# From Array Domains to Abstract Interpretation Under Store-Buffer-Based Memory Models

Thibault Suzanne, Antoine Miné

# From Array Domains to Abstract Interpretation under Store-Buffer-Based Memory Models [⋆]

Thibault Suzanne[1] and Antoine Miné[2]

[1] École Normale Supérieure, PSL Research University; CNRS; Inria
`thibault.suzanne@ens.fr`
[2] Sorbonne Universités, UPMC Univ Paris 6, Laboratoire d'informatique de Paris 6 (LIP6)
`Antoine.Mine@lip6.fr`

**Abstract** We address the problem of verifying concurrent programs under store-buffer-based weakly consistent memory models, such as TSO or PSO. Using the abstract interpretation framework, we adapt existing domains for arrays to model store buffers and obtain a sound abstraction of program states (including the case of programs with infinite state space) parameterised by a numerical domain. Whereas the usual method for this kind of programs implements a program transformation to come back to an analysis under a sequentially consistent model, the novelty of our work consists in applying abstract interpretation directly on the source program, setting a clean foundation for special dedicated domains keeping information difficult to express with program transformations. We demonstrate the precision of this method on a few examples, targetting the TSO model and incidentally being also sound for PSO due to some specific abstraction choice. We discuss an application to fence removal and show that our implementation is usually able to remove as many or more fences, with respect to the state of the art, on concurrent algorithms designed for sequential consistency while still remaining precise enough to verify them.

## 1 Introduction

Multicore architectures have become increasingly important in the performance race of computers, hence concurrent programming is nowadays a wide-spread technique. Its most abundant paradigm is shared memory: all threads run simultaneously on different cores, and they communicate by accessing the same locations in the common memory.

The intuitive execution model of these concurrent programs is the *sequential consistency* [3] [16]. It states that the possible executions of a concurrent program correspond to sequential executions of the interleavings of its threads. However, for optimisation reasons, modern processors or language implementations do

⋆ This work is partially supported by the project ANR-11-INSE-014 from the French Agence nationale de la recherche.

[3] We will use the acronym SC, as it is usually done.

```
                  initial x = 0 && y = 0;

/* Thread 1 */          |  /* Thread 2 */
x = 1;                  |  y = 1;
r1 = y;                 |  r2 = x;
```

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

```
/* Property to check: At labels (bp0; bp1), tail < h1 */
                        int head;

/* ENQUEUE */           |  /* DEQUEUE */
int h1;                 |  int tail, h2;
                        |
while true {            |  tail = head;
    h1 = head;          |  while true {
    bp0:                |      h2 = head;
    h1 = h1 + 1;        |      while (tail >= h2) {
    head = h1;          |          h2 = head;
}                       |      }
                        |      bp1:
                        |      tail = tail + 1;
                        |  }
```

**Figure 2.** A program with unbounded buffers and variable value space

not stick to this description, but add some additional possible behaviours. For instance, after the execution on a `x86` CPU of the program described in Figure 1, it is possible that both registers `r1` and `r2` are equal to `0`, because memory writes are buffered and Thread 1 can read `y` to `0` even after Thread 2 has executed the assignment `y=1`.[4]

The example program of Figure 1 is actually quite simple: the value space of the variables is small (four variables, each one can be equal to `0` or `1`), and the size of the write buffers is bounded by a known and computable limit (that is 1). However, the program described in Figure 2 is much more complicated: none of these properties holds anymore. There are infinitely many reachable states, and unbounded and nested loops. Furthermore, the actual execution model adds again some possible behaviours, with unbounded write buffers. This makes it even harder to reason on its correctness.

Weakly consistent memory models [1] aim to formally describe these additional behaviours in addition to the sequentially consistent ones. However, as this complexity adds to the inherent difficulties of writing correct concurrent programs, automatic verification becomes more and more useful.

[4] The result set of this program is $x = 1, y = 1, r1 \in \{0, 1\}, r2 \in \{0, 1\}$. Most static analyses are able to infer it exactly, and ours will too.

Several works [2,3,4,5,10,14,15,17] have been done in this area, using various static analysis techniques and targetting different memory models (see Section 6). Our work focuses on abstract interpretation [8], by giving an abstract semantic of concurrent programs which takes into account the relaxed memory effects of the TSO model. Amongst works that use this abstract interpretation framework, the usual method consists in applying a transformation on the source program to obtain another program which exhibits the same behaviour when run under sequential consistency. This resulting program is then verified with existing SC analysers.

This paper describes a general method to adapt existing array abstractions in order to represent the buffer part of the state of a concurrent program. Contrary to existing work, this method applies abstract interpretation directly on the source program. By doing so, we set a clean foundation for special dedicated domains keeping information difficult to express with program transformations. In particular, our analysis is able to prove the program in Figure 2 correct, which is impossible with the state of the art since existing analyses are limited to bounded write buffers or programs which are finite-state when they run under SC.

After describing in Section 2 a concrete operational semantics of programs running under the chosen memory model, Section 3 presents the formalisation of an instance of our abstract domain using the summarisation approach from Gopan et al. [11] Section 4 then describes how to formalise more precise and complex abstractions. In Section 5, we give some experimental results obtained with an implementation of the first instance. We finally discuss a comparison with related works in Section 6. Section 7 concludes.

## 1.1 Weak Memory Models

We focus on two similar weak memory models, TSO and PSO, for *total* (resp. *partial*) *store ordering*. TSO is especially known to be the base memory model of the `x86` CPU family [19]. In general, weak memory models can be described in several ways, e.g. by giving axiomatic (based for instance on event reordering) or operational semantics. We will work with the latter.

The operational model of TSO adds to the usual registers (or local variables) and shared memory of the sequentially consistent concurrent programs a *buffer* for each thread of the program. These buffers act as unbounded FIFO queues for the writes issued by the corresponding thread: when some thread executes an instruction such as `x := n` for some shared variable `x`, the value `n` does not reach the memory immediately, but instead the binding `x= $\mapsto$ =n` is pushed into the buffer. Conversely, when a thread attempts to read the value of some shared variable `x`, it effectively reads the value corresponding to the most recent binding of `x` in its own buffer. If this buffer contains no entry for `x`, then its value is read from the memory: a thread cannot directly access the buffer of another thread.

At any time between the execution of two instructions (which we consider as being atomic), the oldest binding of a buffer can be *flushed*: it disappears from the buffer and its value is used to update the memory at the corresponding variable location. The `mfence` specific instruction also flushes every entry of the buffer, enforcing consistency with memory updates.

This model therefore ensures that two consecutive writes issued by a thread cannot be seen in the opposite order by another one. It also ensures that when a write issued by a thread is visible to another one, then it is visible to every other thread with no corresponding entry in its buffer.

PSO works almost the same way, except that there is not a single buffer per thread, but a buffer per thread per shared variable — or equivalently, entries concerning different variables in the buffer can be reordered before they reach the memory. The `mfence` instruction will flush every buffer of the thread which executes it.

We now illustrate the behaviour of buffers by describing a concrete example of an execution of the program in Figure 2. We first note that as it involves only one shared variable `head`, TSO and PSO are equivalent for this program: there is only one buffer used (for thread `ENQUEUE`, as thread `DEQUEUE` does not write anything to `head`), which we will denote as the list of the values written in it (most recent first). Let us set the initial value of `head` for this particular execution to 0.

– After one iteration of the loop of the `ENQUEUE` thread, `h1` is equal to `1`, `head` is still equal to `0` in the shared memory, but the buffer of the `ENQUEUE` thread contains one entry for `head`: `{1}`.
– Then thread `DEQUEUE` executes `tail = head`: it has no entry for `head` in its buffer, thus it reads from the memory, and `tail` is now equal to `0`.
– Thread `ENQUEUE` starts a new iteration. It executes `h1 = head` and reads the most recent entry in its buffer: `h1` is equal to `1`.
– Thread `ENQUEUE` finishes its iteration: its buffer now contains `{2, 1}` and `h1` is equal to `2`.
– Thread `ENQUEUE` starts a new iteration and executes `h1 = head`: `h1` reads from the buffer and is now equal to `2`.
– Thread `ENQUEUE` flushes one entry from its buffer: it now contains `{2}` and `head` is equal to `1`. It reaches label `bp0`.
– Thread `DEQUEUE` enters its outer loop. It executes `h2 = head`, which still reads from the memory. It can now read the write from thread `ENQUEUE` which has been flushed: `h2` is equal to `1`.
– `tail` is equal to `0` and `h2` is equal to `1`: thread `DEQUEUE` does not enter the inner loop and reaches label `bp1`.
– `tail` is equal to `0`, `h1` is equal to `2`: the property holds, but we only tested it for a particular reachable state. Our analysis will be able to prove it for every possible execution path.

For three main reasons, our main target is TSO:

– It is a “real-life” model, corresponding to `x86` processor architecture, whereas PSO is mainly of theoretical interest.

– It fills a sweet spot in the relaxation scale: most algorithms designed with sequential consistency in mind become incorrect, yet one can usually get back the desired behaviour with only a few modifications, such as fence insertion at selected points.
– It is a conceptually simpler and less permissive model than some other more relaxed ones like POWER, C11 or Java, which is a sensible property to lay the ground for works on the latter.

However, the abstraction choices made in our implementation happens to lose enough information to make our analysis also sound for PSO. For this reason, we mention and use this model in our formalisation. Yet we do not seek a precise abstraction of the special `sfence` instruction of PSO (which ensures that all stores preceding it will be flushed before the following ones), as it does not exist on our target model (if needed, it can soundly be abstracted by the identity). Section 4 will present abstractions able to exploit the additional precision of TSO (though we did not implement them).

## 1.2 Use of Array Abstractions

**Presentation of the Technique.** When using these operational models, buffers constitute the main difficulty for abstracting weakly consistent states: they behave as an unbounded FIFO queue whose size can change in a dynamic and non-deterministic way on virtually every execution step. We chose to use array abstractions to build computable abstract representations of these buffers.

On top of usual operations over fixed-size arrays, these abstractions must support (or be extended to support) an operation adding an element, and an operation removing one. Then we adapt them to go from arrays, which are fixed-size structures with immediate access to each element, to buffers, which behave mostly like FIFO queues whose size can change a lot during program execution, but are only accessed either at the beginning or the end, not at arbitrary positions.

A first and direct encoding consists in representing a buffer[5] $B$ of size $N$ with $N$ different variables $B_1, ..., B_N$. Adding an entry $E$ at the top of the buffer amounts to adding $B_{N+1}$, shifting all the variables $(B_{N+1} := B_N; ...; B_2 := B_1)$, and assigning $E$ to $B_1$. We also consider, for each shared symbol `x`, an abstract variable $x^{mem}$ which holds the memory value of `x`. This model is very concrete and results in unbounded buffers. To ensure termination, existing work enforce a fixed or pre-computed bound on the buffers, and the analysis fails if this bound is exceeded. As we will see shortly, our method does not have this limitation.

Dan et al. [10] propose another encoding of the buffers which does not need the shifting operation. The obtained analysis is usually more precise and efficient, but it suffers from the same limitation: buffer sizes must be statically bounded or the analysis may not terminate.

[5] In TSO, these buffers contain a tuple (variable name, variable value). In PSO, as there is one buffer for each variable, they simply contain integers. The abstract variables $B_i$ share the same type.

**Summarisation of Unbounded Buffers.** To get a computable abstraction which allows unbounded buffers, we use the summarisation technique described by Gopan et al. [11]

This technique consists in regrouping several variables under the same symbol. For instance, if we have a state $(x, y, z) \in \{(1, 2, 3); (4, 5, 6)\}$, we can group $x$ and $y$ in a summarised variable $v_{xy}$ and get the possible states as follows:

$$(v_{xy}, z) \in \{(1, 3); (2, 3); (4, 6); (5, 6)\}$$

This is indeed an abstraction, which loses some precision (since we cannot distinguish anymore between $x$ and $y$): the summarised set also represents the concrete state $(x, y, z) = (1, 1, 3)$, which is absent of the original concrete set. The advantages are a more compact representation and, more importantly for us, the ability to represent a potentially infinite number of variables within the usual numerical domains.

We consider the PSO memory model, therefore we have a buffer for each variable for each thread, containing numerical values. For each buffer $x^T$ of the variable $x$ and the thread $T$ where they are defined (that is when $N \geq 2$), we group the variables $x^T_2 ... x^T_N$ into a single summarised variable $x^T_{bot}$, whose possible values will therefore be all the values of these buffer entries. We distinguish all these older entries from $x^T_1$ because it plays a special role as being the source of all reads of $x$ by the thread $T$ (if it is defined). Hence we need to keep it as a non-summarised variable to prevent a major loss of precision on read events.

**Numerical Abstraction of Infinite States.** This abstraction only allows to analyse programs with a finite number of reachable states after summarisation (finite state programs when run under SC). To be able to verify more complex programs, we use abstractions such as numerical domains (e.g. Polyhedra) to represent elements with an unbounded (and potentially infinite) number of states (each one still having an unbounded finite buffer). We will develop this abstraction in the next part, but the key idea is to partition the states with respect to the variables they define, so that the states that define the same variables can be merged into a numerical abstract domain. Using widening, we can then verify the program in Figure 2.

## 2 Concrete Semantics

We consider our program to run under the PSO memory model (therefore our analysis will *a fortiori* be sound for TSO). We specify in Figure 3 the direct encoding domain used to define the corresponding concrete semantics.

**Notations.** *Var* is the set of shared variable symbols, and *VarReg* is the set of registers (or local variables). Unless specified, we use the letters $x, y, z$ for *Var* and $r, s$ for *VarReg*, and $e$ denotes an arithmetic expression over integers and registers only. *Thread* is the set of thread identifiers, typically some $\{1, 2, ..., K\}, K \in \mathbb{N}$.

$$\mathit{VarMem} \triangleq \{x^{mem} \mid x \in \mathit{Var}\} \tag{1}$$

$$\mathit{Mem} \triangleq \mathit{VarMem} \to \mathbb{V} \qquad \mathit{Reg} \triangleq \mathit{VarReg} \to \mathbb{V} \tag{2}$$

$$\forall x \in \mathit{Var}, T \in \mathit{Thread}, N \in \mathbb{N}, \mathit{VarBuf}(x, T, N) \triangleq \left\{x_i^T \mid 1 \leq i \leq N\right\} \tag{3}$$

$$\mathit{Buf}(x, T, N) \triangleq \mathit{VarBuf}(x, T, N) \to \mathbb{V} \tag{4}$$

$$\mathit{BufSizes} \triangleq (\mathit{Var} \times \mathit{Thread}) \to \mathbb{N} \tag{5}$$

$$\mathscr{S} \triangleq \bigcup_{N \in \mathit{BufSizes}} \left( \mathit{Mem} \times \mathit{Reg} \times \prod_{\substack{x \in \mathit{Var} \\ T \in \mathit{Thread}}} \mathit{Buf}(x, T, N(x, T)) \right) \tag{6}$$

$$\mathscr{D} \triangleq \mathcal{P}(\mathscr{S}) \tag{7}$$

**Figure 3.** A concrete domain for PSO programs

$\mathbb{V}$ is the numerical set in which the variables are valued, for instance $\mathbb{Z}$ or $\mathbb{Q}$ (respectively the set of integers and rational numbers). $\mathbb{N}^{>0}$ is the set of strictly positive integers. $\circ$ is the function composition operator.[6] $\prod$ denotes cartesian product: $\prod_{m \leq i \leq n} X_i = X_m \times X_{m+1} \times ... \times X_n$.

Bindings of variables in *Var* exist both in the memory and in the buffers, therefore we duplicate the symbols using *VarMem* and *VarBuf*. In the definition of the states set $\mathscr{S}$, for each $x \in \mathit{Var}, T \in \mathit{Thread}$, $N(x, T)$ is the size of the buffer for the variable $x$ in the thread $T$. For a given state $S$, such a $N$ is unique. We will note it $N_S$ thereafter. A concrete element $X \in \mathscr{D}$ is a set of concrete states: $\mathscr{D} = \mathcal{P}(\mathscr{S})$.

*Remark 1.* $\mathscr{D}$ is isomorphic to a usual numerical points concrete domain. As such, it supports the usual concrete operations for variable assignment ($x := e$), condition evaluation and so on. We will also use the *add* and *drop* operations, which respectively add an unconstrained variable to the environment of the domain, and delete it along with its constraints.

We define in Figure 4 the concrete semantics of the instructions of a thread. Formally, for each instruction `ins` and thread $T$ of the program, we define the concrete operator $[\![ins]\!]_T$ that returns the set of concrete states obtained when the thread $T$ performs this instruction from an input set of states. We build $[\![.]\!]_T$ using basic operations on numerical domains as defined in Remark 1. These operations are noted with $[\![.]\!]$ and operate from sets of states to sets of states. For convenience, we first define $[\![.]\!]_T$ on state singletons, and then lift it pointwise on general state sets.

[6] $(f \circ g)(x)$ is $f(g(x))$. That means operators are listed in the equations in reverse application order.

$$\forall T \in \mathit{Thread}, [\![.]\!]_T : \mathscr{D} \to \mathscr{D} \tag{8}$$

$$\begin{aligned}[\![x := e]\!]_T\{S\} \triangleq [\![x_1^T := e]\!] \circ [\![x_2^T := x_1^T]\!] \circ ...\\ ... \circ [\![x_{N_S(x,T)+1}^T := x_{N_S(x,T)}^T]\!] \circ [\![\mathit{add}\ x_{N_S(x,T)+1}^T]\!]\{S\}\end{aligned} \tag{9}$$

$$[\![r := x]\!]_T\{S\} \triangleq \begin{cases} [\![r := x^{mem}]\!]S & \text{if } N_S(x,T) = 0 \\ [\![r := x_1^T]\!]S & \text{if } N_S(x,T) \geq 1 \end{cases} \tag{10}$$

$$[\![\texttt{mfence}]\!]_T\{S\} \triangleq \begin{cases} S & \text{if } \forall x \in \mathit{Var}, N_S(x,T) = 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{11}$$

$$[\![\texttt{flush}\ x]\!]_T\{S\} \triangleq \begin{cases} \emptyset & \text{if } N_S(x,T) = 0 \\ [\![\mathit{drop}\ x_{N_S(x,T)}^T]\!] \circ [\![x^{mem} := x_{N_S(x,T)}^T]\!]\{S\} & \text{if } N_S(x,T) \geq 1 \end{cases} \tag{12}$$

$$\forall X \in \mathscr{D}, [\![\mathit{ins}]\!]_T X \triangleq \bigcup_{S \in X} [\![\mathit{ins}]\!]_T\{S\} \tag{13}$$

**Figure 4.** Concrete semantics in PSO

The control graph of a program being the product of the control graphs of each thread, the concrete semantics of a program is then the least fixpoint of the equation system described by this graph where edges are labelled by the corresponding operators of Figure 4. The non-determinism of flushes is encoded by a self-loop edge of label $[\![\texttt{flush}\ x]\!]_T$ for each $x \in \mathit{Var}, T \in \mathit{Thread}$ on each vertex in the graph.

*Remark 2.* An equivalent point of view is to consider the asynchronous execution of two parallel virtual processes for each thread: the first one is actually executing the source code instructions of the program and does not perform any flush, and the second one is simply performing an infinite loop of flushes. The non-determinism of this asynchronous parallel execution matches the non-determinism of the flushes.

This equivalent simulation of the execution of the program explains why the concrete operators of each actual instruction do not perform any flush, and especially why $[\![\texttt{mfence}]\!]$ does not actually write anything to the memory: the `mfence` instruction simply prevents the execution of the actual thread to continue until the associated flushing thread has completely emptied the buffer of every variable. Therefore, its semantics are the same of a instruction like `while (some variable can be flushed) {}`, that is a filter on states with no possible flush.

Properties of this concrete semantics are usually undecidable: not only because of the classic reasons (presence of loops, infinite value space, infinitely many possible memory states), but also because buffers have an unbounded possible size. Thus we use the abstract interpretation framework [8] to describe abstract operators whose fixpoint can be computed by iteration on the graph.

$$\mathscr{B}^\flat \triangleq \mathit{Var} \times \mathit{Thread} \rightarrow \{0; 1; 1+\} \tag{14}$$

$$\delta : \mathscr{S} \rightarrow \mathscr{B}^\flat \tag{15}$$

$$\delta(S) \triangleq \lambda(x, T).\begin{cases} 0 & \text{if } N_S(x,T) = 0 \\ 1 & \text{if } N_S(x,T) = 1 \\ 1+ & \text{if } N_S(x,T) > 1 \end{cases} \tag{16}$$

**Figure 5.** A partial information on states buffers to partition them

$$\mathscr{D} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \left(\mathscr{B}^\flat \rightarrow \mathscr{D}\right) \tag{17}$$

$$\alpha_1(X) \triangleq \lambda b^\flat.\left\{S \in X \mid \delta(S) = b^\flat\right\} \tag{18}$$

$$\gamma_1(X_{part}) \triangleq \{S \in \mathscr{S} \mid S \in X_{part}(\delta(S))\} \tag{19}$$

**Figure 6.** The state partitioning abstract domain

## 3 Abstract Semantics

### 3.1 Partitioning

Our first step towards abstraction consists in partitioning the concrete states of some concret element. We do this partition with respect to a partial information, for each thread, on the presence of each variable in its buffer: either it is absent, or it is present once, or it is present more than once. We respectively use the notations 0, 1 and 1+ for these three cases. We define this partitioning criterion $\delta$ in Figure 5.

We then formalise in Figure 6 the resulting domain. We use the usual state partitioning domain, given as a Galois connection $\mathscr{D} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \left(\mathscr{B}^\flat \rightarrow \mathscr{D}\right)$. We emphasise that this abstraction does not lose any information yet.

### 3.2 Summarising

In order to be able to represent and compute potentially unbounded buffer states, we then use summarisation [11]. In each concrete partition where they are defined, we group the variables $x^T_2 ... x^T_N$ into a single summarised variable $x^T_{bot}$.

We denote by $\mathscr{D}^\natural$ the abstract domain resulting from the summarisation of these variables from $\mathscr{D}$, and suppose we have a concretisation function which matches the *representation* notion of Gopan et al. [11]:

$$\gamma_{sum} \ : \ \mathscr{D}^\natural \ \rightarrow \ \mathscr{D}$$

$$[\![\mathit{fold}\ x, y]\!]X = [\![\mathit{drop}\ y]\!](X \cup [\![x := y]\!]X) \tag{20}$$
$$[\![\mathit{expand}\ x, y]\!]X = [\![\mathit{add}\ y]\!]X \cap ([\![\mathrm{swap}\ x, y]\!] \circ [\![\mathit{add}\ y]\!]X) \tag{21}$$

**Figure 7.** Summarisation operations fold and expand

$$\gamma_2 : (\mathscr{B}^\flat \to \mathscr{D}^\natural) \to (\mathscr{B}^\flat \to \mathscr{D}) \tag{22}$$
$$\gamma_2(X^\sharp) \triangleq \lambda b^\flat . \gamma_{sum}(X^\sharp(b^\flat)) \tag{23}$$
$$\gamma : (\mathscr{B}^\flat \to \mathscr{D}^\natural) \to \mathscr{D} \tag{24}$$
$$\gamma \triangleq \gamma_1 \circ \gamma_2 = \lambda X^\sharp . \left\{ S \in \mathscr{S} \mid S \in \gamma_{sum}(X^\sharp(\delta(S))) \right\} \tag{25}$$

**Figure 8.** Summarising the buffers to regain a bounded representation

$\mathscr{D}^\natural$ should also implement the fold and expand operation described by Gopan et al. [11] We recall that $[\![\mathit{fold}\ x, y]\!]X$ is the summarisation operation: $y$ is removed from the environment of $X$ and, for each state, its value is added as a possible value of $x$ (where the remaining part of the state remains the same); and $[\![\mathit{expand}\ x, y]\!]$ is the dual operation that creates a new variable $y$ whose possible values are the same as $x$ (put in another way, it inherits all the constraints of $x$, but the value of $y$ is not necessarily equal to that of $x$).

We give in Figure 7 an equivalent formulation of fold and expand as described by Siegel and Simon [20], which can also be used for implementing them on domains where they are not natively defined. $[\![\mathrm{swap}\ x, y]\!]X$ swaps the value of $x$ and $y$ in each state of $X$.

We formalise in Figure 8 the resulting abstract domain. As Gopan et al. [11] doe not give an abstraction function (not least since it does not necessarily exist after numerical abstraction, see next Section 3.3), we only provide a concretisation function $\gamma_2$ from the summarised domain to the partitioned domain. We also define a global concretisation function $\gamma$ from the summarised domain to the original concrete domain $\mathscr{D}$.

We give the corresponding semantics of our resulting abstract domain in Figure 9. We first define partial abstract operators $\{\!|.|\!\}^\sharp_T$ that give, for one abstract input partition $(b^\flat, X^\natural)$, a partition index $b^\flat_{result}$ and a summarised element $X^\natural_{result}$. Then the full operator $[\![.]\!]^\sharp_T$ is computed partitionwise, joining the summarised elements sent in the same partition index.

**Theorem 1.** *The abstract operators defined in Figure 9 are sound:*

$$\forall X^\sharp \in \mathscr{B}^\flat \to \mathscr{D}^\natural, [\![ins]\!]_T \circ \gamma(X^\sharp) \subseteq \gamma \circ [\![ins]\!]^\sharp_T(X^\sharp)$$

*Proof.* Our abstract domain is built as the composition of two abstractions: the state partitioning and the summarisation. Therefore the soundness ensues from

$$\forall T \in \mathit{Thread}, \{\!|.|\!\} : (\mathscr{B}^\flat \times \mathscr{D}^\natural) \to \mathcal{P}(\mathscr{B}^\flat \times \mathscr{D}^\natural) \tag{26}$$

$$\{\!|r := x|\!\}^\sharp_T(b^\flat, X^\natural) = \begin{cases} \left\{b^\flat, [\![r := x^{mem}]\!]X^\natural\right\} & \text{if } b^\flat(x^T) = 0 \\ \left\{b^\flat, [\![r := x^T_1]\!]X^\natural\right\} & \text{otherwise} \end{cases} \tag{27}$$

$$\{\!|x := e|\!\}^\sharp_T(b^\flat, X^\natural) = \begin{cases} \left\{b^\flat[x^T := 1], [\![x^T_1 := e]\!] \circ [\![\mathit{add}\ x^T_1]\!]X^\natural\right\} & \text{if } b^\flat(x^T) = 0 \\ \left\{b^\flat[x^T := 1+], \right. & \\ \left. [\![x^T_1 := e]\!] \circ [\![x^T_{bot} := x^T_1]\!] \circ [\![\mathit{add}\ x^T_{bot}]\!]X^\natural\right\} & \text{if } b^\flat(x^T) = 1 \\ \left\{b^\flat, [\![x^T_1 := e]\!] \circ [\![\mathit{fold}\ x^T_{bot}, x^{temp}_2]\!] \right. & \\ \quad \left. \circ [\![x^{temp}_2 := x^T_1]\!] \circ [\![\mathit{add}\ x^{temp}_2]\!]X^\natural\right\} & \text{if } b^\flat(x^T) = 1+ \end{cases} \tag{28}$$

$$\{\!|\texttt{mfence}|\!\}^\sharp_T(b^\flat, X^\natural) = \begin{cases} \left\{b^\flat, X^\natural\right\} & \text{if } \forall x \in \mathit{Var}, b^\flat(x_T) = 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{29}$$

$$\{\!|\text{flush } x|\!\}^\sharp_T(b^\flat, X^\natural) = \begin{cases} \emptyset & \text{if } b^\flat(x^T) = 0 \\ \left\{b^\flat[x^T := 0], [\![\mathit{drop}\ x^T_1]\!] \circ [\![x^{mem} := x^T_1]\!]X^\natural\right\} & \text{if } b^\flat(x^T) = 1 \\ \left\{\begin{array}{r} b^\flat, [\![\mathit{drop}\ x^{temp}]\!] \circ [\![x^{mem} := x^{temp}]\!] \\ \circ [\![\mathit{expand}\ x^T_{bot}, x^{temp}]\!]X^\natural; \\ b^\flat[x^T := 1], [\![\mathit{drop}\ x^T_{bot}]\!] \circ [\![x^{mem} := x^T_{bot}]\!]X^\natural \end{array}\right\} & \text{if } b^\flat(x^T) = 1+ \end{cases} \tag{30}$$

$$\forall T \in \mathit{Thread}, [\![.]\!]^\sharp_T : (\mathscr{B}^\flat \to \mathscr{D}^\natural) \to (\mathscr{B}^\flat \to \mathscr{D}^\natural) \tag{31}$$

$$[\![\mathit{ins}]\!]^\sharp_T X^\sharp = \lambda b^\flat. \bigcup^\natural_{\substack{\exists b^\flat_1 \in \mathscr{B}^\flat, \\ (b^\flat, X^\natural) \in \{\!|\mathit{ins}|\!\}^\sharp_T(b^\flat_1, X^\sharp(b^\flat_1))}} X^\natural \tag{32}$$

**Figure 9.** Abstract semantics with summarisation

the soundness results on these two domains, provided we compute the right numerical elements and send them in the right partitions.

Therefore we will not detail the state partitioning arguments, as it is a fairly common abstraction. We will simply check that $\{\!|.|\!\}^\sharp_T$ sends its images into the right partition. However, summarising is less usual, thus we provide more explanations about the construction of the numerical content of the partitions.

The read operation involves no summarised variable, therefore its semantics is direct and corresponds almost exactly to the concrete one. The buffers are left unmodified in the concrete, hence the partition does not change. The fence operation is also an immediate translation of the concrete one.

The write operation distinguishes two cases: if the variable has a count of 0 in the abstract buffer, no summarisation is involved. If it has a count of 1, it involves the variable $x^T_{bot}$, but here summarisation still does not appear, since this variable actually only represents $x_2$ from the concrete state. In both these cases, the abstract operator is almost the same as the concrete one, and there is only one partition in the destination, obtained by adding 1 to the presence of $x_T$.

In the third case, when the variable is present more than once, summarisation does apply. The numerical part is obtained by translating each step of Equation (9) into $\mathscr{D}^\natural$, following the results of Gopan et al. [11] The steps $[\![x_1^T := e]\!]$, $[\![x_2^T := x_1^T]\!]$ and $[\![add\ x^T_{N_S(x,T)+1}]\!]$ are directly translated into the corresponding parts in Equation (28). $[\![addx^T_{N_S(x,T)+1}]\!]$ is translated as the identity, since $x^T_{bot}$ already exists. We now consider the operations of the form $[\![x^T_{i+1} := x^T_i]\!]$, for $i \geq 2$. They all translate into:

$$[\![drop\ x']\!] \circ [\![fold\ x^T_{bot}, x'']\!] \circ [\![x'' := x']\!] \circ [\![add\ x'']\!] \circ [\![expand\ x^T_{bot}, x']\!]$$

After applying $[\![x'' := x']\!]$, $x''$ and $x'$ become interchangeable: they share the exact same constraints. Therefore, we can rewrite this formula as:

$$[\![drop\ x'']\!] \circ [\![fold\ x^T_{bot}, x']\!] \circ [\![x'' := x']\!] \circ [\![add\ x'']\!] \circ [\![expand\ x^T_{bot}, x']\!]$$

Then, $[\![drop\ x'']\!]$ and $[\![fold\ x^T_{bot}, x']\!]$ being independent, we can make them commute:

$$[\![fold\ x^T_{bot}, x']\!] \circ [\![drop\ x'']\!] \circ [\![x'' := x']\!] \circ [\![add\ x'']\!] \circ [\![expand\ x^T_{bot}, x']\!]$$

However, $[\![drop\ x'']\!] \circ [\![x'' := x']\!] \circ [\![add\ x'']\!]$ trivially reduces to the identity. Therefore we obtain $[\![fold\ x^T_{bot}, x']\!] \circ [\![expand\ x^T_{bot}, x']\!]$, which Siegel and Simon also demonstrates to be the identity [20]. All the $[\![x^T_{i+1} := x^T_i]\!]$ for $i \geq 2$ being reduced to the identity in the abstract semantics, we indeed get the formula of Equation (28).

As for the partition, it does not change, since adding one element to a buffer containing more than one element still makes it contain more than one element.

The flush operation is the most complex. When the partition only has the variable once in the buffer, it is direct (with no summarisation) as in the other operations.

However, when a flush is done from a partition which has more than one variable in its buffer, the abstract element represents concrete states where this variable is present twice — the concrete image states having it once, and more than twice — the image states having it more than once.

Hence, in the 1+ case, the abstract flush operator performs two computations, respectively considering both possibilities, and sends these results into two different partitions. The contents of these partitions follow directly from the results of Gopan et al. [11]. □

*Example 1.* Let us consider the result of the execution (from a zeroed memory) by the thread 1 of the instructions $x = 1; x = 2$, with no flush yet. The resulting concrete state is $x^{mem} \mapsto 0$, $x^1_1 \mapsto 2$, $x^1_2 \mapsto 1$. We now consider the abstraction of this resulting concrete state. In the abstract buffer part, $x^1$ will be bound to 1+, and the numerical part will be $x^{mem} \mapsto 0$, $x^1_1 \mapsto 2$, $x^1_{bot} \mapsto 1$. This abstract state does not only describe the actual concrete state, but also other concrete states where the buffer could have other entries with the value 1, such as $x^{mem} \mapsto 0$, $x_1^1 \mapsto 2$, $x_2^1 \mapsto 1$, $x_3^1 \mapsto 1$. Therefore, for soundness, the abstract flush of the last entry of the buffer should not only yield the abstract state $(x^1 \mapsto 1)$, $(x^{mem} \mapsto 1,$ $x_1^1 \mapsto 2)$, but also the abstract state $(x^1 \mapsto 1+)$, $(x^{mem} \mapsto 1, x_1^1 \mapsto 2, x_{bot}^1 \mapsto 1)$.

### 3.3 Numerical Abstraction

Eventually, we abstract the $\mathscr{D}^\natural$ part of our elements using numerical domains. We can do this in a direct way because, after partitioning and summarisation, each partition is a set of summarised states which all have the same dimensions (the same variables are defined in each state of one partition).

We obtain the same kind of formulas for our abstract operators as in Figure 9, as long as we use domains which can provide the `fold` and `expand` operations as described by Gopan et al. [11] For instance, the authors define them for Polyhedra and Octagons, and Figure 7 provides a way to compute them with very common operations of abstract domains, so we can use a wide variety of abstractions at this step. We hence obtain a resulting abstraction which is parameterised by the choice of a numerical abstract domain, which can be fine-tuned to obtain some desirable invariant building capabilities.

The abstract domain built this way allows us to approximate efficiently computations on concrete sets containing an unbounded number of states, each of them presenting buffers of unbounded length. The soundness proof remains the same after using numerical domains to abstract sets of summarised states.

## 4 Towards a Better Precision

We now describe additional abstractions we designed to gain more precision on the analysis. Unlike the basic abstractions described in the previous section, we did not implement these domains for lack of time, but they use the same idea of adapting some array abstraction and show that our framework is generic enough to add some additional information if needed.

### 4.1 Non-Uniform Abstraction of Shape Information

We can first improve the precision by using non-uniform abstractions, e.g. the ones described by Cousot et al. [9] As example, let us consider the program in Figure 10.

During the concrete execution of this program, the value of `x` in the memory can only increase, because of the FIFO property of buffers. However, the previous abstraction does not keep this information: due to summarisation, after widening, the abstract variable $x_{bot}^0$ will have every possible natural integer value. Therefore the instructions `r1 = x` and `r2 = x` in Thread 1 will respectively assign all these possible values to `r1` and `r2`, with no relation between them: the property cannot be verified.

```
                    initial x = 0;

/* Thread 0 */              |   /* Thread 1 */
counter = 0;                |   while(true) {
while(true) {               |       r1 = x;
    counter++;              |       r2 = x;
    x = counter;            |       assert (r1 <= r2);
}                           |   }
```

**Figure 10.** A program with writes in ascending order

To solve this problem, we can use, together with the summarisation, an abstraction able to keep the information "the buffer is sorted". This information will indeed be valid (say in decreasing order) for the buffer of the first thread, and the analyser can know it because of the relation $x_1^0 > x_{bot}^0$ which ensures it stays true after writes into $x$. Therefore the flush operator will be able to keep the relation $x_{bot}^0 > x^{mem}$, the last element of the buffer being the smallest one. Then, between `r1 = x` and `r2 = x`, the analysis will compute that either nothing happens, and `r1` and `r2` have the same value, or some combination of writes (not modifying $x^{mem}$) and flushes (increasing $x^{mem}$) happens, and `r1` is smaller than `r2` (or equal if there is no flush). The property holds.

## 4.2 Handling TSO with Order-Preserving Abstractions

Although all these abstractions were designed to target TSO programs, they are also sound for PSO (and defined using this model, as a consequence). They remain sound for TSO, since PSO is strictly weaker; but may lack precision for some programs to be verified, if these programs rely on the specific guarantee of TSO which is write order preservation even between different variables. Let us consider for instance the program in Figure 11.

This program implements Peterson's lock algorithm for threads synchronisation [18]. In TSO, it is indeed correct as written in Figure 11. In PSO, the following sequence of events could happen:

- Thread 0 writes `true` into `flag_0` and `turn`, these are not flushed yet.
- Thread 1 writes `true` into `flag_1` and `false` into `turn`. `turn` is flushed, but not `flag_1` (which is possible under PSO).
- Thread 0 flushes its buffers. It immediately overwrites the previous `false` value of `turn`, which simulates the absence of this write instruction from the program.
- Thread 0 reads `false` from `flag_1` and `true` from `turn` in the memory. It skips the loop and enters the critical section.
- Thread 1 flushes `flag_1`. It reads `true` from `flag_0` and `turn` in the memory, skips the loop and also enters the critical section.

```
/* Property to check: mutual exclusion at (crit1, crit2) */
      initial not flag_0 && not flag_1 && not turn;

      /* Thread 0 */         |   /* Thread 1 */
      flag_0 = true;         |   flag_1 = true;
      // mfence;             |   // mfence;
      turn = true;           |   turn = false;
      mfence;                |   mfence;
                             |
      f = flag_1;            |   f = flag_0;
      t = turn;              |   t = turn;
      while (f && t) {       |   while (f && not t) {
          f = flag_1;        |       f = flag_0;
          t = turn;          |       t = turn;
      }                      |   }
      crit1:                 |   crit2:
      flag_0 = false;        |   flag_1 = false;
```

**Figure 11.** A program where the order between variables is important

In TSO, this kind of execution cannot happen, since Thread 0 cannot flush `turn` before `flag_1` in the memory. However, since our previous abstractions are sound in PSO, they cannot verify that this program is indeed correct in TSO without uncommenting the two additional `mfence` (which prevent the problematic reordering from happening). To be able to do it, we need to add to the abstraction some information on the write order of different variables. With the summarisation abstraction, the difficulty lies in the non-distinction between the older entries for each variable. If a buffer contains $[y \mapsto 1; y \mapsto 2; x \mapsto 0; y \mapsto 3]$, the summarisation loses the distinction between $y \mapsto 2$ and $y \mapsto 3$, so we cannot express that $x \mapsto 0$ is between them.

We propose to alleviate this problem by extending the summarisation idea to the order between entries: we keep the information that a given abstract variable $x_{\{0,bot\}}$ is more recent than an abstract variable $y_{\{0,bot\}}$ if and only if all the concrete buffer entries represented by this $x$ are more recent that all the concrete entries represented by that $y$. On the buffer $[y \mapsto 1; y \mapsto 2; x \mapsto 0; y \mapsto 3]$, that gives us the only information "$y_0$ is more recent than $x_0$". This can be used to ensure that the flush of the $y \mapsto 1$ entry does not happen before the flush of the $x \mapsto 0$ one, but will allow the flush of $y \mapsto 2$ before $x \mapsto 0$.

Peterson's algorithm could then be proved: `turn` being more recent than `flag_1` in the buffer of Thread 1, it cannot be flushed earlier, hence the problematic behaviour is forbidden.

In these two cases (non-uniform and order-preserving abstractions), the formalisation will be sensibly the same as the one simply using summarisation. The additional precision will be expressed as a finer paritioning: some partitions will have extra information such as "the buffer is sorted", and this information can be used for restricting the possible output values of some abstract operators on these partitions. For instance, the $[\![\texttt{flush}\ x]\!]_T$ operator will return $\bot$ on a partition where $y_{bot}$ is known as being older than $x_{bot}$.

## 5 Experimentations

We implemented our approach and tested it against several concurrent algorithms (for PSO). Our implementation is written in OCaml. It runs with the library ocamlgraph [7], using a contributed module which iteratively computes the fixpoint of the abstract semantics over the product graph of the interleavings of the threads, with the general method described by Bourdoncle [6]. We used the (logico-)numerical relational domains provided by the libraries Apron [13] and Bddapron [12]. Our experiments run on a Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz computer with 4GB RAM. We used the formulas of Siegel and Simon [20] to implement for the Bddapron domains the `expand` and `fold` operations.

Our objective was to be able to verify the correctness of these algorithms after removing the maximal number of fences (with respect to the sequentially consistent behaviour where a fence is inserted after each write). For each algorithm, this correctness was encoded by a safety property, usually some boolean condition expressing a relation between variables that must be true at some given point of the execution.[7]

A secondary goal was to obtain an analysis as fast as possible by adjusting the settings concerning the parametric numerical domain. We tried four relational or weakly relational logico-numerical domains: Octagons, Polyhedra, Octagons with BDD and Polyhedra with BDD.[8] For each of these domains, we give in Figure 12 the number of fences needed to verify the correctness property of the algorithm, the time needed to compute this verification and the memory used (line `Average resident set size` of `$ time -v`).

We also give, for comparison, the results obtained for PSO by Dan et al. (domain AGT[9]) [10]. It should be emphasised that we focus on a different problem: while they provide a fence removal algorithm which uses an existing analyser and a program transformation, we design an analysis which works directly on the source program with fences. We discuss fence removal as a case study, removing them with a systematic method not automated for lack of time and measuring the analysis performances on the program with a minimal fence set. The raw Time

[7] On lock algorithms, the correctness being mutual exclusion, this condition was simply `false`, meaning the program state at the given point (the conjunction of the critical sections of each thread) should be $\bot$.

[8] These domains, provided by Bddapron, are essentially numerical domains for the integer variables linked to the leaves of a Binary Decision Diagram which abstracts the boolean variables.

[9] Abstraction Guided Translation.

and Memory numbers comparison is therefore not accurately relevant, but still a reference order of magnitude to evaluate our performances (our test machine being almost as powerful as theirs). However, the number of fences needed is an objective measurement of the precision reachable by the analysis.

When no result is shown for a particular domain, it means that this domain does not allow the verification of the property. It is in particular the case for programs involving some boolean computations with numerical-only domains, because they do not provide abstractions for such operations (we first tried to implement it by encoding booleans into integers, but it usually led to a big loss of precision, making the extra-complexity to implement it non-naively not worthwile against logico-numerical domains.) Some programs verified by Dan et al. [10] are not shown here, because they require operations that our implementation does not provide yet (such as atomic Compare-And-Swap, or more generally patterns requiring cooperative scheduling[10]).

**Interpretation.** These results show that our method is competitive against the state of the art: in most cases, we are able to verify the property of the algorithm with the same number of fences, and in a space and time efficient way if we compare to the characteristic numbers of the abstraction-guided translation.

In two cases (Loop2 TLM and Queue), we strictly improve the result obtained by Dan et al. [10] by being able to verify the algorithm with no inserted fence, compared to respectively 2 and 1. Queue is the program of Figure 2: on top of an unbounded state space in SC (which their method is able to deal with), it exhibits unbounded buffers. Our method allows performing this verification by being able to precisely represent these buffers, whereas they need a finite computable maximal size, hence they have to insert supplementary fences to prevent an unbounded number of writes to be waiting for a flush.

However, the additional abstraction of summarising the buffer even when its size is actually bounded sometimes comes at a precision cost, and our method is not able with this implementation to verify the Bakery algorithm (yet it is able to verify it when the `while true` infinite loop of `lock/unlock` operations is replaced with some `for` loop of fixed size).

Regarding the numerical domain parameter of our abstraction, the results show that Octagons and Polyhedra almost yield the same precisions and efficiency results. In Queue however, Octagons allow verifying the program with no fence, where Polyhedra fail at it. We did not observe the opposite behaviour, and as Octagons are also usually faster due to complexity results, they seem the best go-to choice for abstracting this kind of programs with our method.

When analysing programs which make some consistent use of logical expressions, the boolean-aware domains of Bddapron offer good properties of precision

[10] We did not implement cooperative scheduling nor atomic Compare-And-Swap, but it would make no difference for our abstraction: the only changes would be in the generation of the control graph.

| **Algorithm** | **Domain** | **Fences** | **Time** | **Mem** | **Domain** | **Fences** | **Time** | **Mem** |
|---|---|---|---|---|---|---|---|---|
| Abp | Oct | - | - | - | Bdd+Oct | 0 | 0.3 | 32 |
| | Poly | - | - | - | Bdd+Poly | 0 | 0.3 | 32 |
| | AGT | 0 | 6 | 167 | | | | |
| Bakery | Oct | - | - | - | Bdd+Oct | - | - | - |
| | Poly | - | - | - | Bdd+Poly | - | - | - |
| | AGT | 4 | 3429 | 10951 | | | | |
| Concloop | Oct | 2 | 0.19 | 38 | Bdd+Oct | 2 | 0.29 | 34 |
| | Poly | 2 | 0.24 | 37 | Bdd+Poly | 2 | 0.29 | 34 |
| | AGT | 2 | 6 | 504 | | | | |
| Dekker | Oct | 4 | 23 | 52 | Bdd+Oct | 4 | 62 | 42 |
| | Poly | 4 | 22 | 50 | Bdd+Poly | 4 | 66 | 43 |
| | AGT | 4 | 121 | 1580 | | | | |
| Kessel | Oct | - | - | - | Bdd+Oct | 4 | 4 | 33 |
| | Poly | - | - | - | Bdd+Poly | 4 | 4 | 34 |
| | AGT | 4 | 6 | 198 | | | | |
| Loop2 TLM | Oct | - | - | - | Bdd+Oct | **0** | 4.3 | 34 |
| | Poly | - | - | - | Bdd+Poly | **0** | 4.2 | 34 |
| | AGT | 2 | 36 | 1650 | | | | |
| Peterson | Oct | 4 | 1.53 | 39 | Bdd+Oct | 4 | 2.77 | 32 |
| | Poly | 4 | 1.53 | 39 | Bdd+Poly | 4 | 2.94 | 33 |
| | AGT | 4 | 20 | 901 | | | | |
| Queue | Oct | **0** | 0.15 | 36.9 | Bdd+Oct | **0** | 0.70 | 34.3 |
| | Poly | 1 | 0.2 | 34.7 | Bdd+Poly | **0** | 0.31 | 33.3 |
| | AGT | 1 | 1 | 108 | | | | |

**Figure 12.** Experimental results. Times in sec, Mem in MB.

of efficiency. The overhead over purely numerical domains is however not negligible on programs which do not need this additional expressivity (up to 100% and more time consumed), so these results tend to suggest that they should not be used by default if not necessary on this kind of program.

Another experimental result that does not appear in this table is the number of partitions actually present in the abstract domain. While the maximum theoretical number is $3^{\text{nb\_threads} \times \text{nb\_var}}$, in practice a lot of them are actually empty. We did not measure it precisely for each test, but for instance, in the Peterson case (3 variables, 2 threads), only one state exhibits 9 non-empty partitions. Most of them have 4 or less, often 2. Our analysis is sparse in that empty partitions are not represented at all; hence, we greatly benefit from the small number of non-empty partitions. Partitioning therefore seems to be a good choice since it yields a significantly better precision while not having too much impact on time and space performances. However we do not know to which level a more precise partitioning, as we describe in Section 4, would still verify this statement.

## 6 Related Work

Ensuring correctness of concurrent programs under weakly memory models has been an increasingly important topic in the last few years.

A lot of work has been focused on finite state programs, usually using model-checking related techniques [5,2,4,15].

Kuperstein et al. [14] propose to use abstract interpretation to model programs with potentially unbounded buffers. However, their method uses abstract domains on a statewise basis: for each state, if the buffer grows above a fixed arbitrary size, it is abstracted; but they do not use abstract domains to represent the possibly still unbounded sets of resulting states. As such, their work is only able to analyse programs which have a finite state space when run under sequential consistency (although they can have an infinite state space in TSO, due to unbounded buffers).

The work of Dan et al. [10], which we compared with in Section 5, can manage programs with infinite sequential consistency state space, but they are limited to bounded buffers.

By contrast to these two methods [14,10], our work uses array abstractions and numerical domains to efficiently represent potentially infinite sets of states with unbounded buffers: our analysis has no limitation on the state space of our program, whether it is on state size or on state number.

A common approach for verifying the correctness of weakly consistent executions is to rely on already existing analyses sound under sequential consistency, by performing source-to-source program transformations to bring back the problem to a SC-analysis [10,3,17]. This especially appears to be a useful technique when coupled with automatic fence generation to get back properties verified under sequential consistency. However, some properties can be difficult to express with a program transformation (for instance "the values in the buffer are increasingly sorted"), and the final SC analyser has no way to retrieve information lost by the transformation. Furthermore, these transformations may not be sufficient to efficiently analyse the program, and one may have to modify the SC tool to take into account the original memory model (for instance, by considering that some variables actually come from one buffer, which would make no sense for the original SC analyser).

We believe that, by applying abstract interpretation directly on the original source program, our method can serve as a good baseline for future work on special dedicated abstractions using advanced information related to the model that leverages these two issues.

## 7 Conclusion

We designed a new method for analysing concurrent programs under store-buffer-based relaxed memory models like TSO or PSO. By adapting array abstractions, we showed how to build precise and robust abstractions, parameterised by a numerical domain, which can deal with unbouded buffer sizes.

We gave a formalisation of a particular abstraction which uses summarisation. We implemented this approach and our experimental results demonstrate that this method gives good precision and performance results compared to the state-of-the-art, and sometimes is able to verify programs with strictly fewer fences thanks to its ability to represent unbounded buffers.

As a future work, we shall focus on scalability. While we obtained good performances on our test cases, our method will suffer from the same problem as the previous ones, that is it does not scale well with the number of threads. By using modular analysis techniques (analysing each thread separately instead of considering the exploding product control graph), we should be able to analyse programs with more than two threads in acceptable times. We believe we can do it while reusing the abstractions defined here. However, it is significantly more complex and raises additional problems that need to be solved.

## References

1. Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.
2. 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.
3. Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013.
4. Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *ACM Sigplan Notices*, volume 45, pages 7–18. ACM, 2010.
5. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Programming Languages and Systems*, pages 533–553. Springer, 2013.
6. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.
7. Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ml functors. *Trends in functional programming*, 8:124–140, 2007.
8. 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.
9. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, pages 105–118. ACM, 2011.
10. Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *Verification, Model Checking, and Abstract Interpretation*, pages 449–466. Springer, 2014.
11. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529. Springer, 2004.

12. Bertrand Jeannet. The bddapron logico-numerical abstract domains library, 2009.
13. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.
14. Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011.
15. Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.
16. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.
17. Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *Static Analysis*, pages 237–252. Springer, 2014.
18. Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
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. Holger Siegel and Axel Simon. Summarized dimensions revisited. *Electronic Notes in Theoretical Computer Science*, 288:75–86, 2012.