A Framework for Consistency Algorithms

Peter Chini
TU Braunschweig, Germany
p.chini@tu-braunschweig.de

Prakash Saivasan
The Institute of Mathematical Sciences, HBNI, Chennai, India
prakashs@imsc.res.in

Abstract

We present a framework that provides deterministic consistency algorithms for given memory models. Such an algorithm checks whether the executions of a shared-memory concurrent program are consistent under the axioms defined by a model. For memory models like SC and TSO, checking consistency is NP-complete. Our framework shows, that despite the hardness, fast deterministic consistency algorithms can be obtained by employing tools from fine-grained complexity.

The framework is based on a universal consistency problem which can be instantiated by different memory models. We construct an algorithm for the problem running in time $\mathcal{O}^*(2^k)$, where k is the number of write accesses in the execution that is checked for consistency. Each instance of the framework then admits an $\mathcal{O}^*(2^k)$-time consistency algorithm. By applying the framework, we obtain corresponding consistency algorithms for SC, TSO, PSO, and RMO. Moreover, we show that the obtained algorithms for SC, TSO, and PSO are optimal in the fine-grained sense: there is no consistency algorithm for these running in time $2^{o(k)}$ unless the exponential time hypothesis fails.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Problems, reductions and completeness

Keywords and phrases Consistency, Weak Memory, Fine-Grained Complexity

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2020.42

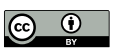
Related Version A full version is available via [20] or `https://arxiv.org/abs/2007.11398`.

1 Introduction

The paper at hand develops a framework for consistency algorithms. Given an execution of a concurrent program over a shared-memory system, consistency algorithms check whether the execution is consistent under the intended behavior of the memory. Our framework takes an abstraction of this intended behavior, a *memory model*, and yields a deterministic consistency algorithm for it. By applying the framework, we obtain provably optimal consistency algorithms for the well-known memory models SC [38], TSO, and PSO [1].

Checking consistency is central in the verification of shared-memory implementations. Such implementations promise programmers consistency guarantees according to a certain memory model. However, due to the complex and performance-oriented design, implementing shared memories is sensitive to errors and implementations may not provide the promised guarantees. Consistency algorithms test this. They take an execution over a shared-memory implementation, multiple sequences of read and write events, one for each thread. Then they check whether the execution is viable under the memory model, namely whether read and write events can be arranged in an interleaving that satisfies the axioms of the model.

In 1997, Gibbons and Korach [32] were the first ones that studied consistency checking as it is considered in this work. They focused on the basic memory model *Sequential Consistency* (SC) by Lamport [38]. In SC, read and write accesses to the memory are atomic making each write of a thread immediately visible to all other threads. Gibbons and Korach showed that


© Peter Chini and Prakash Saivasan;
licensed under Creative Commons License CC-BY

checking consistency in this setting is, in general, NP-complete. Moreover, they considered restrictions of the problem showing that even under the assumption that certain parameters like the number of threads are constant, the problem still remains NP-complete.

The SPARC memory models *Total Store Order* (TSO), *Partial Store Order* (PSO), and *Relaxed Memory Order* (RMO) were investigated by Cantin et al. in [15]. The authors showed that, like for SC, checking consistency for these models is NP-hard. Furbach et al. [31] extended the NP-hardness to almost all models appearing in the Steinke-Nutt hierarchy [46], a hierarchy developed for the classification of memory models. This yields NP-hardness results for memory models like *Causal Consistency* (CC) [37], *Pipelined RAM* (PRAM) [44], *Cache Consistency* [33] or variants of *Processor Consistency* [33, 4]. Bouajjani et al. [11] independently found that checking (variants of) CC for a given execution is NP-hard as well.

We approach consistency checking under the assumption of data-independence [11, 50, 10]. In fact, the behavior of a shared-memory implementation or a database does not depend on precise values in many practical applications [49, 3]. We can therefore assume that in a given execution, a value is written at most once. However, the NP-hardness of checking consistency under SC, TSO, and PSO carries over to the data-independent case [32, 31]. Deterministic consistency algorithms for these models will therefore face exponential running times. By employing a *fine-grained* complexity analysis, we show that one can still obtain consistency algorithms that have only a *mild* exponential dependence on certain parameters. Moreover, we show that the obtained algorithms are provably *optimal*.

Fine-grained complexity analyses are a task of *Parameterized Complexity* [30, 22, 24]. The goal of this new field within complexity theory is to measure the influence of certain parameters on a problem's complexity. In particular, if a problem is NP-hard, one can determine which parameter k of the problem still offers the opportunity for a fast deterministic algorithm. Such an algorithm runs in time $f(k) \cdot poly(n)$, where f is a computable function that only depends on the parameter, and $poly(n)$ is a polynomial in the size of the input n. Problems admitting such algorithms lie in the class FPT of *fixed-parameter tractable* problems. The time-complexity of a problem in FPT is denoted by $\mathcal{O}^*(f(k))$ since $f(k)$ dominates. A fine-grained complexity analysis determines the precise function f that is needed to solve the problem. While finding upper bounds amounts to finding algorithms, lower bounds on f can be obtained from the *exponential time hypothesis* (ETH) [35]. It assumes that n-variable 3-SAT cannot be solved in time $2^{o(n)}$. Among other hardness assumptions, ETH is considered standard in parameterized complexity and was used to derive lower bounds for a variety of problems [22, 39, 21, 16]. A function f is *optimal* when upper and lower bound match.

Our contribution is a framework which yields consistency algorithms that are optimal in the fine-grained sense. Obtained algorithms run in time $\mathcal{O}^*(2^k)$, where k is the number of write events in the given execution. We demonstrate the applicability by obtaining corresponding consistency algorithms for SC, TSO, PSO, and RMO. Relying on the ETH, we prove that for the former three models, consistency cannot be checked in time $2^{o(k)}$. This shows that our framework yields optimal algorithms for these models. Moreover, we are significantly improving upon already existing deterministic algorithms that are usually based on a simple iteration running in time $\mathcal{O}^*(k^k)$. Note that considering other parameters like the number of threads, the number of events per thread, or the size of the underlying data domain yields W[1]-hard problems [42, 32] that are unlikely to admit FPT-algorithms [22, 24].

Our framework is based on a universal consistency problem that can be instantiated by a memory model of choice. We develop an algorithm for this universal problem running in time $\mathcal{O}^*(2^k)$. Then, any instance by a memory model automatically admits an $\mathcal{O}^*(2^k)$-time consistency algorithm. For the formulation of the problem, we rely on the formal framework

of Alglave [5] and Alglave et al. [6] for describing memory models in terms of relations. In fact, checking consistency then amounts to finding a particular *store order* [50] on the write events that satisfies various acyclicity constraints.

For solving the universal consistency problem, we show that instead of a store order we can also find a total order on the write events satisfying similar acyclicity constraints. The latter are algorithmically simpler to find. We develop a notion of *snapshot orders* that mimic total orders on subsets of write events. This allows for shifting from the relation-based domain of the problem to the subset lattice of writes. On this lattice, we can perform a dynamic programming which builds up total orders step by step and avoids an explicit iteration over such which would result in an $\mathcal{O}^*(k^k)$-time algorithm. Keeping track of the acyclicity constraints is achieved by so-called *coherence graphs*. The dynamic programming runs in time $\mathcal{O}^*(2^k)$ which constitutes the time-complexity.

To apply the framework, we follow the formal description of SC, TSO, PSO, and RMO, given in [5, 6] and instantiate the universal consistency problem. Optimality of the algorithms for SC, TSO, and PSO is obtained from the ETH. To this end, we construct a reduction from 3-SAT to the corresponding consistency problem that generates only linearly many write events. The reduction transports the assumed lower bound on 3-SAT to consistency checking.

Related Work. In its general form, consistency checking is NP-hard for most memory models. Furbach et al. [31] show that LOCAL [2] is an exception. Checking consistency under LOCAL takes polynomial time. This also holds for *Cache Consistency* and PRAM if certain parameters of the consistency problem are assumed to be constant. In the case of data-independence, Bouajjani et al. [11] show that checking consistency under CC and variants of CC also takes polynomial time. Wei et al. [48] present a similar result for PRAM. In [50], Bouajjani et al. present practically efficient algorithms for the consistency problems of SC and TSO under data-independence. They rely on the polynomial-time algorithm for CC [11] and obtain a partial store order, which is completed by an enumeration. In theory, the enumeration has a worst-case time complexity of $\mathcal{O}^*(k^k)$. We avoid such an enumeration by a dynamic programming running in time $\mathcal{O}^*(2^k)$. Consistency checking for weaker and stronger notions of consistency, like *linearizability* [34], is considered in [26, 27, 25].

Instead of checking consistency for a single execution of a shared-memory implementation, there were efforts in verifying that all executions are consistent under a certain memory model. Alur et al. show in [7] that for SC, the problem is undecidable. This also holds for CC [11]. Under data-independence, the problem becomes decidable for CC [11]. Verifying *Eventual Consistency* [47] was shown to be decidable by Bouajjani et al. in [12]. There has also been work on other verification problems like reachability and robustness. Atig et al. show in [8] that, under TSO and PSO, reachability is decidable. In [9] the authors extend their results and present a relaxation of TSO with decidable reachability problem. Robustness against TSO was considered in [13] and shown to be PSPACE-complete. This also holds for POWER [40, 45], as shown in [23], and for partitioned global address spaces [14].

Parameterized complexity has been applied to other verification problems as well. Biswas and Enea [10] study the complexity of transactional consistency and obtain an FPT-algorithm in the size and the width of a history. This also yields an algorithm for the serializability problem, proven to be NP-hard by Papadimitriou [43] in 1979. A fine-grained algorithm for serializability under TSO was given in [28]. The authors of [29] present an FPT-algorithm for predicting atomicity violations as well as an intractability result. The parameterized complexity of data race prediction was considered in [42]. Fine-grained complexity analyses were conducted for reachability under *bounded context switching* on finite-state systems [17], and for reachability and liveness on parameterized systems [18, 19].

2 Preliminaries

To state our framework, we introduce some basic notions around memory models and the consistency problem. We mainly follow [6, 5, 50, 11]. Further, we give a short introduction into fine-grained complexity. For standard textbooks in this field, we refer to [30, 24, 22].

Relations, Histories, and Memory Models. We consider the consistency problem: given an execution of a concurrent program and a model of the shared memory, decide whether the execution adheres to the model. Formally, executions consist of *events* modeling write and read accesses to the shared memory. To define these, let Var be the finite set of variables of the program. Moreover, let Val be its finite data domain and Lab a finite set of labels. A *write event* is defined by $w : wr(x, v)$, where $w \in Lab$ is a label, $x \in Var$ is a variable, and $v \in Val$ is a value. The set of write events is defined by $WR = \{w : wr(x, v) \mid w \in Lab, x \in Var, v \in Val\}$. A *read event* is given by $r : rd(x, v)$. The set of read events is denoted by RD. We define the set of all *events* by $E = WR \cup RD$. If it is clear from the context, we omit the label of an event. Given an event $o \in E$, we access the variable of o by $var(o) \in Var$. For a subset $O \subseteq E$, we denote by $WR(O)$ and $RD(O)$ the set of write and read events in O.

For modeling dependencies between events we use strict orders. Let $O \subseteq E$ be a set of events. A *strict partial order* on O is an irreflexive, transitive relation over O[1]. A *strict total order* is a strict partial order that is total. We often refer to the notions without mentioning that they are strict. Given two relations $rel, rel' \subseteq O \times O$, we denote by $rel \circ rel'$ their composition, by rel^+ the transitive closure, and by rel^{-1} the inverse. For variable x, we denote by rel_x the restriction of rel to events on x: $rel_x = \{(o, o') \in rel \mid var(o) = var(o') = x\}$.

Executions are modeled by *histories*. A *history* is a tuple $h = \langle O, po, rf \rangle$, where $O \subseteq E$ is a set of events executed by the threads of the program. The *program order* po is a partial order on O which orders the events of a thread according to the execution. Typically, it is a union of total orders, one for each thread. The relation $rf \subseteq WR(O) \times RD(O)$ is called *reads-from* relation. It specifies the write event providing the value for a read event in the history. Moreover, for each read event $r \in RD(O)$ we have a write event $w \in WR(O)$ such that $(w, r) \in rf$ and if $(w, r) \in rf$, both events access the same variable.

▶ **Example 1.** Consider the history given in Figure 1. It consists of three threads T_1, T_2, and T_{pre} that communicate via the variables x, y, z over the data domain $\{0, 1\}$. The set of events O is given by the events listed in the figure. Each thread processes from top to bottom indicating the program order po. Hence, po is the union of three total orders, one for each thread. For simplicity, we do not draw it. The reads-from relation is determined by the arrows labeled rf. The relation shows that each read event is linked to its corresponding write event. For instance, the two read events $rd(z, 0)$ are linked to the write $wr(z, 0)$. Intuitively this means that in an actual execution, the threads T_1 and T_2 cannot start until T_{pre} finishes and writes $wr(z, 0)$ to the memory since the correct value for z is not available earlier.

Note that in a history, we assume the reads-from relation rf to be given. This is due to the data-independence of shared-memory and database implementations in practice [49, 10, 3, 11, 50]. This means that the behavior of the implementation does not depend on actual values and in an execution, we may assume each value to be written at most once. From such an execution, we can simply read off the relation rf.

[1] Note that the relation has to be irreflexive. This separates it from a usual partial order.

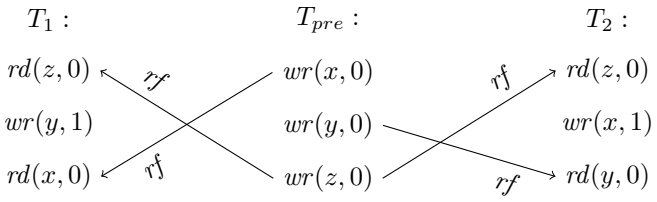


Figure 1 Example of a history. The program order is given implicitly by the arrangement of the events. For each thread, T_1, T_2, and T_{pre}, the program order progresses top to bottom. Formally, it is a union of the resulting three total orders. Arrows labeled by rf show the reads-from relation.

Our framework is compatible with histories that feature *initial writes*. These histories have a write event for each variable writing the initial value of that variable. Formally, these write events are smaller than all other events under program order. If a history $h = \langle O, po, rf \rangle$ is fixed, we abuse notation and also use WR and RD to denote $WR(O)$ and $RD(O)$. For a variable x, we write $WR(x) = \{w \in WR \mid var(w) = x\}$ for the set of write events on x in h. Furthermore, we will later make use of the relation $po\text{-}loc$, defined by restricting po to events on the same variable: $po\text{-}loc = \{(o, o') \in po \mid var(o) = var(o')\}$.

A *memory model* is an abstraction of the memory behavior defining axioms that the relations in a history must adhere to. Formally, a *memory model* MM is a tuple $\mathsf{MM} = (po\text{-}mm, rf\text{-}mm)$. The relation $po\text{-}mm$, also called *preserved program order*, is a subrelation of po describing the structure maintained by the memory model. The latter relation $rf\text{-}mm$ is a subrelation of rf. It shows which write events are visible globally under MM.

Fine-Grained Complexity. For many memory models, the consistency problem is NP-hard [31, 32, 15, 11]. Hence, deterministic consistency algorithms usually face exponential running times. But exponents might only depend on certain parameters of the problem which still allow the algorithm for being fast. Finding such parameters is a task of *parameterized complexity*.

The basis of parameterized complexity are *parameterized problems*. That is, subsets P of $\Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet. An input to P is of the form (x, k), with k being called the *parameter*. A particularly interesting class of parameterized problems are the *fixed-parameter tractable* (FPT) problems. A problem P is FPT if it can be solved by a deterministic algorithm running in time $f(k) \cdot |x|^{\mathcal{O}(1)}$, where f is a computable function only dependent on k. The running time of such an algorithm is usually denoted by $\mathcal{O}^*(f(k))$ to suppress the polynomial part. The class FPT is contained in the class W[1]. Problems that are W[1]-hard are considered intractable since they are unlikely to be FPT.

Given a fixed-parameter tractable problem P, finding an upper bound for f is achieved by constructing an algorithm for P. Lower bounds on f are usually obtained from the *exponential time hypothesis* (ETH) [35]. This standard hardness assumptions asserts that 3-SAT cannot be solved by an algorithm running in time $2^{o(n)}$, where n is the number of variables. A lower bound on f is then obtained by a suitable reduction from 3-SAT to P. We are interested in finding the *optimal* f for the consistency problem where upper and lower bound match. The search for such an f is referred to as *fine-grained complexity*.

3 Framework

We present our framework. Given a model describing the memory, the framework provides an (optimal) deterministic algorithm for the corresponding consistency problem. That is, whether a given history can be scheduled under the axioms imposed by the model. The obtained algorithm can then be used within a testing routine for concurrent programs.

At the heart of the framework is a universal consistency problem that can be instantiated with different memory models. We solve the problem by switching from a relation-based domain, where the problem is defined, to a subset-based domain. On the latter, we can then apply a dynamic programming which constitutes the desired deterministic algorithm.

3.1 Universal Consistency

The basis of our framework is a universal consistency problem which can be instantiated to simulate a particular memory model. For its formulation, we make use of a consistency notion that allows for the construction of a fast algorithm but deviates from the literature [5, 6, 50] at first sight. Therefore, it is proven in Section 4 that instantiating the problem with a particular memory model yields the correct notion of consistency.

We clarify our notion of consistency. Intuitively, a history is consistent under a memory model if it can be scheduled such that certain axioms defined by the model are satisfied. Following the formal framework of [5, 6], finding such a schedule amounts to finding a particular order of the write events that satisfies acyclicity requirements imposed by the axioms. Formally, let $h = \langle O, po, rf\rangle$ be a history and let MM be a memory model described by the tuple $(po\text{-}mm, rf\text{-}mm)$. Then h is called MM-*consistent* if there exists a strict total order tw on the write events WR of h such that the graphs

$$G_{loc} = (O, po\text{-}loc \cup rf \cup tw \cup cf) \quad \text{and} \quad G_{mm} = (O, po\text{-}mm \cup rf\text{-}mm \cup tw \cup cf)$$

are both acyclic. Here, the *conflict relation* cf is defined by $cf = rf^{-1} \circ \bigcup_{x \in Var} tw_x$. Phrased differently, $(r, w) \in cf$ if r is a read event on a variable x, w is a write event on x, and there is a write event w' on x such that $(w', r) \in rf$ and $(w', w) \in tw$.

The acyclicity of G_{loc} is called *uniprocessor* requirement [5] or *memory coherence* for each location [15]. Roughly, it demands that an order among writes to the same location that can be extracted from the history, is kept in tw. The second acyclicity requirement in the definition resembles the underlying memory model MM. If G_{mm} is acyclic, the history can be scheduled adhering to the axioms defined by MM.

▶ **Example 2.** Consider the history given in Example 1. We check consistency under the simple memory model SC. As we will see later in Section 4.2, SC is defined by the tuple $(po\text{-}sc, rf\text{-}sc) = (po, rf)$. For checking consistency, we need to construct the graphs G_{loc} and G_{sc}. To this end, we fix a total order tw on the write events. It is shown as the red edges labeled by tw in Figure 2. Formally, the strict total order tw is the transitive closure of these edges. The next step is to determine the conflict relation cf. It contains two edges. There is an edge $rd(y, 0) \to wr(y, 1)$, shown in blue in Figure 2. This is due to the inverted rf-edge $rd(y, 0) \to wr(y, 0)$ and the tw-edge $wr(y, 0) \to wr(y, 1)$. Note that the latter edge exists in tw (transitive closure) and connects writes to the same variable y which is mandatory for cf. The second cf-edge $rd(x, 0) \to wr(x, 1)$ is obtained similarly but is not shown in the figure.

According to the chosen memory model SC, the graph in Figure 2 shows a subgraph of $G_{sc} = (O, po \cup rf \cup tw \cup cf)$. In fact, only the second conflict edge is missing. But we already obtain a cycle in this graph which traverses as follows:

$$rd(y, 0) \xrightarrow{cf} wr(y, 1) \xrightarrow{tw} wr(x, 1) \xrightarrow{po} rd(y, 0).$$

This constitutes a cycle in G_{sc} and shows that the chosen total order tw does not lead to acyclic graphs and is therefore not a witness for consistency. However, any total order on the write events will cause a cycle implying that the history is not SC-consistent.

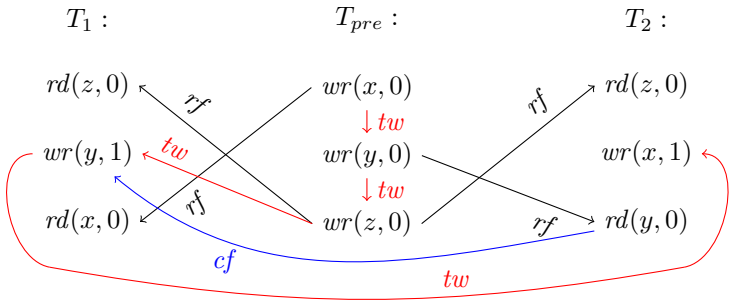


Figure 2 A subgraph of G_{sc}. The total order tw is the transitive closure of the red edges. The blue edge is part of the conflict relation cf. One cf-edge, namely $rd(x,0) \to wr(x,1)$, is missing. The graph contains a cycle showing that the underlying history is not SC-consistent.

Our definition of consistency deviates from the literature in two aspects. First, we demand a total order tw instead of a *store order*, a partial order that is total on writes to the same location [5, 6, 50]. In Section 4 we will show that the resulting notions of consistency are equivalent. A further difference is that we do not explicitly test for *out of thin air values* [41]. For the majority of memory models considered in this work, the test is not necessary as it is implied by the acyclicity of G_{loc} and G_{mm}. But it can easily be added when needed.

We are ready to state the universal consistency problem. To this end, let MM be a fixed memory model. Given a history h, the problem asks whether h is MM-consistent.

MM-*Consistency*
Input: A history $h = \langle O, po, rf \rangle$.
Question: Is h MM-consistent?

Instantiations of the problem by well-known memory models like SC or TSO are typically NP-hard [32, 31]. However, we are interested in a deterministic algorithm for MM-*Consistency*. While we cannot avoid an exponential running time for such an algorithm, a fine-grained complexity analysis can determine the *optimal* exponential dependence. Many parameters of MM-*Consistency* like the number of threads, the maximum size per thread, or the size of the data domain yield parameterizations that are W[1]-hard [42, 32]. Therefore, we conduct a fine-grained analysis for the parameter $k = |WR|$, the number of writes in h. The main finding is an algorithm for MM-*Consistency* running in time $\mathcal{O}^*(2^k)$. The optimality of this approach is shown in Section 5 by a complementing lower bound. We formally state the upper bound in the following theorem. There, $n = |O|$ denotes the number of events in h.

▶ **Theorem 3.** *The problem* MM-*Consistency can be solved in time* $\mathcal{O}(2^k \cdot k^2 \cdot n^2)$.

Note that an algorithm for MM-*Consistency* running in time $\mathcal{O}^*(k^k)$ is immediate. One can iterate over all total orders of WR and check the acyclicity of G_{loc} and G_{mm} in polynomial time. Since we cannot afford this iteration in $\mathcal{O}^*(2^k)$, improving the running time needs an alternative approach and further technical development that we summarize in Section 3.2.

3.2 Algorithm

We present the upper bound for MM-*Consistency* as stated in Theorem 3. Our algorithm is a dynamic programming. It switches from the domain of total orders to subsets of write events and iterates over the latter. The crux is that for a particular subset we do not need to

remember a precise order. In fact, we only need to store that it can be ordered by a so-called *snapshot order* that mimics total orders on subsets. Not having a precise order at hand yields a disadvantage: we cannot just test both acyclicity requirements in the end. Instead, we perform an acyclicity test on a *coherence graph* in each step of the iteration. These graphs carry enough information to ensure acyclicity as it is required by MM-*Consistency*.

We begin our technical development by introducing *snapshot orders*. Intuitively, these simulate total orders of the write events on subsets of writes. Given a subset, a *snapshot order* consists of two parts: a total order on the subset and a partial order. The latter expresses that the complement of the given set precedes the subset but is yet unordered.

▶ **Definition 4.** Let $V \subseteq WR$. A *snapshot order* on V is a union $tw[V] = t[V] \cup r[V]$.

The relation $t[V]$ is a strict total order on V and $r[V] = \{(\overline{v}, v) \mid \overline{v} \in \overline{V}, v \in V\}$ arranges that the elements of $\overline{V}$ are smaller than the elements of V. By $\overline{V}$, we denote the complement of V in the write events, $\overline{V} = WR \setminus V$. Note that $r[V]$ does not impose an order among $\overline{V}$.

A snapshot order is indeed a strict partial order. Even more, when the considered set is the whole write events WR, a snapshot order $tw[WR]$ is a total order on WR. Therefore, MM-consistency can be checked by finding a snapshot order on WR satisfying both acyclicity requirements. The advantage of this formulation is that we can construct such an order from snapshot orders on subsets. Technically, we parameterize[2] the problem along all $V \subseteq WR$.

For the acyclicity requirements, we need a similar parameterization. To this end, let $V \subseteq WR$ be a subset and $tw[V]$ a snapshot order on V. We parameterize the above graphs G_{loc} and G_{mm} via exchanging the total order by the snapshot order:

$$G_{loc}(tw[V]) = (O, po\text{-}loc \cup rf \cup tw[V] \cup cf[V]),$$
$$G_{mm}(tw[V]) = (O, po\text{-}mm \cup rf\text{-}mm \cup tw[V] \cup cf[V]).$$

As above, the conflict relation is defined by $cf[V] = rf^{-1} \circ \bigcup_{x \in Var} tw[V]_x$. Note that for a snapshot order $tw[WR]$ on the whole set of write events, the resulting graphs $G_{loc}(tw[WR])$ and $G_{mm}(tw[WR])$ are exactly those appearing in the acyclicity requirement.

▶ **Example 5.** We reconsider the history of Examples 1 and 2. Our goal is to construct the graph $G_{sc}(tw[V])$ along a snapshot order $tw[V]$. To this end, we first fix a set V. Let $V = \{wr(y,1), wr(x,1)\}$. The set is shown in Figure 3 by the gray highlighted write events. As a snapshot order we chose $tw[V] = t[V] \cup r[V]$, where $t[V]$ consists of only one edge: $wr(y,1) \to wr(x,1)$. Note that this is a total order on V. The edge is shown in Figure 3, it is marked red and labeled by $t[V]$. The relation $r[V]$ is fixed by definition. It contains an edge from each write event in $\overline{V}$ to each write event in V. These are marked green in Figure 3. To construct $G_{sc}(tw[V])$ it is left to determine the relation $cf[V]$. The relation contains two edges, $rd(y,0) \to wr(y,1)$ and $rd(x,0) \to wr(x,1)$. We show the former edge in Figure 3 as well. The latter is omitted to ease readability.

Note that the graph clearly shows that the set V is totally ordered by $t[V]$ while the set $\overline{V}$ is not. The only information that we obtain, from $r[V]$, is that the write events in $\overline{V}$ are *smaller* than the elements in V. In this case, this is already enough to obtain a cycle. This means that each total order on write events that contains $tw[V]$ cannot witness SC-consistency. Note that the total order of Example 2 is such an order.

[2] The parameterization here does not refer to parameterized complexity.

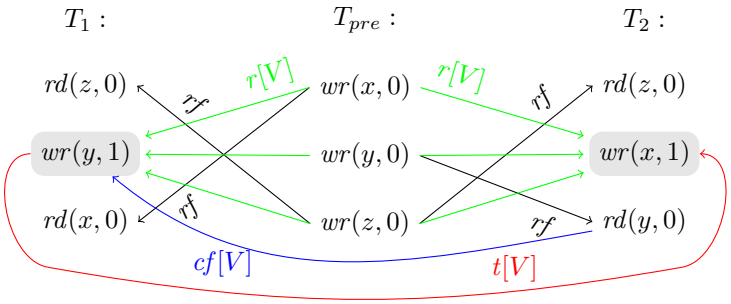


Figure 3 The graph $G_{sc}(tw[V])$ with set $V = \{wr(y,1), wr(x,1)\}$, highlighted gray. The snapshot order $tw[V]$ is given as the union of the total order $t[V]$, marked red, and the partial order $r[V]$, marked green. The relation $cf[V]$ consists of two edges, $rd(y,0) \to wr(y,1)$, shown in blue, and $rd(x,0) \to wr(x,1)$, not shown in the figure.

Now we have the tools to state the parameterization of MM-*Consistency* along subsets of write events. This allows for leaving the domain of total orders and switch to subsets instead. To this end, we define a table T with a Boolean entry $T[V]$ for each $V \subseteq WR$. Entry $T[V]$ will be 1, if there is a snapshot order on V satisfying the acyclicity requirement on both parameterized graphs. Otherwise, $T[V]$ will evaluate to 0. Formally, $T[V]$ is defined by

$$T[V] = \begin{cases} 1, \text{ if } \exists \text{ snapshot ord. } tw[V] : G_{loc}(tw[V]) \text{ and } G_{mm}(tw[V]) \text{ are acyclic,} \\ 0, \text{ otherwise.} \end{cases}$$

The following lemma relates MM-*Consistency* to the table T. It is crucial in our development as it states the correctness of the constructed parameterization. The proof follows from the beforehand definitions and the fact that a snapshot order on WR is already total.

▶ **Lemma 6.** *History h is* MM*-consistent if and only if $T[WR] = 1$.*

We are now left with the problem of evaluating the entry $T[WR]$. Our approach is to set up a recursion among the entries of T and evaluate it via a bottom-up dynamic programming. The recursion will explain how entries of subsets are aggregated to compute entries of larger sets. In fact, write events are added element by element: the recursion shows how an entry $T[V]$ can be utilized to compute the entry of an enlarged set $V \cup \{v\}$, where $v \in \overline{V}$.

When passing from $T[V]$ to $T[V \cup \{v\}]$, we need to provide a snapshot order on $V \cup \{v\}$ that satisfies the acyclicity requirements. A snapshot order on V can always be extended to a snapshot order on $V \cup \{v\}$: we insert v as new minimal element in the contained total order. But we need to keep track of whether the acyclicity is compatible with the new minimal element v. To this end, we perform acyclicity tests on *coherence graphs*. These do not depend on a snapshot order and solely rely on the fact that v is the new minimal element. This will later allow for an evaluation of the table without touching precise orders.

▶ **Definition 7.** Let $V \subseteq WR$ and $v \in \overline{V}$. The *coherence graphs* of V and v are defined by

$$G_{loc}[V,v] = (O, po\text{-}loc \cup rf \cup r[V,v] \cup cf[V,v]),$$
$$G_{mm}[V,v] = (O, po\text{-}mm \cup rf\text{-}mm \cup r[V,v] \cup cf[V,v]).$$

In the definition, relation $r[V,v]$ expresses that $\overline{V \cup \{v\}}$ is smaller than $V \cup \{v\}$ and that v is the minimal element in $V \cup \{v\}$. Formally, it is given by $r[V,v] = r[V \cup \{v\}] \cup \{(v,w) \mid w \in V\}$. The conflict relation is defined by $cf[V,v] = rf^{-1} \circ \bigcup_{x \in Var} r[V,v]_x$.

Coherence graphs are key for the recursion among the entries of T. Assume we are given a snapshot order $tw[V]$ on V meeting the acyclicity requirements of T and we extend it to a snapshot order $tw[V']$ on $V' = V \cup \{v\}$, as above - by inserting v as minimal element of V'. We show that each potential cycle in $G_{loc}(tw[V'])$ or $G_{mm}(tw[V'])$ either implies a cycle in a coherence graph $G_{loc}[V, v]$ or $G_{mm}[V, v]$ or in one of the graphs $G_{loc}(tw[V])$ or $G_{mm}(tw[V])$. If $T[V] = 1$, we can assume the latter graphs to be acyclic. Moreover, if we have checked that the coherence graphs are acyclic as well, we obtain that $T[V'] = 1$. Hence, a recursion should check whether $T[V] = 1$ and whether the corresponding coherence graphs are acyclic.

We formulate the recursion in the subsequent lemma. Note that it is a top-down formulation that only refers to non-empty subsets of write events. An evaluation of the base case is immediate. Entry $T[\emptyset]$ is evaluated to 1 if $G_{loc}(\emptyset) = (O, po\text{-}loc \cup rf)$ and $G_{mm}(\emptyset) = (O, po\text{-}mm \cup rf\text{-}mm)$ are both acyclic. Otherwise it is evaluated to 0.

▶ **Lemma 8.** *Let $V \subseteq WR$ be a non-empty subset. Entry $T[V]$ admits the following recursion:*

$$T[V] = \bigvee_{v \in V} (G_{loc}[V \setminus \{v\}, v] \text{ acyclic}) \wedge (G_{mm}[V \setminus \{v\}, v] \text{ acyclic}) \wedge T[V \setminus \{v\}].$$

We interpret the expression $(G_{loc}[V \setminus \{v\}, v]$ *acyclic*$)$ as a predicate evaluating to 1 if the graph is acyclic and to 0 otherwise. Hence, the recursion requires the existence of a write event $v \in V$ such that both coherence graphs are acyclic and entry $T[V \setminus \{v\}]$ evaluates to 1. A proof of Lemma 8 is provided in the full version of the paper.

With the recursion at hand we can evaluate the table T by a dynamic programming. To this end, we store already computed entries and look them up when needed. An entry $T[V]$ is evaluated as follows. We branch over all write events $v \in V$ and test whether the coherence graphs $G_{loc}[V \setminus \{v\}, v]$ and $G_{mm}[V \setminus \{v\}, v]$ are acyclic. Then, we look up whether $T[V \setminus \{v\}] = 1$. If all three queries are positive, we store $T[V] = 1$. Otherwise, $T[V] = 0$.

The complexity estimation of Theorem 3 is obtained as follows. The table has 2^k many entries that we evaluate, which constitutes the exponential factor. For each entry $T[V]$, we branch over at most k write events $v \in V$. Looking up the value of $T[V \setminus \{v\}]$ can be done in constant time. The following lemma shows that $\mathcal{O}(k \cdot n^2)$ time suffices to construct the coherence graphs and to check them for acyclicity. The latter checks are based on Kahn's algorithm [36] for finding a topological sorting. This completes the proof of Theorem 3.

▶ **Lemma 9.** *Let $V \subseteq WR$ and $v \in \overline{V}$. Constructing the coherence graphs $G_{loc}[V, v]$ and $G_{mm}[V, v]$ and testing both for acyclicity can be done in time $\mathcal{O}(k \cdot n^2)$.*

4 Instantiating the Framework

We show the applicability of our framework and obtain consistency algorithms for the memory models SC, TSO, PSO, and RMO. To this end, we first need to show that our notion of consistency coincides with the notion of consistency used in the literature for these models. This ensures that the obtained algorithms really solve the correct problem. Once this is achieved, we can directly apply the framework to SC, TSO, and PSO. For RMO, we show how the framework can be slightly modified to also capture this more relaxed model.

4.1 Validity

Consistency, as it is considered in the literature, is also known as *validity* [5, 6]. We use the latter name to avoid confusion with our notion of consistency. Before we show that both notions actually coincide, we formally define validity. The definition is based on *store*

orders [5, 6, 50] (also known as *coherence orders*). Given a history $h = \langle O, po, rf\rangle$, a *store order* $ww \subseteq WR \times WR$ takes the form $ww = \bigcup_{x \in Var} ww_x$ so that each ww_x is a strict total order on $WR(x)$. Phrased differently, store orders are unions of total orders on writes to the same variable. Note that, in contrast to a total order on WR, a store order does not have any edge between write events referring to distinct variables.

Validity is similar to consistency. But instead of a total order, the acyclicity requirements need to be satisfied by a store order. Let MM be a memory model described by $(po\text{-}mm, rf\text{-}mm)$. A history $h = \langle O, po, rf\rangle$ is MM-*valid* if there exists a store order so that

$$G^{ww}_{loc} = (O, po\text{-}loc \cup rf \cup ww \cup fr) \quad \text{and} \quad G^{ww}_{mm} = (O, po\text{-}mm \cup rf\text{-}mm \cup ww \cup fr)$$

are acyclic. The *from-read* relation is defined by $fr = rf^{-1} \circ ww$. Note that the definition, as in the case of consistency above, omits checking for out of thin air values. We will later add an explicit test for memory models that require it. This will not affect the complexity.

We show the equivalence of validity and consistency. To this end, we need to prove that a store order can be replaced by a total order on the write events while acyclicity is preserved. The following lemma states the result. It is crucial for the applicability of our framework.

▶ **Lemma 10.** *A history h is* MM*-valid if and only if it is* MM*-consistent.*

Before we give the proof of Lemma 10, we need an auxiliary statement. It shows that a store order ww in G^{ww}_{loc} can be replaced by any linearization of ww without affecting acyclicity. Phrased differently, any total order tw on the write events that contains ww can be inserted into the graph G^{ww}_{loc} - it will still be acyclic. We state the corresponding lemma.

▶ **Lemma 11.** *Let $h = \langle O, po, rf\rangle$ be a history, ww a store order, and tw a total order on WR such that $ww \subseteq tw$. If G^{ww}_{loc} is acyclic, then so is $G^{tw}_{loc} = (O, po\text{-}loc \cup rf \cup tw \cup fr)$.*

The proof of Lemma 11 is given in the full version. We turn to the proof of Lemma 10.

Proof of Lemma 10. First assume that $h = \langle O, po, rf\rangle$ is MM-valid. Then there is a store order ww such that G^{ww}_{loc} and G^{ww}_{mm} are acyclic. Consider the edges of the latter graph. They form a relation $ord\text{-}mm = po\text{-}mm \cup rf\text{-}mm \cup ww \cup fr$. Since G^{ww}_{mm} is acyclic, the transitive closure $ord\text{-}mm^+$ is a strict partial order on O. Hence, there exists a linear extension, a strict total order L containing $ord\text{-}mm^+$. We define $tw = L \cap WR \times WR$. Then, tw is a total order on WR and we have $ww \subseteq L \cap WR \times WR = tw$. We show that G_{loc} and G_{mm} are acyclic. Note that the latter refer to the graphs from the definition of consistency.

The store order ww is contained in tw. Hence, we obtain that $ww_x \subseteq tw_x$ for each variable $x \in Var$. This implies that $ww_x = tw_x$ since ww_x is total on $WR(x)$. We can deduce $ww = \bigcup_{x \in Var} ww_x = \bigcup_{x \in Var} tw_x$ and thus $cf = rf^{-1} \circ \bigcup_{x \in Var} tw_x = rf^{-1} \circ ww = fr$.

Since $fr = cf$, we get the acyclicity of $G_{loc} = G^{tw}_{loc}$ from Lemma 11. The acyclicity of G_{mm} follows since its edges $po\text{-}mm \cup rf\text{-}mm \cup tw \cup cf$ form a subrelation of L. A cycle would mean that L has a reflexive element, but L is a strict order. Hence, h is MM-consistent.

For the other direction, assume that h is MM-consistent. By definition, there is a total order tw on WR such that G_{loc} and G_{mm} are acyclic. We construct the store order $ww = \bigcup_{x \in Var} tw_x$. Note that, since tw_x is total on $WR(x)$, ww is indeed a store order and we have $ww \subseteq tw$. We show that G^{ww}_{loc} and G^{ww}_{mm} are acyclic. In fact, we have that $fr = rf^{-1} \circ ww = cf$. This implies that G^{ww}_{loc} and G^{ww}_{mm} are subgraphs of G_{loc} and G_{mm}, respectively. Hence, the two graphs are acyclic and h is MM-valid. ◀

4.2 Instances

We apply the algorithmic framework to the mentioned memory models and obtain (optimal) deterministic algorithms for their corresponding validity/consistency problem. To this end, we employ the formal description of these models given in [5, 6].

Sequential Consistency. *Sequential Consistency* (SC) is a basic memory model, first defined by Lamport in [38]. Intuitively, SC strictly follows the given program order and flushes each issued write immediately to the memory so that it is visible to all other threads.

Formally, SC is described by the tuple $\mathsf{SC} = (po\text{-}sc, rf\text{-}sc)$ with $po\text{-}sc = po$ and $rf\text{-}sc = rf$. Hence, it employs the full program order and reads-from relation, making the uniprocessor test on G_{loc} obsolete. However, our framework still applies. It yields an algorithm for the corresponding validity/consistency problem running in time $\mathcal{O}(2^k \cdot k^2 \cdot n^2)$. We show in Section 5 that the obtained algorithm is optimal under ETH.

Total Store Ordering. The SPARC memory model *Total Store Order* (TSO) [1] resembles a more relaxed memory behavior. Instead of flushing writes immediately to the memory, like in SC, each thread has an own FIFO buffer and issued writes of that thread are pushed into the buffer. Writes in the buffer are only visible to the owning thread. If the owner reads a certain variable, it first looks through the buffer and reads the latest issued write on that variable. This is called *early read*. At some nondeterministic point, the buffer is flushed to the memory, making the writes visible to other threads as well.

The formal description of TSO is given by the tuple $\mathsf{TSO} = (po\text{-}tso, rf\text{-}tso)$, where $po\text{-}tso = po \setminus WR \times RD$ is a relaxation of the program order, containing no write-read pairs. The relation $rf\text{-}tso = rf_e$ is a restriction of rf to write-read pairs from different threads:

$$rf_e = \{(w, r) \in rf \mid (w, r) \notin po, (r, w) \notin po\}.$$

Unlike in the case of SC, we do not have the full program order and reads-from relation at hand. Hence, the uniprocessor test is essential. Applying the framework yields an algorithm for the validity/consistency problem of TSO running in time $\mathcal{O}(2^k \cdot k^2 \cdot n^2)$. The optimality of the obtained algorithm is shown in Section 5.

Partial Store Ordering. The second SPARC model that we consider is *Partial Store Order* (PSO) [1]. It is weaker than TSO since writes to different locations issued by a thread may not arrive at the memory in program order. Intuitively, in PSO each thread has a buffer per variable where the corresponding writes to the variable are pushed. Like for TSO, threads can read early from their buffers and the buffers are, at some point, flushed to the memory.

Formally, PSO is captured by the tuple $\mathsf{PSO} = (po\text{-}pso, rf\text{-}pso)$. Here, the relation $po\text{-}pso = po \setminus (WR \times RD \cup WR \times WR)$ takes away the write-read pairs and the write-write pairs from the program order and, like for TSO, we have $rf\text{-}pso = rf_e$. Hence, we can apply our framework and obtain an $\mathcal{O}(2^k \cdot k^2 \cdot n^2)$-time algorithm. The obtained algorithm is optimal.

Relaxed Memory Order. We extend the framework to also capture SPARC's *Relaxed Memory Order* (RMO) [1]. The model needs an explicit out of thin air test and allows for so-called *load-load hazards*. We show how both modifications can be built into the framework without affecting the complexity of the resulting consistency algorithm.

The model RMO relies on an additional *dependency relation* resembling address and data dependencies among events in an execution of a program. For instance, if a read event has influence on the value written by a subsequent write event. We assume that the *dependency*

relation dp is given along with a history $h = \langle O, po, rf \rangle$ and is a subrelation of $po \cap (RD \times O)$. The latter means that dp always starts in a read event. With the relation at hand we can perform an out of thin air test. In fact, such a test [5] requires that $(O, dp \cup rf)$ is acyclic. This can be checked by Kahn's algorithm [36] in time $\mathcal{O}(n^2)$. Hence, the test can be added to the framework without increasing the time complexity of the obtained consistency algorithm.

Load-load hazards are allowed by RMO. These occur when two reads of the same variable are scheduled not following the program order. To obtain an algorithm from the framework in this case, we need to weaken the uniprocessor check [5]. In fact, we replace the relation $po\text{-}loc$ by $po\text{-}loc_{llh} = po\text{-}loc \backslash RD \times RD$ and require that the graph $G_{loc-llh} = (O, po\text{-}loc_{llh} \cup rf \cup tw \cup cf)$ is acyclic. The correctness of the framework is ensured since Lemma 10 still holds in this setting. Moreover, the running time of the resulting algorithm is not affected.

With these modifications, we can obtain a consistency algorithm for RMO. Formally, $\mathsf{RMO} = (po\text{-}rmo, rf\text{-}rmo)$ where $po\text{-}rmo = dp$ and $rf\text{-}rmo = rf_e$. Applying the framework with out of thin air test and $G_{loc-llh}$ yields a consistency algorithm running in $\mathcal{O}(2^k \cdot k^2 \cdot n^2)$.

5 Lower Bounds

We show that the framework provides optimal consistency algorithms for SC, TSO, and PSO. To this end, we employ the ETH and prove that checking consistency under these three memory models cannot be achieved in subexponential time $2^{o(k)}$. Since the algorithms obtained in Section 4 match the lower bound, they are indeed optimal.

We begin with the lower bound for SC-Consistency. For its proof, we rely on a characterization of the ETH, known as the *Sparsification Lemma* [35]. It states that ETH is equivalent to the assumption that 3-SAT cannot be solved in time $2^{o(n+m)}$, where n is the number of variables and m is the number of clauses of the input formula. To transport the lower bound to consistency checking, we construct a polynomial-time reduction from 3-SAT to SC-Consistency which controls the number of writes k. Technically, for a given formula φ, the reduction yields a history h_φ that has only $k = \mathcal{O}(n + m)$ many write events and is SC-consistent if and only if φ is satisfiable. By invoking the reduction, an $2^{o(k)}$-time algorithm for SC-Consistency, would yield an $2^{o(n+m)}$-time algorithm for 3-SAT, contradicting the ETH.

▶ **Theorem 12.** SC-*Consistency cannot be solved in time* $2^{o(k)}$ *unless* ETH *fails.*

It is left to construct the reduction. Let φ be a 3-SAT-instance over the variables $X = \{x_1, \ldots, x_n\}$ and with clauses $C_1, \ldots, C_m$. Moreover, let L denote the set of literals. We construct a history h_φ the number of writes of which depends linearly on $n + m$.

The main idea of the reduction is to mimic an evaluation of φ by an interleaving of the events in h_φ. To this end, we divide evaluating φ into three steps: (1) choose an evaluation of the variables, (2) evaluate the literals accordingly, and (3) check whether the clauses are satisfied. For each of these steps we have separate threads taking care of the task. Scheduling them in different orders will yield different evaluations. An overview is given in Figure 4.

Figure 4 presents h_φ as a collection of threads. The program order is obtained from reading threads top to bottom. The reads-from relation is given since each value is written at most once to a variable. Hence, there is always a unique write event providing the read value.

We elaborate on the details of the reduction. For realizing Step (1), we construct two threads, $T_0(x)$ and $T_1(x)$, for each variable $x \in X$. These mimic an evaluation of the variable and consist of only one write event. Thread $T_0(x)$ writes 0 to x, thread $T_1(x)$ writes 1. If $T_0(x)$ gets scheduled before $T_1(x)$, variable x is evaluated to 1 and to 0 otherwise. Hence, the thread that is scheduled later will determine the actual evaluation of x.

$T_0(x):$	$T_1(x):$	$T_0(\ell):$	$T_1(\ell):$	$T^1(C):$	$T^2(C):$	$T^3(C):$
$wr(x,0)$	$wr(x,1)$	$rd(x,0)$	$rd(x,1)$	$rd(\ell_3,0)$	$rd(\ell_1,0)$	$rd(\ell_2,0)$
		$wr(\ell,c)$	$wr(\ell,d)$	$rd(\ell_1,1)$	$rd(\ell_2,1)$	$rd(\ell_3,1)$
		$rd(x,0)$	$rd(x,1)$			

Figure 4 Parts of the history h_φ for a variable $x \in X$, a literal $\ell \in L$, and a clause $C = \ell_1 \vee \ell_2 \vee \ell_3$. Values of c and d depend on ℓ. If $\ell = x$, then $c = 0, d = 1$. Otherwise, $c = 1, d = 0$.

In Step (2), we propagate the evaluation of the variables to the literals. To this end, we construct two threads for each literal $\ell \in L$. Let $\ell = x/\neg x$ be a literal on variable $x \in X$. The first thread $T_0(\ell)$ is responsible for evaluating ℓ when x is evaluated to 0. It first performs a read event $rd(x,0)$, followed by $wr(\ell,c)$ and $rd(x,0)$. The value c depends on the literal: if $\ell = x$, then $c = 0$. Otherwise $c = 1$. Note that the read events guard the write event. This ensures that $T_0(\ell)$ can only run if x is already evaluated to 0 and once $T_0(\ell)$ is running, the evaluation of x cannot change until the thread finishes. Thread $T_1(\ell)$ behaves similar. It evaluates the literal ℓ when x is evaluated to 1. Both threads cannot interfere. Like for the variables, the later scheduled thread determines the actual evaluation of the literal.

It is left to evaluate the clauses. For a clause $C = \ell_1 \vee \ell_2 \vee \ell_3$, we have threads $T^1(C)$, $T^2(C)$, and $T^3(C)$ as shown in Figure 4. It is the task of these threads to ensure that at least one literal in C evaluates to 1. To see this, assume we have the contrary, an evaluation of the variables (and the literals) such that ℓ_1, ℓ_2, and ℓ_3 evaluate to 0. Due to the construction, ℓ_1 storing 0 implies that $wr(\ell_1,1)$ preceded the write event $wr(\ell_1,0)$. Hence, the read event $rd(\ell_1,1)$ in $T^1(C)$ must have already been scheduled. In particular, it has to occur before $rd(\ell_1,0)$ in $T^2(C)$. Since ℓ_2 and ℓ_3 also store 0, we get a similar dependency among their reads: $rd(\ell_2,1)$ occurs before $rd(\ell_2,0)$ and $rd(\ell_3,1)$ occurs before $rd(\ell_3,0)$. Due to program order, we obtain a dependency cycle involving all these reads:

$$rd(\ell_1,1) \to rd(\ell_1,0) \to rd(\ell_2,1) \to rd(\ell_2,0) \to rd(\ell_3,1) \to rd(\ell_3,0) \to rd(\ell_1,1).$$

An arrow $r \to r'$ means that r has to precede r' in an interleaving of the events in h_φ. Since cycles cannot occur in an interleaving, the threads can only be scheduled properly when a satisfying assignment is given. The construction of a proper schedule is subtle. We provide details in the full version of the paper. The following lemma states the correctness.

▶ **Lemma 13.** *Formula φ is satisfiable if and only if the history h_φ is* SC*-consistent.*

Clearly, h_φ can be constructed in polynomial time. We determine the number of write events. For each variable $x \in X$ and each literal $\ell \in L$, we introduce two write events. Hence, $k = 2 \cdot n + 2 \cdot |L|$. Since there are at most $3 \cdot m$ many literals in φ, we get that k is bounded by $2 \cdot n + 6 \cdot m$, a number linear in $n + m$. This finishes the proof of Theorem 12.

We obtain lower bounds for TSO and PSO by constructing a similar reduction from 3-SAT to TSO and PSO-Consistency. To this end, we extend the above reduction by only adding read events that enforce sequential behavior. Intuitively, we can force the FIFO buffers of TSO and PSO to push each issued write to the memory immediately. Then, the above correctness argument still applies. The number of write events does not change and is still linear in $n + m$. This yields the following result. Details are given in the full version.

▶ **Theorem 14.** TSO *and* PSO*-Consistency cannot be solved in time $2^{o(k)}$ unless* ETH *fails.*

6 Conclusion

We studied the problem of checking whether an execution of a shared-memory concurrent program is consistent under the intended behavior of the memory, formalized by a memory model. The main finding is a framework which, given a memory model, yields a deterministic consistency algorithm for it. Obtained algorithms run in time $\mathcal{O}^*(2^k)$, where k is the number of writes in the execution. Technically, the framework works on an abstract memory model and can be instantiated by a concrete one. We applied it to obtain $\mathcal{O}^*(2^k)$-time consistency algorithms for SC, TSO, PSO, and RMO. This improves on the formerly known $\mathcal{O}^*(k^k)$-time algorithms for these models. Furthermore, for SC, TSO, and PSO we have proven that the obtained algorithms are optimal in the fine-grained sense. To this end, we employed the exponential time hypothesis to show that deterministic consistency algorithms for these models cannot run in time $2^{o(k)}$ unless the ETH fails. Our framework relies on the assumption of data-independence. It is an interesting question, and considered future work, whether one can obtain a similar framework yielding optimal algorithms if the assumption is dropped.

References

1 The sparc architecture manual - version 8 and version 9, 1992,1994.
2 H. Sinha A. Heddaya. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston University, 1992.
3 P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2013.
4 M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, page 251–260. ACM, 1993.
5 J. Alglave. A formal hierarchy of weak memory models. *Formal Methods Syst. Des.*, 41(2):178–210, 2012.
6 J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.
7 R. Alur, K. L. McMillan, and D. A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.
8 M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.
9 M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 26–46. Springer, 2012.
10 R. Biswas and C. Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019.
11 A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On verifying causal consistency. In *POPL*, pages 626–638. ACM, 2017.
12 A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *POPL*, pages 285–296. ACM, 2014.
13 A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP*, volume 6756 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2011.
14 G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. A theory of partitioned global address spaces. In *FSTTCS*, volume 24 of *LIPIcs*, pages 127–139. Schloss Dagstuhl, 2013.
15 J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):663–671, 2005.
16 J. Chen, B. Chor, M. Fellows, X. Huang, D. W. Juedes, I. A. Kanj, and G. Xia. Tight lower bounds for certain parameterized np-hard problems. *Inf. Comput.*, 201(2):216–231, 2005.

17 P. Chini, J. Kolberg, A. Krebs, R. Meyer, and P. Saivasan. On the complexity of bounded context switching. In *ESA*, volume 87 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl, 2017.

18 P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. In *TACAS*, volume 10806 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2018.

19 P. Chini, R. Meyer, and P. Saivasan. Complexity of liveness in parameterized systems. In *FSTTCS*, volume 150 of *LIPIcs*, pages 37:1–37:15. Schloss Dagstuhl, 2019.

20 P. Chini and P. Saivasan. A framework for consistency algorithms. *CoRR*, abs/2007.11398, 2020.

21 M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. On problems as hard as CNF-SAT. *ACM Trans. Algorithms*, 12(3):41:1–41:24, 2016.

22 M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. Springer, 2015.

23 E. Derevenetc and R. Meyer. Robustness against power is pspace-complete. In *ICALP*, volume 8573 of *Lecture Notes in Computer Science*, pages 158–170. Springer, 2014.

24 R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

25 M. Emmi and C. Enea. Monitoring weak consistency. In *CAV*, volume 10981 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2018.

26 M. Emmi and C. Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018.

27 M. Emmi, C. Enea, and J. Hamza. Monitoring refinement via symbolic reasoning. In *PLDI*, pages 260–269. ACM, 2015.

28 C. Enea and A. Farzan. On atomicity in presence of non-atomic writes. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pages 497–514. Springer, 2016.

29 A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2009.

30 F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. Springer, 2010.

31 F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63:1–63:25, 2015.

32 P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

33 J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, 1991.

34 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

35 R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *JCSS*, 62(2):367–375, 2001.

36 A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

37 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

38 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

39 D. Lokshtanov, D. Marx, and S. Saurabh. Slightly superexponential parameterized problems. In *SODA*, pages 760–776. SIAM, 2011.

40 S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2012.

41 J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *POPL*, pages 378–391. ACM, 2005.

42 U. Mathur, A. Pavlogiannis, and M. Viswanathan. The complexity of dynamic data race prediction. In *LICS*, pages 713–727. ACM, 2020.

43 C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

44 J. S. Sandberg R. J. Lipton. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

45 S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

46 R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

47 D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183. ACM, 1995.

48 H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu. Verifying PRAM consistency over read/write traces of data replicas. *CoRR*, abs/1302.5161, 2013.

49 P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, pages 184–193. ACM, 1986.

50 R. Zennou, A. Bouajjani, C. Enea, and M. Erradi. Gradual consistency checking. In *CAV*, volume 11562 of *Lecture Notes in Computer Science*, pages 267–285. Springer, 2019.