TSO.

Notations. *Shared* is the set of shared variable symbols, *Local* is the set of thread-local variables (or registers). Unless specified, we use the letters x, y, z for *Shared* and r for *Local*. $\mathbb{V}$ is the value space of variables, for instance $\mathbb{Z}$ or $\mathbb{Q}$. e is an arithmetic expression over elements of $\mathbb{V}$ and *Local* (we decompose expressions involving *Shared* variables into reads of these variables into *Local* variables and actually evaluating the expression over these locals). $\circ$ is function composition. $\mathbb{L}$ is a set of *program points* or *control labels*.

Remark 1. $\mathscr{D}$ is isomorphic to a usual vector space. As such, it supports usual operations such as variable assignment ($x := e$) or condition and expression evaluation. We will also use the *add* and *drop* operations, which respectively add an unconstrained variable to the domain, and delete a variable and then project on the remaining dimensions.

$$
\begin{array}{rll}
Mem \triangleq Shared \to \mathbb{V} & & \text{Shared memory} \\
TLS \triangleq Local \to \mathbb{V} & & \text{Thread Local Storage (registers)} \\
\forall x \in Shared, Buf_x^T \triangleq \bigcup_{N \in \mathbb{N}} \left(\left\{ x_1^T, ..., x_N^T \right\} \to \mathbb{V} \right) & & \text{Buffers} \\
\mathscr{S} \triangleq Mem \times TLS \times \prod_{\substack{x \in Shared \\ T \in Thread}} Buf_x^T & & \text{Program states} \\
\mathscr{D} \triangleq \mathcal{P}(\mathscr{S}) & & \text{Sets of program states} \\
\mathscr{C} \triangleq Thread \to \mathbb{L} & & \text{Control states}
\end{array}
$$

Figure 4: A concrete domain for PSO programs.

As the variables in *Shared* live both in the buffers and the memory, we will use the explicit notation x^{mem} for the bindings of *Mem*. We represent a buffer of length N of the thread T for the variable x by N variables $x_1^T, ..., x_N^T$ containing the buffer entries in order, x_1^T being the most recent one and x_N^T the oldest one.

This concrete domain has been used by in our previous work [20] to define the concrete non-modular semantics of the programs. For each statement corresponding to a control graph edge *stmt* and for each thread T, they define the operator $[\![stmt]\!]_T : \mathscr{D} \to \mathscr{D}$ that computes the set of states reachable when T executes *stmt* from any state in an input set. $[\![x := e]\!]_T$ adds the value of e into the buffer of T for the variable x, shifting the already present x_i^T. $[\![r := x]\!]$ reads x_1^T, or, if not defined (the buffer is empty), x^{mem}. $[\![\texttt{flush}\ x]\!]_T$ removes the oldest entry of x and writes its value in x^{mem}. $[\![\texttt{mfence}]\!]_T$ ensures that all buffers of T are empty before executing subsequent operations. The formal semantics is recalled in Figure 5. For convenience reasons, we define $[\![.]\!]$ on state singletons $\{S\}$ and then lift it pointwise to any state set.

The standard way of using this semantics consists in constructing the product control graph modeling all interleavings of thread executions of a program from the control graph of each thread it is composed of. The semantics of the program is then computed as the least fixpoint of the equation system described by this graph, whose vertices are control states (elements of $\mathscr{C}$ as defined in Figure 4) and edges are labelled by operators of Figure 5. The non-determinism of flushes can be encoded by a self-loop edge of label $[\![\texttt{flush}\ x]\!]_T$ for each $x \in Shared$, $T \in Thread$ on each vertex in the graph. However, we will now state a lemma that will provide us a new and more efficient computation method.

Lemma 1 (Flush commutation). *Let $x \in Shared$ and $[\![op_{\not x}]\!]$ be an operator that neither writes to nor reads from x, that is either $[\![y := expr]\!]$, $[\![r := y]\!]$, $[\![r := expr]\!]$ or $[\![\texttt{condition}]\!]$, with $\forall y \in Shared, y \neq x \Rightarrow y \notin \texttt{condition}$. Then:*

$$\forall T \in \mathit{Thread}, [\![.]\!]_T : \mathscr{D} \to \mathscr{D}$$

$$[\![x := e]\!]_T\{S\} \triangleq [\![x_1^T := e]\!] \circ [\![x_2^T := x_1^T]\!] \circ ... \circ [\![x^T_{L_S^T(x)+1} := x^T_{L_S^T(x)}]\!] \circ [\![\mathit{add}\ x^T_{L_S^T(x)+1}]\!]\{S\}$$

$$[\![r := x]\!]_T\{S\} \triangleq \begin{cases} [\![r := x^{mem}]\!]S & \text{if } L_S^T(x) = 0 \\ [\![r := x_1^T]\!]S & \text{if } L_S^T(x) \geq 1 \end{cases} \qquad [\![\texttt{mfence}]\!]_T\{S\} \triangleq \begin{cases} S & \text{if } \forall x, L_S^T(x) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\texttt{flush}\ x]\!]_T\{S\} \triangleq \begin{cases} \emptyset & \text{if } L_S^T(x) = 0 \\ [\![\mathit{drop}\ x^T_{L_S^T(x)}]\!] \circ [\![x^{mem} := x^T_{L_S^T(x)}]\!]\{S\} & \text{if } L_S^T(x) \geq 1 \end{cases}$$

$$\forall X \in \mathscr{D}, [\![\mathit{stmt}]\!]_T X \triangleq \bigcup_{S \in X} [\![\mathit{stmt}]\!]_T\{S\}$$

Figure 5: Concrete interleaving semantics in PSO.

$$\forall S \in \mathscr{S}, \forall T \in \mathit{Thread}, [\![\texttt{flush}\ x]\!]_T \circ [\![op_{\not x}]\!]S = [\![op_{\not x}]\!] \circ [\![\texttt{flush}\ x]\!]_T S$$

Proof. We consider S as a numerical point, each variable being a dimension in the state space. We distinguish two cases:

Case 1: $L_S^T(x) = 0$. $[\![\texttt{flush}\ x]\!]_T S = \emptyset$, thus $[\![op_{\not x}]\!]([\![\texttt{flush}\ x]\!]_T S) = \emptyset$. $[\![op_{\not x}]\!]$ does not add any entry to the buffer of x and T, since $[\![x := e]\!]$ is the only operator that does it. Therefore $L_S^T([\![op_{\not x}]\!]S) = 0$, which implies $[\![\texttt{flush}\ x]\!]_T([\![op_{\not x}]\!]S) = \emptyset$.

Case 2: $L_S^T(x) > 0$. $[\![op_{\not x}]\!]$ does not modify the value of $x^T_{L_S^T(x)}$, and does not use the value of the dimension x^{mem}. Therefore $[\![x^{mem} := x^T_{L_S^T(x)}]\!]$ commutes with $[\![op_{\not x}]\!]$. $[\![op_{\not x}]\!]$ does not use the value of $x^T_{L_S^T(x)}$ either, therefore $[\![op_{\not x}]\!]$ also commutes with $[\![\mathit{drop}\ x^T_{L_S^T(x)}]\!]$. Chaining both commutations makes $[\![op_{\not x}]\!]$ commute with $[\![\texttt{flush}\ x]\!]_T$. □

This flush commutation allows us to avoid computing the flush of each variable from each thread at each control state, and to compute only the flush of the variables that have been affected by the statements leading to this control state. Specifically, when computing the result of an edge labelled with $[\![op_x]\!]_T$ (where $[\![op_x]\!]$ denotes an operator that reads from or writes to the *Shared* variable x) from a concrete element X, we do not only compute $[\![op_x]\!]_T X$, but:

$$[\![\texttt{flush}\ x]\!]^* \circ [\![op_x]\!]_T X$$

where :

$$[\![\texttt{flush}\ x]\!]^* X \triangleq \mathrm{lfp}(\lambda Y. X \cup \bigcup_{T \in \mathit{Thread}} [\![\texttt{flush}\ x]\!]_T Y)$$

That is, we compute the result of a *closure by flush* after applying the operator. Note that flushes are computed from all threads, not only the one performing $[\![op_x]\!]$. The lemma states that no other flush is needed. The result $\mathcal{R} : \mathscr{C} \to \mathscr{D}$ of the analysis can be stated as a fixpoint on the product control graph:

$$\widetilde{[\![op]\!]_T} : \mathscr{D} \to \mathscr{D} \triangleq \lambda X. \begin{cases} [\![\texttt{flush } x]\!]^* \circ [\![op]\!]_T X & \text{if } [\![op]\!] \text{ acts on } x \in \mathit{Shared} \\ [\![op]\!]_T X & \text{otherwise} \end{cases}$$

$$R_0 : \mathscr{C} \to \mathscr{D} = \lambda c.\ \texttt{if } c \texttt{ is initial then } \top \texttt{ else } \bot$$

$$\mathcal{R} = \text{lfp } \lambda R. R_0 \cup \left(\lambda c. \bigcup_{c' \xrightarrow{op}_T c \text{ edges}} \widetilde{[\![op]\!]_T} R(c') \right)$$

This property will prove itself even more useful when going into modular analysis.

Remark 2. As long as we stay in the concrete domain, this computation method has no effect on precision. However, this is no longer necessarily true when going into the abstract, and we found this method to be actually more precise on some examples: the flush abstract operator may induce information loss, and the new method performs less flush operations, thus retaining more precision.

2.2 Modular Concrete Semantics

We rely on Miné's interference system [15] to elaborate a thread-modular semantics from the monolithic previous one, as well as a computation method.

Transition Systems. The interference-based semantics can be expressed in the most general way when resorting to *labelled transition systems* rather than to equation systems (that are described by the control graph based analysis). We follow Cousot and Cousot [5] and express the transition system associated to our concurrent programs as a set $\Sigma = \mathscr{C} \times \mathscr{S}$ of *global states*, a set $I \subseteq \Sigma$ of *initial states*, and a *transition relation* $\tau \subseteq \Sigma \times \mathit{Thread} \times \Sigma$. We write $\sigma \xrightarrow{T}_\tau \sigma'$ for $(\sigma, T, \sigma') \in \tau$, which denotes that executing a step from thread T updates the current global state σ into the state σ'. We refer to Cousot [5] and Miné [15] for the formal definition of such a system, which is quite standard.

The semantics of this transition system specifies that a global state σ is reachable if and only if there exists a finite sequence of states $\sigma_1, ..., \sigma_n$ and some (not necessarily different) threads $T_\alpha, T_\beta, ..., T_\psi, T_\omega \in \mathit{Thread}$ such that $I \xrightarrow{T_\alpha}_\tau \sigma_1 \xrightarrow{T_\beta}_\tau ... \xrightarrow{T_\psi}_\tau \sigma_n \xrightarrow{T_\omega}_\tau \sigma$.

Local States. The monolithic transition system uses as global states a pair of a global control information in $\mathscr{C}$ and a memory state in $\mathscr{S}$. The modular transition system defines the local states of a thread T by reducing the control part to that of T only. By doing so, one retrieves a semantics that has the same structure as when performing a sequential analysis of the thread. However, the control information of the other threads is not lost, but kept in auxiliary variables

$pc_{T'}$ for each $T' \in Thread, T' \neq T$. This is needed for staying complete in the concrete, and useful to remain precise in the abstract world. We denote by $\mathscr{S}_T$ the states in $\mathscr{S}$ augmented with these $pc_{T'}$ variables. Local states of $T \in Thread$ thus live in $\Sigma_T = \mathbb{L} \times \mathscr{S}_T$. We define the domain $\mathscr{D}_T \triangleq \mathcal{P}(\mathscr{S}_T)$.

Interferences. Interferences model interaction and communication between threads. The interferences set $\mathscr{I}_T$ caused by a thread T are transitions produced by T: $\mathscr{I}_T \triangleq \left\{ \sigma \xrightarrow{T}_\tau \sigma' \in \tau \mid \sigma \textit{ is a state reachable from } I \right\}$.

Computation Method. The method for computing an interference modular semantics works with two least fixpoint iterations:

- The inner fixpoint iteration computes, for a given interference set, the local states result of a thread. It also produces the interferences set generated by this thread executions. It will ultimately compute the program state reachability, one thread at a time.
- The outer fixpoint iteration computes fully the inner fixpoint, using the generated interferences from one inner analysis as an input of the next one. It goes on, computing the inner fixpoint for each thread at each iteration, until the interferences set is stabilised with increasing sets of interferences starting from an empty set.

The outer least fixpoint computation is a standard run-until-stabilisation procedure. The inner fixpoint is alike sequential program fixpoint computation, with the specificity of interference that we will describe. We refer to Miné [15] for the complete development on general transition systems, while we focus here on the specific case of the language of Section 2.1 under weak memory models.

This analysis method is thread modular in the sense that it analyses each thread in isolation from other thread code. It must still take into account the interferences from other threads to remain sound. Furthermore, this is a constructive method: we infer the interference set from scratch rather than relying on the user to provide it. This is why we need to iterate the outer fixpoint computation as opposed to analysing each thread separately only once. Practically, we observe that the number of outer iterations until stabilisation is very small (less than 5) on typical programs.

Let us consider the graph representation of the inner fixpoint computation. As already stated, it takes an interference set as an input and works like a sequential program analysis, except when computing the result of an edge transfer operation, the analyser also uses the origin and the resulting local states to build an interference corresponding to the transition associated to the edge. As $\mathscr{S}_T$ holds the control information about other threads, as long as we stay in the concrete domain, all the information needed to build this interference is available. The analyser also needs to take into account the transition from other threads: this is done through an interference application phase that can be performed just after computing the local state attached to a vertex. Amongst all interferences,

the analyser picks the ones whose origin global state is compatible with the current local state (which means they model transitions that can happen from this local state); then it updates the local state, adding the destination global states of these interferences as possible elements.

On a thread analysis with a SC model, these two phases are well separated: first, a *generation* phase computes a destination state as well as generated interferences. Then the analyser joins the destination states from all incoming vertices to get the resulting current state at the current label. After this, the *application* phase applies candidate interferences, and the fixpoint engine can move to the next vertex to be computed. However, it works differently in a relaxed memory setting, due to flush self-loop edges: one wants to avoid useless recomputations of incoming edges by computing a flushing fixpoint before applying interferences. These flushes generate interferences themselves, that must be taken into account.

Yet we showed earlier, for the monolithic analysis, that it was equivalent to compute flushes only when needed (which is more efficient), that is after operations on the same variable, with which they do not commute. This works the same way in modular analyses: when applying interferences from other threads, one can in particular apply interferences that interact with a variable in the shared memory. These applications do not commute with flushes of this variable: therefore, one must close by flush with respect to a variable after applying interferences that interact with this variable.

3 Abstract Semantics

3.1 Abstracting Local States

We abstract the local state of a thread T in a similar way to our previous work [20]. We first forget the variables that represent the buffer entries from other threads than T (but we keep their local variables). We define in Figure 6 this abstraction. The intuition behind this projection is that these entries are not observable by the current thread, yet it will still be aware of them once they are flushed, because they will be found in the accessible shared memory. As a consequence, forgetting them is an abstraction that can lose precision in the long run, but it is necessary for scalability.

We then partition the states with respect to a partial information, for each variable, on the length of the corresponding buffer: either it is empty (we note this information 0), or it contains exactly one entry (we note this 1), or it contains more than one (we note this 1+). The partitioning function, δ_T, is given in Figure 7a. We use the notation $L_S^T(x)$ as the length of the buffer of the variable x for the thread T in the state S.

We use a state partitioning abstraction [5] with respect to this criterion, the resulting domain being defined in Figure 7b. We recall that the partitioning itself does not lose any information: $\underset{\alpha_p}{\overset{\gamma_p}{\leftleftarrows}}$ is a Galois isomorphism.

The intuition behind this particular partitioning is twofold: first, since our operations behave differently depending on the buffer lengths, we regroup together

$$\mathscr{S}^\sharp_T \triangleq \prod_{\substack{T' \in Thread \\ T' \neq T}} pc_{T'} \times Mem \times TLS \times \prod_{x \in Shared} Buf^T_x \qquad \mathscr{D}^\sharp_T \triangleq \mathcal{P}(\mathscr{S}^\sharp_T)$$

$$\mathscr{D}_T \underset{\alpha_\pi}{\overset{\gamma_\pi}{\leftrightarrows}} \mathscr{D}^\sharp_T$$

$$\gamma_\pi(X^\sharp_T) \triangleq \left\{ PC, M, S, (T,x) \mapsto B^T_x \in \mathscr{S} \mid PC, M, S, x \mapsto B^T_x \in X^\sharp_T \right\}$$

$$\alpha_\pi(X) \triangleq \left\{ PC, M, S, x \mapsto B^T_x \in \mathscr{D}^\sharp_T \mid PC, M, S, (T,x) \mapsto B^T_x \in X \right\}$$

Figure 6: Forgetting other threads buffers as a first abstraction.

$$\mathscr{B}^\flat \triangleq Shared \to \{0; 1; 1+\} \qquad \text{Abstract buffer lengths}$$

$$\forall T \in Thread, \delta_T : \mathscr{S}^\sharp_T \to \mathscr{B}^\flat$$

$$\delta_T(S^\sharp_T) \triangleq \lambda x. \begin{cases} 0 & \text{if } L^T_{S^\sharp_T}(x) = 0 \\ 1 & \text{if } L^T_{S^\sharp_T}(x) = 1 \\ 1+ & \text{if } L^T_{S^\sharp_T}(x) > 1 \end{cases} \qquad \text{State partitioning criterion}$$

(a) A partial information on states buffers

$$\mathscr{D}^\sharp_T \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} \left(\mathscr{B}^\flat \to \mathscr{D}^\sharp_T \right)$$

$$\alpha_p(X^\sharp_T) \triangleq \lambda b^\flat. \left\{ S^\sharp_T \in X^\sharp_T \mid \delta_T(S^\sharp_T) = b^\flat \right\}$$

$$\gamma_p(X^{part\sharp}_T) \triangleq \left\{ S^\sharp_T \in \mathscr{S}^\sharp_T \mid S^\sharp_T \in X^{part\sharp}_T(\delta_T(S^\sharp_T)) \right\}$$

(b) The state partitioning abstract domain.

Figure 7: State-partitioning w.r.t. an abstraction of buffer lengths

the states with the same abstract lengths in order to get uniform operations on each partition; second, each state in every partition defines the same variables (including buffer variables, as explained in Remark 1), thus the numerical abstraction presented later will benefit from this partitioning: we can use a single numeric abstract element to represent a set of environments over the same variable and buffer variable set.

The next step uses the summarisation technique described by Gopan et al. [6] In each partition, we separate the variables $x^T_2 ... x^T_N$ (up to the size N of the buffer for x in T) from x^T_1 and regroup the former into a single summarised variable x^T_{bot}. The summarisation abstraction is then lifted partition-wise to the partitioned states domain to get a final summarised and partitioned abstract domain. This domain is used through a Galois connection $\mathscr{D}^\sharp_T \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \mathscr{D}^{Sum}_T$, as defined by Gopan et al. [6]

Abstracting the Control. We also need to develop a new abstraction for the control part of the local states. This control part was not present in the states of the original monolithic semantics [20], which iterated its fixpoint over an explicit product control state. The superiority of the thread-modular analysis lies in the possibility of choosing the control abstraction to be as precise or fast as one wants. In particular, one can emulate the interleaving analysis (the concrete modular semantics being complete).

Several control representations have been proposed by previous authors [14,16]. Our domain is parametric in the sense that we can choose any control abstraction and plug it into the analysis. However, we tried a few ones and will discuss how they performed as well as our default choice.

No abstraction. The first option is to actually not abstract the control. This emulates the interleaving analysis.

Flow-insensitive abstraction. This abstraction [14] simply forgets the control information about the other threads. The intra-thread analysis remains flow-sensitive regarding the thread itself. Albeit very fast, this is usually too imprecise and does not allow verifying a wide variety of programs.

Control-partitioning abstraction. This technique was explored in sequential consistency by Monat and Miné [16] and consists in selecting a few abstract labels that represent sets of labels, and only distinguishing between different abstract labels and not between two labels mapped to the same abstract label. This is a flexible choice since one can modulate the precision of the analysis by refining at will the abstraction. In particular, one can retrieve the flow-insensitive abstraction by chosing a single abstract label, and the precise representation by mapping each concrete label to itself.

We settled on the general control-partitioning abstraction and manually set our locations program by program. Additional work is needed to propose an automatic method that is both precise enough and does not add too many abstract labels that slow down the analyses.

Formally, we define for each thread T a partition $L_T^\sharp$ of the control points in $\mathbb{L}$ of T. Consider the program of Figure 2. The partition that splits after the critical section end is $L_T^\sharp = \{[1 \,..\, l_1]\,; [l_2 \,..\, end]\}$. Note that this partition does not formally need to be composed of intervals. Once this partition is defined, we denote as $\dot{\alpha}_{\mathbb{L}_T} : \mathbb{L} \to L_T^\sharp$ the mapping from a concrete control label to the partition block to which it belongs: for instance, with the previous example, $\dot{\alpha}_{\mathbb{L}_T}(l_{crit\ start}) = [1 \,..\, l_1]$. With no abstraction, $L_T^\sharp = \mathbb{L}$ and $\dot{\alpha}_{\mathbb{L}_T} = \lambda l.l$, and with a flow-insensitive abstraction, $L_T^\sharp = \{\top\}$ and $\dot{\alpha}_{\mathbb{L}_T} = \lambda l.\top$.

Numerical Abstraction. We eventually regroup the original thread state and the control parts of the local state in a numerical abstraction. Since control information can be represented as an integer, this does not change much from the non-modular abstraction. The partitioning has been chosen so that every summarised state in the same partition defines the same variables (in particular, the buffer ones x_1^T and x_{bot}^T). Thus a well-chosen numerical abstraction can be

$$\mathscr{D}_T^\natural \triangleq \mathscr{B}^\flat \to \mathscr{D}^N \qquad \gamma : \mathscr{D}_T^\natural \to \mathscr{D}_T$$
$$\gamma(X_T^\natural) \triangleq \left\{ S \in \mathscr{S}_T \mid S \in \gamma_\pi \circ \gamma_S \circ \gamma_N(X_T^\natural(\delta(S))) \right\}$$

Figure 8: Final local states abstraction

applied directly to each partition. This abstraction will be denoted as the domain $\mathscr{D}^N$, and defined by a concretisation γ_N (since some common numerical domains, such as polyhedras, do not possess an abstraction α_N that can be used to define a Galois connection).

Our analysis is parametric w.r.t. the chosen numerical abstraction: one can modulate this choice to match some precision or performance goal. In our implementation, we chose numerical domains that allowed us to keep the control information intact after partitioning, since it was usually required to prove our target programs. Namely, we used the Bddapron [9] library, which provides logico-numerical domains implemented as numerical domains (such as octagons or polyhedras) on the leaves of decision diagrams (which can encode bounded integers, therefore control points, with an exact precision). As control information is a finite space, this does not affect the calculability of the semantics.

The resulting global composed domain is recapped in Figure 8. For convenience, we consider the $\dot{\gamma}_{\mathbb{L}_T}$ concretisation of abstract domains to be integrated to the γ_N definition of the numerical final abstraction, since both are strongly linked.

3.2 Abstracting Interferences

We recall that interferences in the transition system live in $\Sigma \times \mathit{Thread} \times \Sigma$. They are composed of an origin global state, the thread that generates them, and the destination global state. We group interference sets by thread: one group will thus be an abstraction of $\mathcal{P}(\Sigma \times \Sigma)$. We represent the control part of Σ as a label variable pc_T for each $T \in \mathit{Thread}$.

To represent pairs in $\Sigma \times \Sigma$, we group together the origin and the destination global states in a single numerical environment. We use the standard variable names for the origin state, and use a *primed* version v' of each variable v for the destination domain. This is a common pattern for representing input-output relations over variables, such as function application.

We then apply the same kind of abstractions as in local states: we forget every buffer variable of every thread (including the thread indexing each interference set), and we abstract the control variables of each thread, using the same abstraction as in local states, which is label partitioning.

We partition the interferences with respect to the shared variable they interact with (which can be None for interferences only acting on local variables). This allows us to close-by-flush after interference application considering only the shared variables affected, as we discussed in Section 2.2.

After doing that, we use a numerical abstraction for each partition. Although one could theoretically use different numerical domains for local states and interferences, we found that using the same one was more convenient: since interference application and generation use operations that manipulate both local states and interferences (for instance, interferences are generated from local states, then joined to already existing interferences), it is easier to use operations such as join that are natively defined rather than determining similar operators on two abstract elements of different types.

3.3 Abstract Operations

Operators for computing local states and generating interferences can be derived from our abstraction in the usual way: we obtain the corresponding formulas by reducing the equation $f^\sharp = \alpha \circ f \circ \gamma$. The local state ones are practically identical to the monolithic ones [20], we will not restate them here.

We express in Figure 9 the resulting interference generation operators for flush and shared memory writing. The local state transfer operators are almost the same as in non-modular abstract interpretation, and the other interference generators follow the same general pattern as these two, so we did not write them for space reasons. $\mathscr{D}_T^\sharp$ is the abstract domain of local states, and $\mathscr{I}^\sharp$ are abstract interferences. $\rhd$ denotes function application ($x \rhd f \rhd g$ is $g(f(x))$). We write ${}^{l_1}[\![stmt]\!]_T^{\sharp l_2}$ for the application of the abstract operator $[\![stmt]\!]_T^\sharp$ between control labels l_1 and l_2. Note that l_1 and l_2 are concrete labels (linked to the location of the statement *stmt* in the source program, and the corresponding control graph vertices).

We draw the attention of the reader on the $[\![x := r]\!]_T$ interference generator: it does only update the control labels of T. Indeed, the performed write only goes into T's buffer, which is not present in the interferences. The actual write to the memory will be visible by other threads though the flush interference, that will be generated later (during the flush closure).

We refer to Monat and Miné [16] for the interference *application* operator, that does not change from sequential consistency (the difference being that after using *apply*, one will close by *flush*).

Soundness. The soundness proof of this analysis builds upon two results: the soundness of the monolithic analysis [20], and the soundness of the concrete interference analysis [15]. Our pen-and-paper proof is cumbersome, hence we will simply explain its ideas: first, we already gave a formal soundness proof for the monolithic abstract operators [20]. Our local operators being almost the same, their soundness proof is similar. Miné [15] also shows that the interference concrete analysis is both sound and complete. We show that our interference operators soundly compute both the control and the memory part of the concrete transitions: the control part only maps a label to its abstract counterpart, and the memory part also stems from the monolithic analysis.

$$Var \triangleq Shared \cup Local \cup \{pc_T \mid T \in Thread\}$$

$$extend_T \triangleq \begin{cases} \mathscr{D}_T^\sharp \to \mathscr{I}^\sharp \\ X^\sharp \mapsto add(X^\sharp, \{v' \mid v \in Var\} \cup \{pc_T\}) \end{cases}$$

$$x \in Shared \qquad T \in Thread \qquad r \in Local \qquad l_1, l_2 \in \mathbb{L}$$

$$[\![.]\!]^\sharp : \mathscr{D}_T^\sharp \to \mathscr{I}^\sharp$$

$${}^{l_1}[\![r := x]\!]_T^{\sharp l_2} X^\sharp =$$
$$extend_T(X^\sharp)$$
$$\rhd [\![\forall v \in Var, v' := v]\!]$$
$$\rhd [\![pc_T := \dot{\alpha}_{\mathbb{L}_T}(l_1)]\!] \circ [\![pc'_T := \dot{\alpha}_{\mathbb{L}_T}(l_2)]\!]$$
$$\rhd [\![r' := x_1^T \ \texttt{if}\ L_S^{T\flat} = 1\ \texttt{else}\ x^{mem}]\!]$$
$$\rhd [\![\forall y \in Shared, drop \left\{y_1^T, y_{bot}^T\right\}]\!]$$

$${}^{l_1}[\![x := r]\!]_T^{\sharp l_2} X^\sharp =$$
$$extend_T(X^\sharp)$$
$$\rhd [\![\forall v \in Var, v' := v]\!]$$
$$\rhd [\![pc_T := \dot{\alpha}_{\mathbb{L}_T}(l_1)]\!] \circ [\![pc'_T := \dot{\alpha}_{\mathbb{L}_T}(l_2)]\!]$$
$$\rhd [\![\forall y \in Shared, drop \left\{y_1^T, y_{bot}^T\right\}]\!]$$

$$\exists L_S^{T\flat}(x) \neq 0 \implies {}^{l_1}[\![\texttt{mfence}]\!]_T^{\sharp l_2} X^\sharp = \bot^\sharp$$
$$\nexists L_S^{T\flat}(x) = 0 \implies {}^{l_1}[\![\texttt{mfence}]\!]_T^{\sharp l_2} X^\sharp =$$
$$extend_T(X^\sharp)$$
$$\rhd [\![\forall v \in Var, v' := v]\!]$$
$$\rhd [\![pc_T := \dot{\alpha}_{\mathbb{L}_T}(l_1)]\!] \circ [\![pc'_T := \dot{\alpha}_{\mathbb{L}_T}(l_2)]\!]$$
$$\rhd [\![\forall y \in Shared, drop \left\{y_1^T, y_{bot}^T\right\}]\!]$$

$$L_S^{T\flat}(x) = 0 \implies {}^{l_1}[\![\texttt{flush}\ x]\!]_T^{\sharp l_1} X^\sharp = \bot^\sharp$$
$$L_S^{T\flat}(x) \in \{1, 1+\} \implies {}^{l_1}[\![\texttt{flush}\ x]\!]_T^{\sharp l_1} X^\sharp =$$
$$extend_T(X^\sharp)$$
$$\rhd [\![\forall v \in Var, v' := v]\!]$$
$$\rhd [\![pc_T := \dot{\alpha}_{\mathbb{L}_T}(l_1)]\!] \circ [\![pc'_T := \dot{\alpha}_{\mathbb{L}_T}(l_1)]\!]$$
$$\rhd [\![x^{mem\prime} := x_1^T \ \texttt{if}\ L_S^{T\flat} = 1\ \texttt{else}\ x_{bot}^T]\!]$$
$$\rhd [\![\forall y \in Shared, drop \left\{y_1^T, y_{bot}^T\right\}]\!]$$

Figure 9: Abstract operators for interference generation.

4 Experimentations

We implemented our method and tested it against a few examples. Our prototype was programmed with the OCaml language, using the BDDApron logico-numerical domain library and contributing a fixpoint engine to ocamlgraph. Our experiments run on a Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz computer with 8 GB RAM. We compare against our previous work [20].

Improvements on Scaling. To test the scaling, we used the N-threads version of the program of Figure 2, and timed both monolithic and modular analyses when N increases. Results are shown in Figure 10. They show that the modular analysis does indeed scale better than the monolithic one: the performance ratio between both methods is exponential. However, the modular analysis still has an exponential curve, and is slower than in sequential consistency where it was able to analyse hundreds of threads of the same program in a couple of hours [16].

We believe this difference is mainly due to the fact that, in SC, adding a thread only adds so much code for the analyser to go through. This is not the

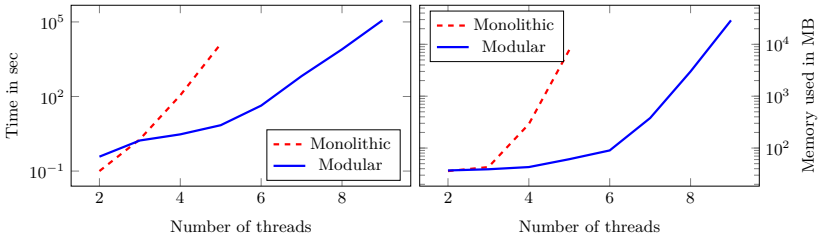

Figure 10: Scaling results.

Test	abp	concloop	kessel	dekker	peterson	queue	bakery
Non-modular	✓	✓	✓	✓	✓	✓	*timeout after days*
Modular	✓	✓	✗	✓	✓	✓	✗

Figure 11: Precision results on small programs.

case in relaxed models, where adding a thread also increases the size of program states, due to its buffers. Therefore the 8 threads version of the program has not only 4 times as much code to analyse than the 2 threads version, but this code also deals with a 4 times bigger global state: the analysis difficulty increase is twofold, leading to a greater analysis time augmentation.

Testing the precision. Modular analysis, after abstraction, provides a more scalable method than a monolithic one. This comes at a cost: the additional abstraction (for instance on control) may lead to precision loss. To assess this precision, we compare with our previous results [20] in Figure 11.

The analysis of these programs aims to check safety properties expressed as logico-numerical invariants. These properties mostly are mutual exclusions: at some program points (the combinations of at least two thread critical section control points), the abstraction should be $\bot$ (or the property `false` should hold).

The modular analysis was able retain the needed precision to prove the correctness of most of these programs, despite the additional abstraction. However, it does fail on two tests, `kessel` and `bakery`. We believe that it could also pass these ones with a better control partitioning, but our heuristics (see the next paragraph) were not able to determine it.

Note that `bakery` is significantly bigger than the other examples. Although our analysis could not verify it, it did finish (in a few minutes with the most aggressive abstractions), whereas the non-modular one was terminated after running for more than a day. This is not a proper scaling improvement result due to the failure, but it is worth noticing.

All the programs certified correct by our analysis are assumed to run under the PSO model. Yet some programs may be correct under the stronger TSO model but not under PSO: for instance, one can sometimes remove some fences (between two writes into different locations) of a PSO valid program and get

a TSO (but no longer PSO) valid program. Our prototype will not be able to check these TSO programs, since it is sound w.r.t. PSO.

Our main target being TSO, this can be a precision issue, which one can solve by adding additional fences. However, we observed that all those tests, except `peterson`, were validated using the minimal set of fences for the program to be actually correct under TSO; this validates our abstraction choice even with TSO as a target. We already proposed a method to handle TSO better by retrieving some precision [20]: this technique could also be implemented within our modular framework if needed.

Leveraging our Method in Production. For realistic production-ready analyses, one should likely couple this analysis with a less precise, more scalable one, such as a non-relational or flow-insensitive one [11,14]. The precise one should be used on the small difficult parts of the programs, typically when synchronisation happens and precision is needed to model the interaction between threads. Then the scaling method can be used on the other parts, for instance when threads do large computations without interacting much. As, to be scalable, a concurrent program analysis must be thread-modular anyway, we also believe this analysis lays a better ground for this kind of integration than a monolithic one.

We also recall that our method requires the user to manually select the control abstraction. The control partition is specified by adding a `label` notation at chosen separation points. Most of the time, partitioning at loop heads is sufficient. We believe this could be fully automated but are not able to do it yet. Practically, we found that few trials were needed to find reasonably good abstractions: putting label separations on loops heads and at the control point where the properties must be check was often more than enough. An automatic discovery of a proper control partitioning is left to future work and would be an important feature of a production-ready analyser.

Finally, real-life complex programs feature some additional traits that are not part of our current semantics. Some, such as pointer and heap abstraction or shape analysis, are orthogonal to our work: dedicated domains can be merged with ours to modelise it. Others are linked to the concurrency model, such as atomic operations like `compare-and-swap` and `lock` instructions. The former could be quickly added to our analyser: one needs to evaluate the condition, conditionally perform the affectation, and flush the memory (like `mfence` would); all this without generating or applying interferences inbetween. The latter could also be added with a little more work: the idea would be to generate interferences abstracting a whole `lock/unlock` block transition instead of individual interferences for each statement in the block.

5 Related Work

Thread-modular and weak memory analyses has been investigated by several authors [1, 2, 4, 7, 8, 10, 13, 15–17], yet few works combine both. Nonetheless, it was shown [3, 14] that non-relational analyses that are sound under sequential

consistency remain sound under relaxed models. Thus some of these works can also be used in a weakly consistent memory environment, if one accepts the imprecision that comes with non-relational domains. In particular, Miné [14] proposes a sound yet imprecise (flow-insensitive, non-relational) analysis for relaxed memory.

Ridge [18] has formalised a rely-guarantee logics for x86-TSO. However, his work focuses on a proof system for this model rather than static analysis. Therefore he proposes an expressive approach to express invariants, which is an asset for strong proofs but is less practical for a static analyser which abstracts away this kind of details to build a tractable analysis.

Kusano et al. [11] propose a thread-modular analysis for relaxed memory models, including TSO and PSO. They rely on quickly generating imprecise interference sets and leverage a Datalog solver to remove interferences combinations that can be proved impossible. However, unlike ours, their interferences are not strongly relational in the sense that they do not hold control information and do not link the modification of a variable to its old value. Thus this method will suffer from the same kind of limitations as Miné's flow insensitive one [14].

6 Conclusion

We designed an abstract interpretation based analysis for concurrent programs under relaxed memory models such as TSO that is precise and thread-modular. The specificity of our approach is a relational interference abstract domain that is weak-memory-aware, abstracting away the thread-specific part of the global state to gain performance while retaining enough precision through partitioning to keep the non-deterministic flush computation precise. We implemented this approach, and our experimental results show that this method does scale better than non-modular analysis with no precision loss. We discussed remaining scalability issues and proposed ways to solve them in a production analyser.

Future work should focus on more relaxed memory models such as POWER and C11. We believe that interference-based analysis lays a solid ground to abstract some of these model features that are presented as communication actions between threads. However, besides being more relaxed, these models are also significantly more complex and some additional work needs to be done to propose abstractions that reduce this complexity to get precise yet efficient analyses.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, pages 134–156. Springer, 2016.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*, pages 308–332. Springer, 2015.

3. Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. Soundness of data flow analyses for weak memory models. In *Programming Languages and Systems*, pages 272–288. Springer, 2011.
4. Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proceedings of the ACM on Programming Languages*, 1(1), 2018.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
6. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529. Springer, 2004.
7. Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *ACM SIGPLAN Notices*, volume 42, pages 266–277. ACM, 2007.
8. Lukáš Holík, Roland Meyer, Tomáš Vojnar, and Sebastian Wolff. Effect summaries for thread-modular analysis. In *SAS*, pages 169–191. Springer, 2017.
9. Bertrand Jeannet. The BDDApron logico-numerical abstract domains library, 2009.
10. Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 799–809. ACM, 2016.
11. Markus Kusano and Chao Wang. Thread-modular static analysis for relaxed memory models. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 337–348. ACM, 2017.
12. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.
13. Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Iterated process analysis over lattice-valued regular expressions. In *PPDP*, pages 132–145. ACM, 2016.
14. Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *ESOP*, volume 11, pages 398–418. Springer, 2011.
15. Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *VMCAI*, pages 39–58. Springer, 2014.
16. Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *VMCAI*, pages 386–404. Springer, 2017.
17. Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. Thread-local semantics and its efficient sequential abstractions for race-free programs. In *SAS*, pages 253–276. Springer, 2017.
18. Tom Ridge. A rely-guarantee proof system for x86-TSO. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 55–70. Springer, 2010.
19. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.
20. Thibault Suzanne and Antoine Miné. From array domains to abstract interpretation under store-buffer-based memory models. In *SAS*, pages 469–488. Springer, 2016.