Boosting Sequential Consistency Checking using Saturation

Rachid Zennou[1,2], Mohamed Faouzi Atig[3], Ranadeep Biswas[1], Ahmed Bouajjani[1], Constantin Enea[1], and Mohammed Erradi[2]

[1] Université de Paris, France {ranadeep, abou, cenea}@irif.fr
[2] ENSIAS, Mohammed V University, Rabat, Morocco, {rachid.zennou, mohamed.erradi}@gmail.com
[3] Uppsala University, Uppsala, Sweden mohamed_faouzi.atig@it.uu.se

Abstract. We address the problem of checking that an execution of a shared memory concurrent program is sequentially consistent (SC). This problem is NP-hard due to the necessity of finding a total order between the write operations that induces an acyclic happen-before relation. We propose an approach allowing to avoid falling systematically in the worst case, and to check SCness in polynomial-time in most cases in practice. The approach is based on a simple yet powerful saturation-based procedure for computing write constraints that must hold for SCness, allowing on one hand fast detection of SC violations, and on the other hand reducing drastically the search space for a total order witnessing SCness.

1 Introduction

Sequential Consistency (SC, for short) [19] is a fundamental model of shared memory, where write and read operations are atomic, and operations issued by different threads are interleaved arbitrarily while the order between operations issued by a same thread is preserved. SC offers the best programming abstraction, since each write operation is considered to be immediately visible to all threads. While adopting SC as a memory model is desirable by memory users as it simplifies their task, implementing sequential consistency is extremely complex and error prone due to various optimisations and complex caching mechanisms that must be adopted in order to achieve acceptable performances. Therefore, it is important to develop automated methods and tools for checking that the executions of an implementation of the memory are sequentially consistent (for every possible client, or for some given client). A crucial problem for developing SC conformance testing tools, is checking if a given single execution is SC. This problem has been shown to be hard. Intuitively, it amounts in finding a total order on write operations that explains the execution, in the sense that the happen-before relation induced by this order (that includes causality and conflict constraints between writes and reads) is acyclic. It has been shown that the problem is NP-complete [15, 17], which means that in the worst case, it is necessary to enumerate the exponentially many possible store orderings in order to solve the problem. Therefore, it is important to investigate methods that avoid

falling systematically in the worst case, and that are able to solve the problem in polynomial time (in the size of the execution) as often as possible in practice.

In [25], we introduced *gradual consistency checking* (GCC, for short) to address this issue. The approach consists in using weaker consistency models (than SC) that are known to be polynomially checkable, such as causal consistency, in two ways. First, finding violations for these "cheaper" models allows to detect efficiently many of the SC violations. Second, and this is the important point, GCC uses weak consistency models for which checking conformance is based on computing, by a polynomial time fixpoint calculation, a set of order constraints on writes that are *included in every store order witnessing SC conformance*, if any. So, computing these constraints reduces the number of pairs of writes for which an order must be found non-deterministically. In [25], we proposed for that a model called Convergence Causal Memory (CCM, for short) that is stronger than all known variants of causal consistency, constructed by combining the constraints imposed by CCv [8] and CM [3, 20].

Then, a natural question is how far the GCC approach can be pushed (i.e., is CCM the strongest model that can be used in this approach)? This paper tackles this question. Our main contribution is the definition of a new consistency criterion called *weak sequential consistency* (wSC, for short) that can be used for this purpose. wSC is defined using a simple *saturation rule* for introducing *store order* constraints. Compared to the definition of CCM, the one of wSC is much more natural and simpler. Interestingly, we prove that wSC is strictly stronger than CCM. This is due to the fact that wSC saturation computes a larger set of constraints on pairs of writes than CCM. Then, the question is still whether it is possible to do better using a saturation-based approach. This question leads to the following more general one: Given an execution that is SC, let us call the SC-kernel of this execution the intersection of all store order relations allowing to establish that the execution is SC (i.e., for which the induced happen-before relation is acyclic). Then, is the store order imposed by wSC always equal to the SC-kernel when the execution is SC? More generally, is it possible to compute the SC-kernel of any execution using saturation when the execution is SC?[4]

First, we show that the wSC saturation rule does not compute the whole SC-kernel in general. We analyze the reason of this by providing several families of counterexamples. We show that there are order constraints that must be imposed on pairs of writes to avoid happen-before cycles including not only one conflict (as wSC saturation does), but several (actually any number) of conflicts involving an arbitrary number of writes. Moreover, we show that in order to impose an order constraint on pairs of writes, in some cases it is necessary to enumerate the possible order of several other pairs of writes, and the number of these pairs can be arbitrarily high. This shows that the design of a saturation-based schema for computing the SC-kernel would require the addition of an unbounded number

[4] The facts that checking SC conformance is NP-hard and that saturation-based computations are polynomial-time do not imply P = NP: given an arbitrary execution, the saturation-based computation would lead to a set of store order constraints, but whether they can be extended to a total order witnessing SC-ness must be checked.

of saturation rules. This provides and interesting insight on the hard instances of the SC checking problem. (Though, this leaves open the theoretical question of the complexity of computing the SC-kernel of an SC execution).

Nevertheless, an interesting question is how far is wSC saturation from computing the SC-kernel in practice? We show experimentally that, interestingly, in practice[5], wSC allows to compute the full SC-kernel in most of the cases (more than 74% of the considered executions), and in general it computes almost the whole SC-kernel (around 99.9% of it). The experiments also show that CCM computes 100% of the SC-kernel for only 0.7% of the executions of the considered benchmark. This shows that the wSC saturation rule is very powerful and efficient in practice, despite its simplicity (and that it is theoretically not complete). In fact, as discussed above, we could have considered other saturation rules to define stronger and stronger consistency models approximating SC. But our experiments show that the benefit would not be important w.r.t. what is already achieved with wSC.

Furthermore, we compare the performances of GCC using CCM versus GCC using wSC. In each case we apply the corresponding saturation procedure to compute a partial order on writes (or partial store order), and then the completion of this order to a *total* order is done using a SAT solver. The two algorithms obtained this way are called CCM+ENUM and wSC+ENUM. Our experiments show that wSC+ENUM is significantly more efficient and more scalable than CCM+ENUM.

Finally, we compare our methods with the approach implemented in DBCOP [7] based on a polynomial search algorithm for checking SC-ness assuming that the number of threads is fixed [1, 7]. While DBCOP is efficient for a small number of threads, its performances degrade quite fast when this number grows, whereas WSC+ENUM is efficient and scales very well in this case. Then, we consider combining saturation with DBCOP. We use wSC saturation to compute a large set of store order constraints that are given to DBCOP in order to reduce the number of interleavings to be explored for SC conformance checking. We obtain this way an efficient algorithm, called wSC+DBCOP, that has better performances than both DBCOP and wSC+ENUM.

Related work. The problem of checking whether a history is SC has been proved to be NP-hard by Gibbons and Korach [17]. In [7, 1], this problem is shown to be polynomial in the size of the history when the number of threads is fixed. The problem of verifying that a finite-state shared-memory implementations (over a bounded number of threads, variables, and values) has been shown to be undecidable by Alur et al. [5].

Several static techniques have been developed to prove that a shared-memory implementation (or cache coherence protocol) satisfies SC [2, 5, 9–12, 14, 16, 18, 21, 22], however only few have addressed dynamic techniques such as testing and runtime verification (which scale to more realistic implementations).

[5] We consider executions of 4 cache coherence protocols within the Gem5 platform.

The idea of using weaker approximations of a memory consistency model (TSO) in order to detect violations has been used, e.g., in [23]. In that paper the authors use a form of saturation that corresponds to a variant of causal consistency (similar to convergence consistency [8]). However, their method is not complete. This idea of saturation is generalized in the framework of gradual consistency checking introduced in [25] where SC is approximated using several variants of causal consistency (including a new one called CCM).

The McVerSi framework [13] addresses test generation (i.e., finding clients that increase the probability of uncovering bugs in shared memory implementations). Their methodology for checking SC lies within the context of white-box testing, i.e., the user is required to annotate the shared memory implementation with events that define the store order in an execution. In the approach we follow, the implementation is treated as a black-box requiring less user intervention.

2 Preliminaries

We introduce in this section basic notions that will be used throughout the paper. We use similar notations and definitions as in [4, 25].

Binary Relations. For a binary relation $\mathsf{r} \subseteq A \times A$ over a given set A, we use r^+ (resp. r^*) to denote the transitive (resp. reflexive transitive) closure of r. We use r^{-1} to denote the inverse relation of r (i.e., $(a,b) \in \mathsf{r}^{-1}$ iff $(b,a) \in \mathsf{r}$). We say that r is a partial order if it is irreflexive (i.e., $(a,a) \notin \mathsf{r}$ for all $a \in A$). We say that r is total if, for every $a, b \in A$, we have either $(a,b) \in \mathsf{r}$ or $(b,a) \in \mathsf{r}$. For two binary relations r_1 and r_2, we use $\mathsf{r}_1 \circ \mathsf{r}_2$ (resp. $\mathsf{r}_1 \cup \mathsf{r}_2$) to denote the composition (resp. union) of r_1 and r_2, i.e., $(a,b) \in \mathsf{r}_1 \circ \mathsf{r}_2$ iff there is an $c \in A$ such that $(a,c) \in \mathsf{r}_1$ and $(c,b) \in \mathsf{r}_2$ (resp. $(a,b) \in \mathsf{r}_1 \cup \mathsf{r}_2$ iff $(a,b) \in \mathsf{r}_1$ or $(a,b) \in \mathsf{r}_2$).

Programs. We consider multi-threaded programs over a set of shared variables $\mathsf{Var} = \{x, y, \ldots\}$. Let Val be an unspecified set of values and $\mathsf{OId} \subseteq \mathbb{N}$ be a set of operation identifiers. We assume that the set of (visible) operations issued by the threads of the program are read and write operations. Formally, the set Op of operations reading or writing a value v to a variable x is defined as $\mathsf{Op} = \{\mathsf{read}_i(x,v), \mathsf{write}_i(x,v) : i \in \mathsf{OId}, x \in \mathsf{Var}, v \in \mathsf{Val}\}$. We omit operation identifiers when it is clear from the context. We use $\mathcal{R}$, (resp. $\mathcal{W}$) to denote the set of read (resp. write) operations. Given an operation $o \in \mathsf{Op}$, we use $\mathsf{var}(o)$ to denote the variable accessed by o. Let O be a subset of Op. We use $\mathcal{R}(O)$ (resp. $\mathcal{W}(O)$) to denote the set of read (resp. write) operations in O.

Histories. A *history* is an abstraction of a program execution. It consists of a set of write or read operations ordered according to two relation: (1) a *partial program order* po that totally orders operations issued by the same thread, and (2) a *write-read* relation wr that identifies the write operation from which each read operation gets it value. Formally, a *history* $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ is a set of operations O along with a strict partial *program order* po and a *write-read* relation $\mathsf{wr} \subseteq \mathcal{W}(O) \times \mathcal{R}(O)$, such that the inverse of wr is a total function and if $(\mathsf{write}(x,v), \mathsf{read}(x',v')) \in \mathsf{wr}$, then $x = x'$ and $v = v'$. We assume that every *history* includes a write operation writing the initial value for each variable

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(y,1)$
$\mathsf{read}(y,0)$	$\mathsf{read}(x,1)$

(a) CCM, wSC and SC

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(x,2)$
$\mathsf{read}(y,0)$	$\mathsf{read}(y,0)$
$\mathsf{write}(y,1)$	$\mathsf{write}(y,2)$
$\mathsf{read}(x,1)$	$\mathsf{read}(x,2)$

(b) CCM but not wSC nor SC

t_0:	t_1:	t_2:	t_3:	t_4:	t_5:
$\mathsf{read}(z,2)$	$\mathsf{write}(x,1)$	$\mathsf{write}(t,1)$	$\mathsf{read}(z,2)$	$\mathsf{read}(z,1)$	$\mathsf{read}(z,1)$
$\mathsf{write}(y,2)$	$\mathsf{write}(y,1)$	$\mathsf{write}(s,1)$	$\mathsf{write}(x,2)$	$\mathsf{write}(t,2)$	$\mathsf{write}(s,2)$
$\mathsf{read}(x,1)$	$\mathsf{write}(z,1)$	$\mathsf{write}(z,2)$	$\mathsf{read}(y,1)$	$\mathsf{read}(s,1)$	$\mathsf{read}(t,1)$

(c) wSC but not SC

Fig. 1: Comparison of different consistency models.

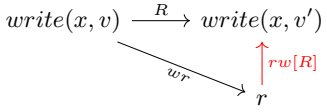
x. These write operations precede all other operations in po. Mentioning these initial write operations is omitted when it is clear from the context.

In the following, we assume also that each value is written at most once. This is not a restriction since shared-memory implementations (or cache coherence protocols) are data-independent [24] in practice, i.e., their behavior doesn't depend on the concrete values read or written in the program, and therefore any potential buggy behavior can be exposed by executions where each value is written at most once. Observe that in this case, the *write-read* relation can be easily extracted by just looking to the value fetched by each read operation.

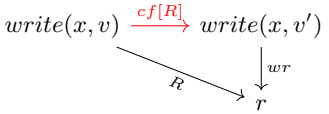
Sequential Consistency. In the following, we recall the formal definition of the Sequential Consistency (SC) memory model [4]. A *history* $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ is *sequentially consistent* if there exists a total relation (called *store order*) $\mathsf{ww} \subseteq \mathcal{W}(O) \times \mathcal{W}(O)$ such that the relation $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ is acyclic, where rw is the *read-write* relation defined by $\mathsf{rw} = \mathsf{wr}^{-1} \circ \mathsf{ww}$. Intuitively, rw expresses the fact that when a read operation $\mathsf{read}(x,v)$ reads a value v from a write operation $\mathsf{write}(x,v)$, and some other write operation $\mathsf{write}(x,v')$ comes after $\mathsf{write}(x,v)$ in the store order, then there is a conflict between $\mathsf{read}(x,v)$ and $\mathsf{write}(x,v')$, and $\mathsf{read}(x,v)$ must happen before $\mathsf{write}(x,v')$.

Figure 1a shows a *history* that is SC. Since $\mathsf{read}(y,0)$ should precede $\mathsf{write}(y,1)$, this *history* admits a total order where the operations of thread t_0 are executed before thread t_1 operations. Figure 1b presents a *history* that does not satisfy SC. The reason is that a total order cannot be found. Since $\mathsf{read}(x,1)$ reads the value from $\mathsf{write}(x,1)$ and $\mathsf{read}(x,2)$ reads the value from $\mathsf{write}(x,2)$, all operations of t_0 should be executed before the operations of t_1, or vice versa. This does not allow either t_0 or t_1 to read the value 0 on variable y.

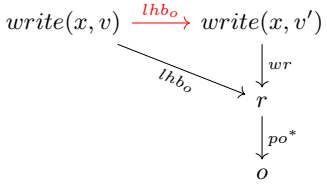
Convergent Causal Memory. The gradual consistency checking approach for checking SC in [25] relies on the use of a weak consistency model called Convergent Causal Memory (CCM) as a polynomially checkable SC approximation. CCM is defined as a strengthening of existing variants of causal consistency. We omit here the definition of these variants and give directly the formal definition of CCM as presented in [25] . For that, some preliminary notions must be intro-



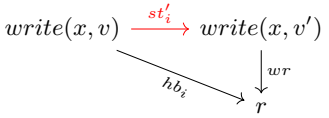
(a) Read-write rw[R]



(b) Conflict order cf[R]



(c) Local happen-before lhb_o



(d) Partial store order st'_i

Fig. 2: Definitions of relations used to define consistency models.

duced. Given a binary relation R on the set of operations, let $R_{\mathbb{WW}}$ (resp. $R_{\mathbb{WR}}$) denotes the projection of R on pairs of writes (resp. pairs of writes and reads) on the same variable. We define also the parametric read-write relation $\mathsf{rw}[R]$ as follows: $\mathsf{rw}[R] = \mathsf{wr}^{-1} \circ R_{\mathbb{WW}}$ (see Fig.2a), i.e.,

$$(\mathsf{read}(x, v), \mathsf{write}(x, v')) \in \mathsf{rw}[R] \text{ iff } (\mathsf{write}(x, v), \mathsf{write}(x, v')) \in R \text{ and } (\mathsf{write}(x, v), \mathsf{read}(x, v)) \in \mathsf{wr}$$

Let co be the *causality relation* defined as the transitive closure of the union of the program order and the write-read relation (i.e., $\mathsf{co} = (\mathsf{po} \cup \mathsf{wr})^+$). Then, we consider a *local happen-before relation* defined with respect to each operation. Given a *history* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$, for every operation o in h, lhb_o[6] is the smallest transitive relation such that:

- if two operations are causally related, and each one is causally related to o, then they are related by the *local happen-before* relation lhb_o, i.e., $(o_1, o_2) \in \mathsf{lhb}_o$ if $(o_1, o_2) \in \mathsf{co}$, $(o_1, o) \in \mathsf{co}$, and $(o_2, o) \in \mathsf{co}^*$, and
- two writes w_1 and w_2 are related by the *local happen-before* relation lhb_o (Fig.2c) if w_1 is lhb_o-related to a read taking its value from w_2, and that read is issued before o by the same thread executing o, i.e.,

$$(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{lhb}_o \text{ if } (\mathsf{write}(x, v), \mathsf{read}(x, v')) \in \mathsf{lhb}_o, (\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}, \text{ and } (\mathsf{read}(x, v'), o) \in \mathsf{po}^*, \text{ for some } \mathsf{read}(x, v').$$

Finally, a *history* $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ is conform to CCM if $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}]$ is acyclic, where the *partial store order* pww is defined by

$$\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \text{ with } \mathsf{lhb} = \Big(\bigcup_{o \in O} \mathsf{lhb}_o\Big)^+$$

[6] This relation was denoted hb_o in [25]. We denote it lhb_o to avoid confusion with other happen-before relations considered in the paper.

where the conflict relation $\mathsf{cf}[R]$ induced by a relation R is defined as $\mathsf{cf}[R] = R_{\mathbb{WR}} \circ \mathsf{wr}^{-1}$ (Fig.2b), i.e.,

$$(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{cf}[R] \text{ iff } (\mathsf{write}(x,v), \mathsf{read}(x,v')) \in R \text{ and } (\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}, \text{ for some } \mathsf{read}(x,v')$$

Notice that the relation rw used in the definition of SC corresponds to $\mathsf{rw}[\mathsf{ww}]$ according to this parametric definition.

3 Weak Sequential Consistency

We propose in this section a new consistency model (called Weak Sequential Consistent) obtained by computing a partial store order using a simple *saturation rule*. This partial store order is inductively defined unlike the SC case where the total store order is existentially quantified. Formally, let st and hb be the smallest relations such that:

$$\begin{aligned} \mathsf{st} &= ((\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{hb}_{\mathbb{WW}})^+ \\ \mathsf{hb} &= (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}])^+ \\ \mathsf{rw}[\mathsf{st}] &= \mathsf{wr}^{-1} \circ \mathsf{st} \end{aligned}$$

Recall that $\mathsf{hb}_{\mathbb{WR}}$ (resp. $\mathsf{hb}_{\mathbb{WW}}$) denote the projection of the relation hb on pairs of writes and reads (resp. pairs of writes on the same variable). Intuitively, the store order st contains the composition of the projection of happen-before relation on pairs of writes and reads and write-read relation, union the projection of happen-before on pairs of writes.

The happen-before relation is similar to the SC one (which corresponds to $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$), it is just that, the store order is deterministically computed using the above saturation rule. Then, a *history* $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ is weakly sequentially consistent (wSC) if hb is acyclic.

Our first contribution consists in showing that wSC is stronger than CCM (which is already stronger than all known variants of causal consistency) [25].

Lemma 1. *If a* history *satisfies* wSC*, then it satisfies* CCM*.*

Proof. Let $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ be a *history* satisfying wSC i.e., $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}]$ is acyclic. We prove that $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}])^+ \subseteq \mathsf{hb}$ (hence the history satisfies also CCM). We will first show that for every operation o in h, $\mathsf{lhb}_o \subseteq \mathsf{hb}$. For that we will prove that hb satisfies the two properties of lhb_o:

- If $(o_1, o_2) \in \mathsf{co}$, $(o_1, o) \in \mathsf{co}$, and $(o_2, o) \in \mathsf{co}^*$ then $(o_1, o_2) \in \mathsf{hb}$ trivially holds (since $\mathsf{co} \subseteq \mathsf{hb}$), and
- if $(\mathsf{write}(x,v), \mathsf{read}(x,v')) \in \mathsf{hb}$ and $(\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}$ then $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in (\mathsf{hb} \circ \mathsf{wr}^{-1})$ and hence $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{st}$ and $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{hb}$.

Thus, we have that $\mathsf{lhb}_o \subseteq \mathsf{hb}$ and hence $\mathsf{lhb} \subseteq \mathsf{hb}$.

Let us now show that $\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \subseteq \mathsf{st}$. It is easy to see that $\mathsf{lhb}_{\mathbb{WW}} \subseteq \mathsf{hb}_{\mathbb{WW}}$ (since $\mathsf{lhb} \subseteq \mathsf{hb}$). By definition, we have also that $\mathsf{cf}[\mathsf{lhb}] = (\mathsf{lhb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1})$ and hence $\mathsf{cf}[\mathsf{lhb}] \subseteq (\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1})$. This implies that $\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \subseteq \mathsf{st} = ((\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{hb}_{\mathbb{WW}})^+$. Finally, it is easy to deduce that $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}])^+ \subseteq \mathsf{hb} = (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}])^+$. □

The reverse of this lemma does not hold. Figure 1b presents a *history* that satisfies CCM but not wSC. A possible partial store order for CCM is to consider that the writes of each thread are not visible to the other thread. The *history* does not satisfy wSC. Since $\mathsf{rw}[st]$ is included in hb, $\mathsf{read}(y, 0)$ is visible to $\mathsf{write}(y, 2)$ then $\mathsf{write}(x, 1)$ precedes $\mathsf{read}(x, 2)$ in hb. Thus, $\mathsf{write}(x, 2)$ should be executed before $\mathsf{write}(x, 1)$. Similarly $\mathsf{write}(x, 2)$ precedes $\mathsf{read}(x, 1)$ in hb as well and $\mathsf{write}(x, 1)$ should be executed before $\mathsf{write}(x, 2)$.

We prove now that wSC is indeed weaker than SC. For that, we need to consider the subrelations of st and hb obtained by iterative least fixpoint calculation. Let $\mathsf{st} = \bigcup_i st_i$ and $\mathsf{hb} = \bigcup_i hb_i$ where $st_i = (hb_{i\mathbb{WW}} \cup st'_i)^+$ and st'_i (Fig.2d) is defined by:

$$(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in st'_i \text{ iff } (\mathsf{write}(x, v), \mathsf{read}(x, v')) \in hb_i \text{ and } (\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$$

where, for every $i \geq 0$, hb_i is defined by:

$$hb_0 = (\mathsf{po} \cup \mathsf{wr})^+$$
$$hb_{i+1} = (hb_i \cup st_i \cup \mathsf{rw}[st_i])^+$$

We now show that the partial store order st_i is included in any store order ww witnessing for SC satisfaction.

Lemma 2. *Let* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *be a* history *and* ww *be a total store order such that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ *is acyclic. Then,* $st_i \subseteq \mathsf{ww}$ *and* $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$.

Proof. The proof is by induction on the index i of hb_i and st_i.

Base Case (i=0). We have $hb_0 = (\mathsf{po} \cup \mathsf{wr})^+$ is included in $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$. Since $hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$, if $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in hb_0$ and there exists a $\mathsf{read}(x, v')$ such that $(\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$, then $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$. Otherwise, assuming by contradiction that $(\mathsf{write}(x, v'), \mathsf{write}(x, v)) \in \mathsf{ww}$, we get $(\mathsf{read}(x, v'), \mathsf{write}(x, v)) \in \mathsf{rw}$. Since $\mathsf{write}(x, v), \mathsf{read}(x, v')) \in hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$, this implies that there is a cycle in $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ which is a contradiction. So, $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$. Thus, st'_0 is included in ww and hence $st_0 = (hb_{0\mathbb{WW}} \cup st'_0)^+$ is also in ww since $hb_{0\mathbb{WW}} \subseteq \mathsf{ww}$ (otherwise it leads to a contradiction since $hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ and $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is acyclic).

Induction Step. Assume that $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ and $st_i \subseteq \mathsf{ww}$. Now, let's show that this holds for $i + 1$ as well. By induction hypothesis, $st_i \subseteq$

ww, so using the definition of $\mathsf{rw}[st_i]$ we have $\mathsf{rw}[st_i] \subseteq \mathsf{rw}$. Then, $hb_{i+1} = (hb_i \cup st_i \cup \mathsf{rw}[st_i])^+ \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$. Now, we show that $st'_{i+1} \subseteq \mathsf{ww}$. If $(\mathsf{write}(x,v), \mathsf{read}(x,v')) \in hb_i$ and $(\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}$, then $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{ww}$. Otherwise, using the same argument as in the base case, we get that $(\mathsf{read}(x,v'), \mathsf{write}(x,v)) \in \mathsf{rw}$ and a contradiction of the fact that $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is acyclic. Hence, if $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in st'_{i+1}$ then $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{ww}$ and so $st'_{i+1} \subseteq \mathsf{ww}$. Furthermore, we have $\mathsf{hb}_{i+1\mathbb{WW}} \subseteq \mathsf{ww}$ since $hb_{i+1} \subseteq (po \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ (otherwise it leads to a contradiction of the fact that $(po \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is acyclic). Since $st_{i+1} = (\mathsf{hb}_{i+1\mathbb{WW}} \cup st'_{i+1})^+$, $st'_{i+1} \subseteq \mathsf{ww}$ and $\mathsf{hb}_{i+1\mathbb{WW}} \subseteq \mathsf{ww}$, we get that $st_{i+1} \subseteq \mathsf{ww}$ (since ww is a total store order). □

As an immediate corollary of Lemma 2, we get:

Lemma 3. *If a* history *satisfies SC, then it satisfies wSC.*

Proof. The proof is by contradiction. Assume that a history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ satisfies SC and it does not satisfy wSC. Since h satisfies SC, then there exists a total store order ww such that $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ is acyclic. Since h does not satisfy wSC, this means that hb is cyclic. Since $\mathsf{hb} = \bigcup_i hb_i$ and $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ (from Lemma 2), we can deduce that $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is also cyclic which constitutes a contradiction. □

The reverse of the above lemma doesn't hold. Figure 1c shows a *history* which satisfies wSC but it is not SC. To show that it satisfies wSC, one can consider a partial store order st where the writes $\mathsf{write}(z,1)$ and $\mathsf{write}(z,2)$ are not ordered. In the other hand, since there is no valid store order for the writes $\mathsf{write}(z,1)$ and $\mathsf{write}(z,2)$, this *history* does not satisfy SC.

Notice that, at each step of the calculation of hb and st, at least one pair of operations is added to one of these two relations and that number of such pairs is polynomially bounded (in the size of the computation). Thus, the acyclicity of hb can be decided in polynomial time.

Theorem 1. *Checking whether a* history *h satisfies wSC can be done in polynomial time in the size of the* history*.*

4 The Sequential Consistency Kernel

Given a history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ that satisfies SC, we define the SC-Kernel of h as the intersection of all store order orders allowing to establish the SCness of h. We know already, from the previous section (Lemma 2), that the store order st, computed by the wSC saturation procedure, is included in any total

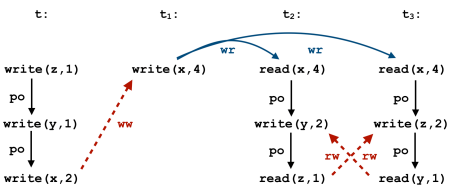


Fig. 3: SC-Kernel counter example

store order ww such that $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ is acyclic. This means that the computed st is always a subset of SC-Kernel. Then, the question is whether the computed store order st is equal to SC-Kernel or not.

In the following, we show that the saturation procedure of wSC may in some cases not be able to compute the SC-Kernel (but rather a strict subset of it). To see why, consider the history given in Fig. 3. The wSC rules do not generate any st relation and therefore the saturation procedure of wSC returns that the store order st is empty while the happens-before relation hb is equal to $(\mathsf{po} \cup \mathsf{wr})^+$. However, any total store order ww that allows to show the SCness of this history should order $\mathsf{write}(x, 4)$ before $\mathsf{write}(x, 2)$ (and hence the pair $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is in the SC-Kernel). We prove that $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ belongs to the SC-Kernel by contradiction. Assume that $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is not in SC-Kernel. Then, there is a total store order ww such that (1) $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in ww (represented in Fig. 3 by a dashed arrow) and (2) $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is acyclic (since the history h is SC). However, if $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in ww then $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is not acyclic (as shown in Fig. 3 by the dashed arrows) and which is a contradiction.

One way to overcome this problem is to include such a pattern in the definition of st'_i used in the saturation procedure. Thus, the definition of st'_i is updated as follows: $(\mathsf{write}(x, v'), \mathsf{write}(x, v)) \in st'_i$ iff one of the following cases holds:

- $(\mathsf{write}(x, v'), \mathsf{read}(x, v)) \in hb_i$ and $(\mathsf{write}(x, v), \mathsf{read}(x, v)) \in \mathsf{wr}$, or
- $(\mathsf{write}(z, v_z), \mathsf{write}(x, v))$, $(\mathsf{write}(y, v_y), \mathsf{write}(x, v))$, $(\mathsf{write}(x, v'), \mathsf{write}(y, v'_y))$, $(\mathsf{write}(y, v'_y), \mathsf{read}(z, v_z))$, $(\mathsf{write}(x, v'), \mathsf{write}(z, v'_z))$, $(\mathsf{write}(z, v'_z), \mathsf{read}(y, v_y))$ are in hb_i and $(\mathsf{write}(z, v_z), \mathsf{read}(z, v_z))$, $(\mathsf{write}(y, v_y), \mathsf{read}(y, v_y))$ are in wr.

Observe that the pattern added to st'_i contains six write operations. Unfortunately, this pattern is not enough to allow us to capture the SC-Kernel. In fact, we can construct a family of counter-examples (see Fig. 4) such that in order to capture all of them, we need to add to the relation st'_i patterns involving a strictly increasing number of writes (which is not feasible in practice).

One way to address the problem raised by the family of counter-examples given in Fig. 4 is to guess for a given pair of writes $\mathsf{write}(x, v)$ and $\mathsf{write}(x, v')$ that are not related by the computed store relation st (i.e., $(\mathsf{write}(x, v), \mathsf{write}(x, v'))$ and $(\mathsf{write}(x, v'), \mathsf{write}(x, v))$ are not in st) one possible order and check if it can make the history h infeasible under SC and if it is the case we add the other order to st. For instance, in the history given in Fig. 3, one would guess that the $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in st. This guess makes the history infeasible under SC due to the existence of a cycle in $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ and hence $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is added to st. Observe that this still results in a saturation procedure which works in polynomial time since we are only allowed to guess the order of at most two unrelated writes.

So the question is whether this extended saturation procedure calculates the SC-Kernel. Alas, this is not true. Consider the history given in Fig. 5. The previous saturation procedure of wSC (augmented with the guessing of the order of one pair of writes) results in an empty store order st. However, this history satisfies SC and $(\mathsf{write}(x, 1), \mathsf{write}(x, 2))$ and $(\mathsf{write}(t, 2), \mathsf{write}(t, 1))$ are in SC-Kernel.

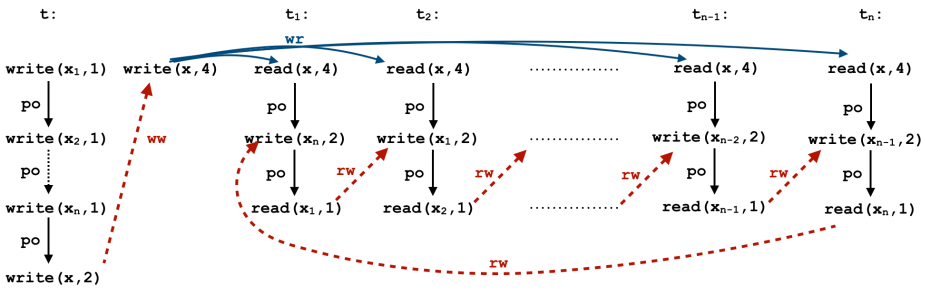


Fig. 4: SC-Kernel counter-examples with cycles involving an arbitrary number of writes

In fact, ordering $\mathsf{write}(x,2)$ before $\mathsf{write}(x,1)$ and $\mathsf{write}(t,2)$ before $\mathsf{write}(t,1)$ creates a happens-before cycle in the top-left block of Figure 5 (in similar manner to the example given in Fig. 3). While ordering $\mathsf{write}(x,2)$ before $\mathsf{write}(x,1)$ and $\mathsf{write}(t,1)$ before $\mathsf{write}(t,2)$ creates a happens-before cycle in the top-right block of Fig. 5. Finally, ordering $\mathsf{write}(x,1)$ before $\mathsf{write}(x,2)$ and $\mathsf{write}(t,1)$ before $\mathsf{write}(t,2)$ creates a happens-before cycle in the top-middle block of Fig. 5. This shows the necessity of augmenting the saturation procedure with the enumeration of the order between two pairs of writes in order to compute the SC-Kernel. Even worst, we can easily extend the history given in Figure 5 in order to force the enumeration of the order between several pairs of writes in order to be able to compute the SC-Kernel. The main idea is to add a number of blocks (in similar manner to the examples given in Figure 3 and Figure 4) to forbid all order combinations between certain pairs of write except one.

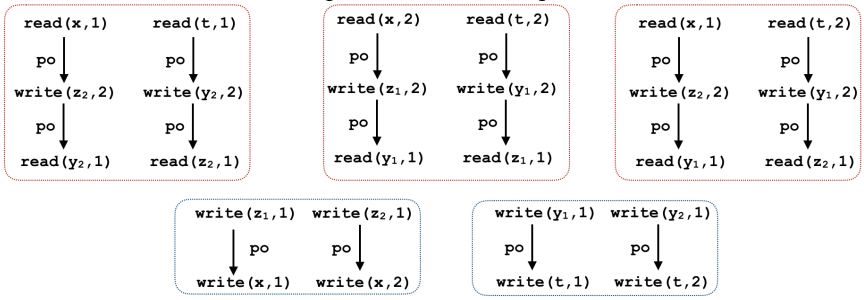


Fig. 5: SC-Kernel counter-example requiring the enumeration of the possible order between two pairs of writes

5 Algorithms for checking SC conformance

We define in this section algorithms for SC checking that exploit the partial store order st computed by the wSC saturation. Following the approach of gradual consistency checking [25], we start by checking that the given history is wSC. If not, then we conclude that it is not SC neither (by Lemma 3). If yes, we exploit st in order to enhance the SC verification of the history. This verification amounts in finding a total store order extending st. To solve this problem we adopt two

approaches, one is based on reducing the SC verification problem to SAT i.e., a direct encoding of the axioms defining SC into a propositional formula, and the second one is based on using the bounded-thread approach of [1, 7] implemented in the tool DBCOP. Both of these approaches are enhanced by the fact that they will use the st constraints in order to reduce their search space. The two obtained algorithms are called wSC+ENUM and wSC+DBCOP, respectively.

The algorithm wSC+ENUM uses an encoding of SC conformance of a given history (defined with its po and wr constraints) as the satisfaction of a Boolean formula. The latter expresses the constraints on the relations involved in the definition of SC, including the fact that the *store oder* ww is a total order relation (so every pair of writes must be order in one direction or the other), and that the happen-before relation (i.e., $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$) is transitive and acyclic. Moreover, the order constraints corresponding to the relation st computed for wSC are added to the formula since $\mathsf{st} \subseteq \mathsf{ww}$.

The algorithm wSC+DBCOP is based on the algorithm implemented in DBCOP [7]. Given a history (again defined by its po and wr relations), DBCOP searches for an interleaving of all the operations of the history that respects the constraints imposed by SC. Then, wSC+DBCOP is an adaptation of DBCOP that exploits st in addition to po and wr as fixed constraints during its search.

For our experiments in next section, we will compare wSC+ENUM and wSC+DBCOP to each other, to DBCOP, and also to CCM+ENUM which is the analogous of wSC+ENUM using CCM saturation instead of wSC saturation. CCM+ENUM is the algorithm proposed in [25].

6 Experimental results

We evaluate in this section the efficiency of our approach and its scalability. We first report on the efficiency of the wSC saturation in computing the SC-kernel. Then, we present an evaluation of the approach in checking SC conformance by taking into account two parameters: the number of operations and the number of threads. The experimental results consider three kinds of benchmarks: The first one consists of only valid histories (i.e., satisfying SC). The second one consists of invalid histories (i.e., violating SC). The third benchmark consists of mixture of valid and invalid histories. These benchmarks are generated by running random clients on realistic cache coherence protocols within the Gem5 simulator [6] in system emulation mode. We use 4 cache coherence protocols that are available in Gem5: MI, MEOSI Hammer, MESI Two Level, and MEOSI AMD Base.

Approximating the SC-kernel. We know already that the store orders computed by the saturation procedures of CCM and wSC are part of the SC-kernel (Lemma 2). The questions are then what is the computed proportion of the SC-kernel, and what is the proportion of the set of pairs of writes in the execution that are not ordered by the saturation procedures. Our experimental results show that wSC computes the SC-kernel in 74.24% of all the 1742 tested histories, and that for the rest of the histories, it computes in average 99.97% of their kernel. For CCM, we found that it computes the SC-kernel only in 0.7% of the same set

of executions. We also found that the wSC saturation procedure orders 98.51% of the pairs of writes of a history in average, and that CCM orders in average 97,89% of the pairs of writes. This is interesting since in terms of coverage of the sets of pairs of write, CCM is not far from wSC, however, only for very few histories it can fully cover its SC-kernel.

SC conformance checking for valid histories. We consider in this section the case of histories that satisfy SC. The experiments are made by varying the number of operations and the number of threads. For each number of operations (threads), we have tested 200 histories and computed the running time average.

Figure 6 reports the running time (in seconds) of the 4 algorithms wSC+ENUM, CCM+ENUM, DPCOP, and wSC+DBCOP while increasing the number of operations from 200 to 800 (by an increase of 100) with a fixed number of 6 threads. It shows that for a relatively small number of threads, DBCOP has the best performances, while wSC+ENUM has good performances and is clearly superior than CCM+ENUM. This can be partly explained by the difference in the coverage of store order constraints between the two algorithms, but most importantly by their time complexity. In fact, the difference in the coverage in average between the two algorithms is small (98.51% vs 97,89%). Thus, the time complexity of the two algorithms plays also an important role: for CCM, the saturation schema requires computing local happen-before relation for each operation, which is very expensive compared with the much simpler saturation schema in wSC.

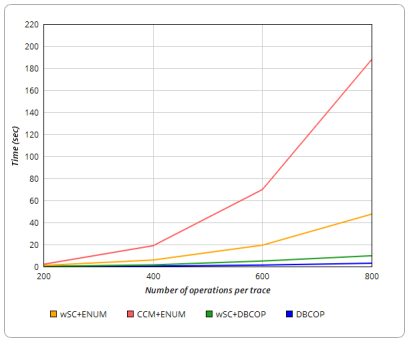
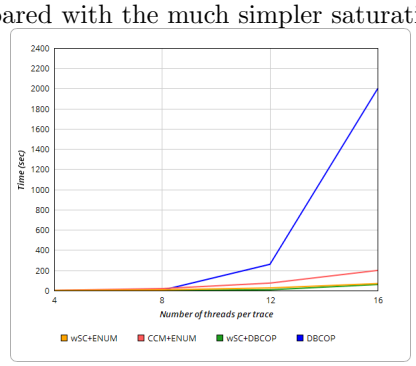
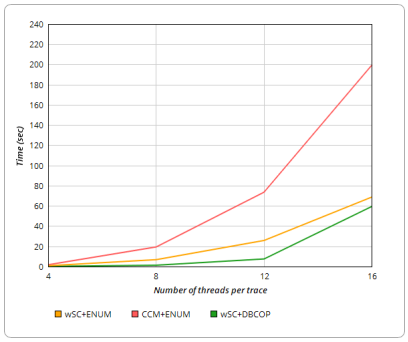


Fig. 6: Checking SCness for valid histories while varying the number of operations.



(a) Comparing all approaches.

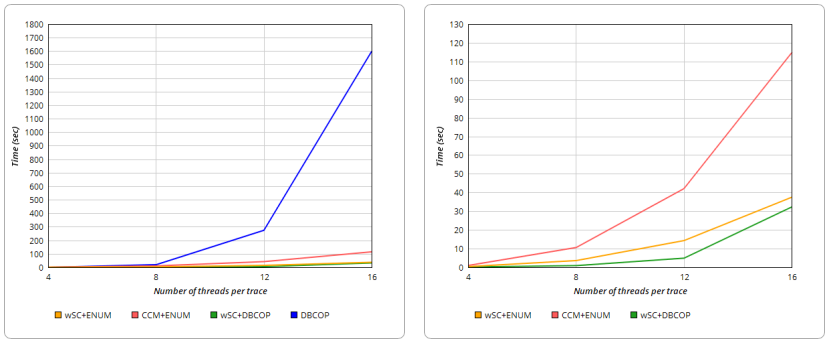


(b) Comparison of wSC+ENUM, CCM+ENUM and wSC+DBCOP.

Fig. 7: Checking SC for valid histories while varying the number of threads.

Figure 7 reports the running time while increasing the number of threads from 4 to 16, by steps of 4. We have considered 50 operations per thread. Notice

that increasing the number of threads increases also the total number of operations. Figure 7(a) shows that the performances of DBCOP degrade beyond 8 threads, while the other algorithms exploiting saturation are more scalable. wSC+DBCOP achieves the best performances while wSC+ENUM performs better than CCM+ENUM. Figure 7(b) is a zoom of Figure 7(a) for a smaller time scale in order to examine more closely the separation between CCM+ENUM, wSC+ENUM, and wSC+DBCOP. It can be seen that the combination of wSC saturation with DBCOP leads to an efficient procedure that takes advantage from the DBCOP strategy for small number of threads, and exploits wSC saturation to stay scalable when both the number of threads and operations increase.



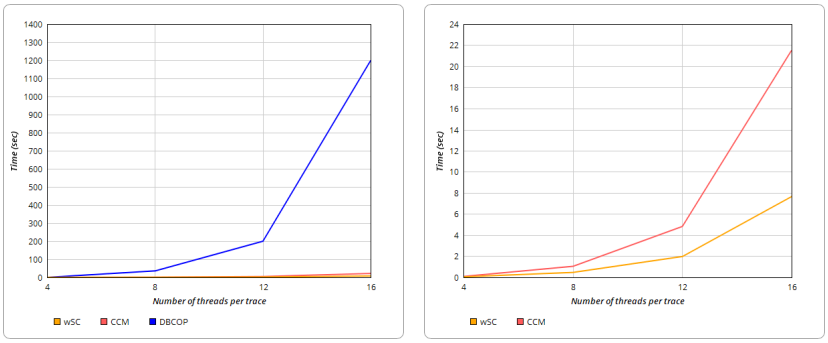
(a) Comparing all approaches.

(b) Comparison of wSC+ENUM, CCM+ENUM and wSC+DBCOP.

Fig. 8: Checking SC for a set of 50% of valid and 50% of invalid histories.

SC conformance checking for (in)valid histories. We now consider a set of histories containing 50% of violations. The violations are generated by randomly changing the write-read relation: for some reads, chosen randomly, we modify the writes from which they get their values. The new writes are chosen randomly within a bounded distance from their corresponding reads. As in the previous paragraph, we consider histories with 4 to 16 threads and we test 200 histories for each number of threads. The experimental results are presented in Figure 8 and they are very similarly to the case with only valid histories.

SC conformance checking for invalid histories. In the following, we consider invalide histories with 4 to 16 threads and 50 operations per thread. For each number of threads, we consider 100 histories and compute the average running time. Since all found violations are already wSC violations, we only compare the saturation steps of wSC, CCM, and DBCOP. Fig. 9b shows that wSC is more efficient than CCM. In addition, wSC captures more SC violations: 1,25% of the violations are not captured by CCM. Fig. 9 shows that wSC has better performance, by factors of 70 times (in the 8 threads case) and higher, compared to DBCOP. In fact, wSC terminates in less than 8 seconds for all the tested histories. This shows the efficiency of wSC in detecting consistency violations. Furthermore, wSC scales very well when increasing the number of threads (and therefore the total number of operations).



(a) Comparison of wSC, CCM and DBCOP.

(b) Comparison of wSC and CCM.

Fig. 9: Checking SC for invalid histories.

7 Conclusion

We have proposed an efficient approach for verifying the conformance of an execution to SC (known to be NP-hard). The approach is based on using a powerful saturation rule for computing in polynomial time a large subset of the SC-kernel of the given execution. Our experimental results show that in practice (1) this allows to catch very quickly almost all SC-violations, and (2) our method allows to compute almost always the whole SC-kernel, and leaves only a very small number of store order constraints to be found in order to check SC-ness. We considered two ways for finding the remaining constraints: either using SAT-solving, or using the search procedure of DBCOP. The latter option, exploiting saturation to enhance DBCOP, is the best one experimentally, leading to a performant and scalable algorithm. An interesting problem for future work is the development of similar approaches for other consistency models for which the conformance verification problem is NP-hard, such as for instance the Total Store Order (TSO) model.

References

1. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA) (2019)
2. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. STTT **18**(5), 495–516 (2016)
3. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Comput. **9**(1), 37–49 (1995)
4. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
5. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1-2), 167–188 (2000)

6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The Gem5 Simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011)
7. Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA) (2019)
8. Burckhardt, S.: Principles of Eventual Consistency. now publishers (2014)
9. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the futurebus+ cache coherence protocol. In: Agnew, D., Claesen, L.J.M., Camposano, R. (eds.) CHDL. IFIP Transactions, vol. A-32, pp. 15–30. North-Holland (1993)
10. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: CAV. LNCS, vol. 1855, pp. 53–68. Springer (2000)
11. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design **23**(3), 257–301 (2003)
12. Eiríksson, Á.T., McMillan, K.L.: Using formal verification/analysis methods on the critical path in system design: A case study. In: Wolper, P. (ed.) CAV. LNCS, vol. 939, pp. 367–380. Springer (1995)
13. Elver, M., Nagarajan, V.: Mcversi: A test generation framework for fast memory consistency verification in simulation. In: HPCA. pp. 618–630. IEEE Computer Society (2016)
14. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS. pp. 352–359. IEEE Computer Society (1999)
15. Furbach, F., Meyer, R., Schneider, K., Senftleben, M.: Memory-model-aware testing: A unified complexity analysis. ACM Trans. Embedded Comput. Syst. **14**(4), 63:1–63:25 (2015)
16. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
17. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997)
18. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9**(1/2), 41–75 (1996)
19. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979)
20. Perrin, M., Mostefaoui, A., Jard, C.: Causal consistency: beyond memory. In: PPoPP. pp. 26:1–26:12. ACM (2016)
21. Pong, F., Dubois, M.: A new approach for the verification of cache coherence protocols. IEEE Trans. Parallel Distrib. Syst. **6**(8), 773–787 (1995)
22. Qadeer, S.: Verifying sequential consistency on shared-memory multiprocessors by model checking. IEEE Trans. Parallel Distrib. Syst. **14**(8), 730–741 (2003)
23. Roy, A., Zeisset, S., Fleckenstein, C.J., Huang, J.C.: Fast and generalized polynomial time memory consistency verification. In: CAV. LNCS, vol. 4144, pp. 503–516. Springer (2006)
24. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: POPL. pp. 184–193. ACM Press (1986)
25. Zennou, R., Bouajjani, A., Enea, C., Erradi, M.: Gradual consistency checking. In: CAV. LNCS, vol. 11562, pp. 267–285. Springer (2019)