Coarser Equivalences for Causal Concurrency

AZADEH FARZAN, University of Toronto, Canada

UMANG MATHUR, National University of Singapore, Singapore

Trace theory (formulated by Mazurkiewicz in 1987) is a principled framework for defining equivalence relations for concurrent program runs based on a commutativity relation over the set of atomic steps taken by individual program threads. Its simplicity, elegance, and algorithmic efficiency makes it useful in many different contexts including program verification and testing. It is well-understood that the larger the equivalence classes are, the more benefits they would bring to the algorithms and applications that use them. In this paper, we study relaxations of trace equivalence with the goal of maintaining its algorithmic advantages.

We first prove that the largest appropriate relaxation of trace equivalence, an equivalence relation that preserves the order of steps taken by each thread *and* what write operation each read operation observes, does not yield efficient algorithms. Specifically, we prove a *linear space lower bound* for the problem of checking, in a streaming setting, if two arbitrary steps of a concurrent program run are *causally concurrent* (i.e. they can be reordered in an equivalent run) or *causally ordered* (i.e. they always appear in the same order in all equivalent runs). The same problem can be decided in *constant space* for trace equivalence. Next, we propose a new commutativity-based notion of equivalence called *grain equivalence* that is strictly more relaxed than trace equivalence, and yet yields a constant space algorithm for the same problem. This notion of equivalence uses commutativity of *grains*, which are sequences of atomic steps, in addition to the standard commutativity from trace theory. We study the two distinct cases when the grains are contiguous subwords of the input program run and when they are not, formulate the precise definition of causal concurrency in each case, and show that they can be decided in *constant space*, despite being strict relaxations of the notion of causal concurrency based on trace equivalence.

1 INTRODUCTION

In the last 50 years, several models have been introduced for concurrency and parallelism, of which Petri nets [Hack 1976], Hoare's CSP [Hoare 1978], Milner's CCS [Milner 1980], and event structures [Winskel 1987] are prominent examples. Trace theory [Diekert and Rozenberg 1995] is a paradigm in the same spirit which enriches words (or sequences) by a very restricted yet widely applicable mechanism to model parallelism: some pairs of *events* (atomic steps performed by individual threads) are determined statically to be *independent* (or commutative), and any two sequences that can be transformed to each other through swaps of consecutive independent events are identified as *trace equivalent.* In other words, it constructs a notion of equivalence based on *commutativity* of individual events. The simplicity of trace theory, first formulated by Mazurkiewicz in 1987 [Mazurkiewicz 1987], has made it highly popular in a number of areas in computer science, including programming languages, distributed computing, computer systems, and software engineering. The brilliance of trace theory lies in its simplicity, both conceptually and in yielding simple and efficient algorithms for several core problems in the context of concurrent and distributed programs. It has been widely used in both dynamic program analysis and in construction of program proofs. In dynamic program analysis, it has applications in predictive testing, for instance in data race prediction [Elmas et al. 2007; Flanagan and Freund 2009; Itzkovitz et al. 1999; Kini et al. 2017; Smaragdakis et al. 2012] and in

Authors' addresses: Azadeh Farzan, azadeh@cs.toronto.edu, University of Toronto, Toronto, Canada; Umang Mathur, umathur@comp.nus.edu.sg, National University of Singapore, Singapore, Singapore.

2023. 2475-1421/2023/8-ART $15.00
https://doi.org/

prediction of atomicity violations [Farzan and Madhusudan 2006; Farzan et al. 2009; Sorrentino et al. 2010] among others. In verification, it is used for the simplification of verification of concurrent and distributed programs [Desai et al. 2014; Drăgoi et al. 2016; Farzan 2023; Farzan et al. 2022; Farzan and Vandikas 2019, 2020; Genest et al. 2007].

The philosophy behind most applications of trace theory is that a single representative replaces an entire set of equivalent runs. Therefore, these applications would clearly benefit if larger sets of concurrent runs could soundly be considered equivalent. This motivates the key question in this paper: Can we retain all the benefits of classical trace theory while soundly enlarging the equivalence classes to improve the algorithms that use them? It is not difficult to come up with sound equivalence relations with larger classes. Abdulla et al. [2019] describe the sound equivalence relation *reads-from equivalence* which has the largest classes that remain sound and is defined in a way to preserve two essential properties of concurrent program runs: (1) the total order of *events* in each thread must remain the same in every equivalent run, and (2) each read event must observe the value written from the same write event in every equivalent run. The former is commonly known as the preservation of *program order*, and the latter as the preservation of the *reads-from* relation. These conditions guarantee that any equivalent run is also a valid run of the same concurrent program, and all the observed values (by read events) remain the same, which implies that the outcome of the program run must remain the same. In other words, both control flow and data flow are preserved.

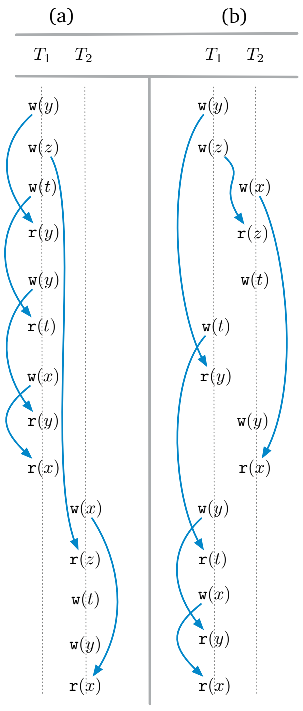


Fig. 1. Read-From Equivalence

Consider the concurrent program run illustrated in Fig. 1(a), and let us focus on the order of the read event $\mathtt{r}(x)$ from thread T_1 and the write event $\mathtt{w}(x)$ from thread T_2. In classical trace theory, these events are dependent (or non-commutatitive) and they must be appear in the same order in every member of the equivalence class of the run. This implies that the illustrated run belongs in an equivalence class of size 1. On the other hand, there exist other reads-from equivalent runs; one such run is illustrated in Fig. 1(b), in which the two aforementioned events have been reordered. The arrows connect each write event to the read event that reads its value, which remain unchanged between the two runs.

We give a formal argument for why (the more relaxed) reads-from equivalence is not as useful as trace equivalence from an algorithmic standpoint. One of the most fundamental algorithmic questions in this context is: Given two events e and e' in a run ρ, do they always appear in the same order in every member of the equivalence class of ρ or can they be reordered in an equivalent run? In the former case, we call e and e' *causally ordered*, and otherwise *causally concurrent*. For example, the events $\mathtt{r}(x)$ from thread T_1 and $\mathtt{w}(x)$ from thread T_2 are causally ordered under trace equivalence but causally concurrent under reads-from equivalence. On the other hand, $\mathtt{w}(z)$ event of thread T_1 and the $\mathtt{r}(z)$ event of thread T_2 are causally ordered under both equivalence relations.

For trace equivalence, it is possible to decide if two events are causally ordered or concurrent, using a *single pass constant space* algorithm; if the length of the run is assumed to be the size of the input, and other program measures such as the number of threads and the number of shared variables are considered to be constants. In Section 3, we prove that if *equivalence* is defined as broadly as reads-from equivalence, then this check can no longer be done in *constant space* by proving a *linear space* lower bound (Theorem 3.1). In particular, we prove this for two closely related variants of this problem: (1) the decision about the ordering of two specific events (as discussed in the example of Fig. 1), and (2) the decision about the ordering of any two occurrences of a specific (atomic) action (e.g. are any two $\mathtt{w}(x)$ actions unordered?). Both problems are closely related to predictive testing of violations of generic correctness properties for concurrent programs, such as data race freedom [Elmas et al. 2007; Flanagan and Freund 2009; Huang et al. 2014; Itzkovitz et al. 1999; Kini et al. 2017; Pavlogiannis 2019; Smaragdakis et al. 2012], deadlock freedom [Kalhauge and Palsberg 2018; Tunç et al. 2023] and atomicity [Farzan and Madhusudan 2006; Farzan et al. 2009; Mathur and Viswanathan 2020; Sorrentino et al. 2010], and have applications in dynamic partial order reduction techniques [Abdulla et al. 2019; Flanagan and Godefroid 2005; Kokologiannakis et al. 2022] for model checking of concurrent programs. In all such contexts, having a monitor whose state space does not depend on the length of the input program run, which may include billions of events, is highly desirable. Thus far, the only existing instance of such a monitor has been based on trace equivalence.

We propose a new notion of equivalence for concurrent program runs, which in terms of expressivity lies in between trace and reads-from equivalences, and retains the highly desirable algorithmic simplicity of traces. The idea is based on enriching the classical commutativity relation of trace theory to additionally account for commutativity of certain *sequences of events*, called *grains*. A grain can be an arbitrarily long sequence of operations, which can belong to multiple threads. What motivates this definition is that in places where swapping a pair of individual events may not be possible, groups of operations (as grains) may still commute *soundly*, meaning without disturbing the program order or the reads-from relation. For two grains to be swappable, they must be adjacent and contiguous, at the time they are swapped.

As an example, consider the concurrent program run in Fig. 2(a). Four grains are marked. Observe that the grains of the same colour soundly commute. Grains of different colour always share a thread and therefore commuting them would break program order. First, assume that only the two blue grains exist. One can then use a sequence of swaps that are either standard swaps permitted by trace equivalence, or a swap of the two blue grains, and transform the run in (a) to the one illustrated in Fig. 2(b), and hence reorder the two $\mathtt{w}(z)$ events. We call the run in (b) to be *grain-equivalent* to the one in (a). Observe that this is not possible under trace equivalence. A similar observation can be made if we consider only the green grains, and the goal of reordering the two $\mathtt{w}(x)$ operations. However, if all four grains are considered together, since the grains of different colour do not commute, nothing can be swapped in Fig. 2(a). Once something is decided to be part of a grain, it must always move together with the rest of the grain. This turns out to be a key to algorithmic simplicity.

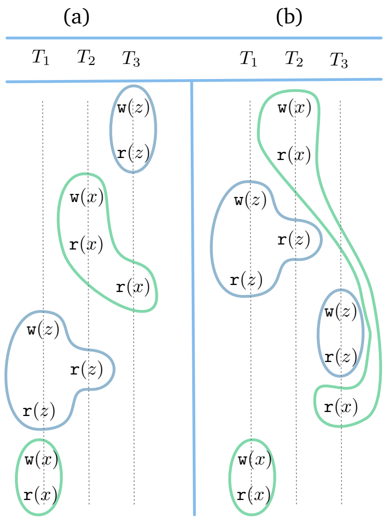

(a)
(b)
T_1 T_2 T_3
T_1 T_2 T_3
w(z) r(z) w(x) r(x) r(x) w(z) r(z) r(z) w(x) r(x)
w(x) r(x) w(z) r(z) r(z) w(z) r(z) r(x) w(x) r(x)

Fig. 2. Commuting Grains

In Section 4, we formally define the set of runs that are soundly equivalent to a program run based on a choice of grains that, as in Fig. 2(a), appear as contiguous subwords of the program run. We observe that different choices of grains can imply the concurrency of different pairs of events. We define *grain concurrency* of two events formally as the existence of a choice of grains and a sound commutativity relation (over the grains) such that the two events can be (soundly) reordered in a *grain-equivalent run* up to those choices.

The construction of a monitor based on trace equivalence vitally relies on the fact that the alphabet of concurrent programs is finite and consequently there are finitely many choices of commutativity relations over this alphabet. Note that, once any subword can become a new entity that participates in commutativity-based reasoning, the number of these subwords is no longer bounded. Neither is the set of possible commutativity relations over them. In Section 5, we prove by construction that *grain-equivalence* can be monitored in constant-space. This construction relies on a key insight that the monitor can maintain *summaries* for these grains that belong to a finite universe, and correctly check the concurrency status of two events.

Let us revisit the example runs in Fig. 2. The run in Fig. 2(b) is the result of commuting two blue grains in the run in Fig. 2(b). The first green grain, however, is no longer a contiguous subword and therefore cannot be seen as a potential grain in the solution we have outlined so far. In Section 6, we formally expand the definition of grains so that these so-called *scattered grains* can be considered as candidates as well. This requires a leap in the definition of the set of words that are soundly equivalent to the input run. In the case of contiguous grains, the set of equivalent runs maintains the characteristic of classic trace theory that one can transform the input run to any inferred equivalent run through a sequence of valid swaps. With *scattered grains*, we establish soundness by deferring to the more general definition of *reads-from equivalence*, and consequently forfeit the characteristic of transformation through swaps.

Surprisingly, however, this weaker definition still maintains the property that it can be monitored in constant-space. First, there is the additional complication that unlike contiguous subwords where at most one grain is *active* (open) at any given time, there may be an unbounded number of *active* scattered grains in a concurrent program run at any given time. Nevertheless, we prove that if an outcome can be decided based on an arbitrary choice of scattered grains, then it can also be decided based on a choice of scattered grains in which the number of active grains is bounded. With a bounded number of active grains (in contrast to just a single one), there is another complication where two active grains, which belong to the middle of a chain witnessing that e and e' are ordered, are only discovered to be ordered long after e and e' have been visited. In Section 7, we construct a monitor that resolves these problems and soundly checks the concurrency of a pair of events based on all possible choices of scattered grains, which we call *scattered grain concurrency.*

Even though the *scattered grain concurrency* monitor subsumes the *grain concurrency* monitor in expressivity, the paper presents these monitors separately, since the former admits a characterization based on swaps and the latter does not. Moreover, this permits us to introduce the relevant ideas in tandem with the problems they help solve. In summary, the paper presents the following results:

- We prove that there is no constant-space algorithm that checks if two arbitrary events are causally ordered or concurrent under the reads-from equivalence relation by establishing a linear space lower bound, a quadratic time-space tradeoff bound, a conditional quadratic time lower bound as well as the non context-freeness of this problem. We complement these lower bounds by showing that the problem can nevertheless be solved in polynomial time as well as in deterministic linear space. (Section 3).
- We propose *grain equivalence* as the means of defining a set of runs that are soundly equivalent to a given program run and form a strictly larger set than the trace equivalence class of the run, and a strictly smaller set than the reads-from equivalence class of it. This notion of equivalence

is constructed based on commutativity of certain pairs of words over the alphabet of concurrent program operations that appear as contiguous subwords of the input program run (Section 4).

- We introduce the notion of *grain concurrency* that attempts to find a witness for causal concurrency of a pair of events based on all possible choices of grains. We further weaken the definition of *grain concurrency* by permitting *scattered grains* to be soundly used for reasoning in equivalence and call this *scattered grain concurrency* (Section 6).
- We give a space efficient algorithm that soundly checks *grain concurrency* for a pair of events, i.e. whether the two events are causally ordered or concurrent up to *grain equivalence* (Section 5), and a space efficient algorithm that soundly checks *scattered grain concurrency* based on all possible choices of scattered grains (Section 7).

2 PRELIMINARIES

A string over an alphabet Σ is a finite sequence of symbols from Σ. We use $|w|$ to denote the length of the string w and $w[i]$ to denote the i^{th} symbol in w. The concatenation of two strings w, w' will be denoted by ww'.

2.1 Trace Equivalence

Antoni Mazurkiewicz popularized the use of partially commutative monoids for modelling executions of concurrent systems[Mazurkiewicz 1987]. We discuss this formalism here. An independence relation over Σ is a symmetric irreflexive binary relation $\mathbb{I} \subseteq \Sigma \times \Sigma$. The Mazurkiewicz equivalence (or trace equivalence) relation induced by $\mathbb{I}$, denoted[1] $\equiv_{\mathcal{M}}$ is then the smallest equivalence over Σ^* such that for any two strings $w, w' \in \Sigma^*$ and for any two letters $a, b \in \Sigma$ with $(a, b) \in \mathbb{I}$, we have $wabw' \equiv_{\mathcal{M}} wbaw'$. A Mazurkiewicz trace is then an equivalence class of $\equiv_{\mathcal{M}}$.

Mazurkiewicz partial order. An equivalence class of $\equiv_{\mathcal{M}}$ can be succinctly represented using a partial order on the set of *events* in a given string. Events are unique identifiers for the different occurrences of symbols in a string. Formally, for a string w, the set of events of w, denoted Events_w is the set of pairs of the form $e = (a, i)$ such that $a \in \Sigma$ and there are at least i occurrences of a in w. Thus, an event uniquely identifies a position in the string — the pair (a, i) corresponds to the unique position $j \leq |w|$ such that $w[j] = a$ and there are exactly $i - 1$ occurrences of a before the index j in w. Observe that if w and w' are permutations of each other, then $\mathsf{Events}_w = \mathsf{Events}_{w'}$. For an event $e = (a, i)$, we use the shorthand $w[e]$ to denote the label a. Often, we will use the position j itself to denote the event (a, i) when the distinction is immaterial. Fix $w \in \Sigma^*$. The Mazurkiewicz (or trace) partial order for w, denoted $\prec_{\mathcal{M}}$ is then, the transitive closure of the relation $\{(e, f) \mid e, f \in \mathsf{Events}_w,\ e \text{ occurs before } f \text{ in } w\ \land\ (w[e], w[f]) \in \mathbb{D}\}$.

For Mazurkiewicz traces, the corresponding partial order is a sound and complete representation of an equivalence class [Mazurkiewicz 1987].

2.2 Concurrent alphabet and dependence

For modeling runs or executions of shared memory multi-threaded concurrent programs, we will consider the alphabet consisting of reads and writes. Let us fix *finite* sets $\mathcal{T}$ and $\mathcal{X}$ of thread

[1]The equivalence is parametric on the independence relation $\mathbb{I}$. In this view, a notation like $\equiv^{\mathbb{I}}_{\mathcal{M}}$ would be more precise. In favor of readability, we skip this parametrization; the independence relation $\mathbb{I}$ will always be clear.

identifiers and memory location identifiers respectively. The concurrent alphabet Σ_{Conc} we consider in the rest of the paper is:

$$\Sigma_{\text{Conc}} = \{\langle t, o, x\rangle \mid t \in \mathcal{T}, o \in \{\mathtt{r}, \mathtt{w}\}, x \in \mathcal{X}\}$$

For a symbol $a = \langle t, o, x\rangle \in \Sigma_{\text{Conc}}$, we say $\mathsf{thr}(a) = t$, $\mathsf{op}(a) = o$ and $\mathsf{var}(a) = x$. A concurrent program *run* or *execution* is a string over Σ_{Conc}. For a run $w \in \Sigma^*_{\text{Conc}}$ and event $e \in \mathsf{Events}_w$, we overload the notation and use $\mathsf{thr}(e)$, $\mathsf{op}(e)$ and $\mathsf{var}(e)$ in place of $\mathsf{thr}(w[e])$, $\mathsf{op}(w[e])$ and $\mathsf{var}(w[e])$ respectively. Since the focus of the rest of the article will be on concurrent program runs, we will omit the subscript Conc, and unless Σ is not explicitly defined, we will assume it is Σ_{Conc}.

We use the following independence (commutativity) relation:

$$\mathbb{I}_{\mathcal{M}} \quad = \quad \Sigma \times \Sigma - \{(a, b) \mid \mathsf{thr}(a) = \mathsf{thr}(b) \vee (\mathsf{var}(a) = \mathsf{var}(b) \wedge \mathtt{w} \in \{\mathsf{op}(a), \mathsf{op}(b)\})\} \qquad (1)$$

This defines an appropriate trace monoid for the alphabet of concurrent program actions. We refer to the equivalence classes a concurrent program run w in this monoid as $[w]_{\mathcal{M}}$. This trace monoid is provably sound in the following sense:

Remark 1. For a given concurrent program run $w \in \Sigma$, every member of $[w]_{\mathcal{M}}$ preserves the *program order* and *reads-from* relations induced by w.

Next, for ease of notation, we will often denote labels as $\langle t, o(x)\rangle$ (for example $\langle t, \mathtt{w}(x)\rangle$) in place of the expanded version $\langle t, o, x\rangle$. We will also, at times, omit the thread identifier and use the shorthand $e = o(x)$ to denote that $\mathsf{op}(e) = o$ and $\mathsf{var}(e) = x$.

2.3 Reads-From Equivalence

A natural notion of equivalence of program runs in the context of shared memory multi-threaded program is *reads-from* equivalence. We formalize this notion here.

Program Order and Reads-from mapping. The program order, or thread order induced by a concurrent program run $w \in \Sigma^*$ orders any two events belonging to the same thread. Formally, $\mathsf{po}_w = \{(e, f) \mid e, f \in \mathsf{Events}_w, \mathsf{thr}(e) = \mathsf{thr}(f), e \text{ occurs before } f \text{ in } w\}$. The reads-from mapping induced by w maps each read event to the write event it observes. In our context, this corresponds to the last conflicting write event before the read event. Formally, the reads-from mapping is a partial function $\mathsf{rf}_w : \mathsf{Events}_w \hookrightarrow \mathsf{Events}_w$ such that $\mathsf{rf}_w(e)$ is defined iff $\mathsf{op}(e) = \mathtt{r}$. Further, for a read event e occurring at the i^{th} index in w, $\mathsf{rf}_w(e)$, is the unique event f occurring at index $j < i$ for which $\mathsf{op}(f) = \mathtt{w}$, $\mathsf{var}(f) = \mathsf{var}(e)$ and there is no other write event on $\mathsf{var}(e)$ (occurring at index j') such that $j < j' < i$. Here, and for the rest of the paper, we assume that every read event is preceded by some write event on the same variable.

Reads-from equivalence. Reads-from equivalence is a semantic notion of equivalence on concurrent program runs, that distinguishes between two runs based on whether or not they might produce different outcomes. We say two runs $w, w' \in \Sigma^*$ are *reads-from equivalent*, denoted $w \equiv_{\mathsf{rf}} w'$ if $\mathsf{Events}_w = \mathsf{Events}_{w'}$, $\mathsf{po}_w = \mathsf{po}_{w'}$ and $\mathsf{rf}_w = \mathsf{rf}_{w'}$. That is, for w and w' to be reads-from equivalent, they should be permutations of each other and must follow the same program order, and further, every read event e must read from the same write event in both w and w'. Reads-from equivalence is a strictly coarser equivalence than trace equivalence for concurrent program runs. That is, whenever $w \equiv_{\mathcal{M}} w'$, we must have $w \equiv_{\mathsf{rf}} w'$; but the converse is not true.

Example 2.1. Consider the two runs (denoted σ and σ') shown in Fig. 1(a) and Fig. 1(b) respectively. Observe that σ and σ' have the same set of events, and program order ($\mathsf{po}_\sigma = \mathsf{po}_{\sigma'}$). Also, for each read event e, $\mathsf{rf}_\sigma(e) = \mathsf{rf}_{\sigma'}(e)$. This means $\sigma \equiv_{\mathsf{rf}} \sigma'$. Consider now the permutation of σ

corresponding to the sequence $\sigma'' = e_1 \dots e_8 e_{10} e_9 e_{11} \dots e_{14}$, where e_i denotes the i^{th} event of σ from the top. σ'' does not have the same reads-from mapping as σ since $\mathsf{rf}_{\sigma''}(e_9) = e_{10} \neq e_7 = \mathsf{rf}_{\sigma}(e_9)$, and thus $\sigma'' \not\equiv_{\mathsf{rf}} \sigma$.

Soundness of Equivalence. The focus of this work is to develop equivalences that are coarser than Mazurkiewicz equivalence and are also *sound*, where soundness will be with respect to reads-from equivalence. Given a run $w \in \Sigma^*$ and a set $S_w \subseteq \Sigma^*$, we say that S_w is sound for w if $S_w \subseteq \{w' \mid w \equiv_{\mathsf{rf}} w\}$. Likewise, an equivalence relation $\sim$ over Σ^* is said to be sound if for every $w \in \Sigma^*$, the equivalence class $[w]_\sim$ is sound for w. Hence, Remark 1 precisely states that trace equivalence defined based on the independence relation $\mathbb{I}_{\mathcal{M}}$ is sound in this sense.

3 CAUSAL CONCURRENCY UNDER READS-FROM EQUIVALENCE

In scenarios like dynamic partial order reduction in stateless model checking, or runtime predictive monitoring, one is often interested in the *causal relationship* between actions. Understanding causality at the level of program runs often amounts to answering whether there is an equivalent run that witnesses the inversion of order between two events.

The efficiency of determining causal concurrency is then key in designing efficient techniques in the aforementioned contexts. When deploying such techniques for monitoring large scale software artifacts exhibiting executions with billions of events, a desirable goal is to design monitoring algorithms that can be efficiently implemented in an online 'incremental' fashion that store only a constant amount of information, independent of the size of the execution being monitored [Roşu and Viswanathan 2003]. In other words, an ideal algorithm would observe events in an execution in a single forward pass, using a small amount of memory. This motivates our investigation of the efficiency of checking causal ordering.

We first formally define the relevant algorithmic questions in the context of any equivalence relation on concurrent program runs, and then study their complexity under reads-from equivalence.

3.1 Causal Concurrency and Ordering

Formally, let $\sim$ be an equivalence over Σ^* such that when two runs are equivalent under $\sim$, they are also permutations of each other. Let $w \in \Sigma^*$ be a concurrent program run, and let $e, f \in \mathsf{Events}_w$ be two events occurring at indices i and j (with $i < j$). We say that e and f in w are *causally ordered* under $\sim$ if for every $w' \sim w$, e and f appear in the same order as they do in w. If e and f are not causally ordered under an equivalence $\sim$, we say that they are causally concurrent under $\sim$.

The following are the key algorithmic questions we investigate in this paper.

Problem 3.1 (Checking Causal Concurrency Between Events)**.** Let $\sim$ be an equivalence relation over Σ^*. Given a program run $w \in \Sigma^*$, and two events $e, f \in \mathsf{Events}_w$, the problem of checking causal concurrency between events asks if e and f are causally concurrent under $\sim$.

In the context of many applications in testing and verification of concurrent programs, one often asks the following more coarse grained question.

Problem 3.2 (Checking Causal Concurrency Between Symbols)**.** Let $\sim$ be an equivalence relation over Σ^*. Given a program run $w \in \Sigma^*$, and two symbols (or letters) $c, d \in \Sigma$, the problem of checking causal concurrency between symbols (or letters) asks to determine if there are events $e, f \in \mathsf{Events}_w$ such that $w[e] = c$, $w[f] = d$ and e and f are causally concurrent under $\sim$.

If one has an oracle for deciding causal concurrency between symbols, then one can use it to check causal concurrency between events. In particular, assume $c = w[e]$ and $d = w[f]$ and consider the alphabet $\Delta = \Sigma \uplus \{c^{\diamond_1}, d^{\diamond_2}\}$, where $c^{\diamond_1}$ and $d^{\diamond_2}$ are distinct marked copies of the letters $c, d \in \Sigma$. Consider the string $w' \in \Delta^*$ with $|w'| = |w|$ such that $w'[g] = w[g]$ for every $g \notin \{e, f\}$, $w[e] = c^{\diamond_1}$ and $w[f] = d^{\diamond_2}$. Then, causal concurrency (respectively orderedness) of symbols $c^{\diamond_1}$ and $d^{\diamond_2}$ under $\sim'$ implies the causal concurrency (respectively orderedness) of events e and f under $\sim$, where the equivalence $\sim'$ is the same as $\sim$, modulo renaming letters $c^{\diamond_1}$ and $d^{\diamond_2}$ to c and d respectively.

3.2 Computational Hardness in Checking Concurrency

For the case of trace equivalence, more specifically under $\equiv_{\mathcal{M}}$, causal concurrency can be determined in a constant space streaming fashion. This result is somewhat known amongst the experts in the field, but it does not appear in the following specific way anywhere in the literature.

Proposition 3.1 (Causal concurrency for trace equivalence). Given an input $w \in \Sigma^*$ and symbols $c, d \in \Sigma$, the causal concurrency between c and d under $\equiv_{\mathcal{M}}$ can be determined using a single pass constant space streaming algorithm.

In Section 5, we give a constructive proof to this lemma by presenting a constant space monitoring algorithm. Next, we show that the same is not achievable for the case of reads-from equivalence — we show that any algorithm for checking causal ordering (for the semantic notion of equivalence $\equiv_{\mathsf{rf}}$) must use linear space in a streaming setting.

Theorem 3.1 (Linear space hardness). Any streaming algorithm that checks the causal concurrency of a pair of symbols under $\equiv_{\mathsf{rf}}$ in a streaming fashion uses linear space, even for program runs containing just 2 threads and 6 variables.

The key idea behind the proof of Theorem 3.1 is to exploit and generalize the intricate pattern in the runs in Fig. 1. The idea is that determining if two specific events are causally concurret in such a pattern, relies on successively inferring the concurrency status of linearly many pairs of events, placed arbitrarily far away in the past and/or in the future, which is impossible for a one pass streaming algorithm that only uses sub-linear space. The formal proof is presented in Appendix A.1.

In fact, we use similar ideas as in the proof of the above statement to also establish a lower bound on the time-space tradeoff for the problem of determining causal concurrency, even when we do not bound the number of passes; formal proof is deferred to Appendix A.1:

Theorem 3.2 (Quadratic time space lower bound). For any algorithm (streaming or not) that checks if a pair of symbols are causally concurrent under $\equiv_{\mathsf{rf}}$ in $S(n)$ space and $T(n)$ time on an input run of length n, we must have $S(n) \cdot T(n) \in \Omega(n^2)$.

The linear lower bound in Theorem 3.1 establishes non-regularity of any monitor for causal concurrency of symbols. We can further refine this result and show that the problem of determining causal concurrency of symbols is also not context-free. We establish the following result by invoking a pumping lemma argument for context free languages on sets of runs that observe intricate patterns akin to the one in Fig. 1.

Theorem 3.3 (Non Context-freeness). Let $c, d \in \Sigma$. There exists no nondeterministic pushdown automaton that accepts exactly the runs w such that both c and d appear in w and are causally concurrent in it under $\equiv_{\mathsf{rf}}$.

Finally, conditioned on the widely believed Strong Exponential Time Hypothesis (SETH) [Impagliazzo and Paturi 2001], we establish a quadratic time lower bound for Problems 3.1 and 3.2.

Theorem 3.4. Assume SETH. The problem of determining the causal concurrency of a pair of events in w under $\equiv_{\mathsf{rf}}$ cannot be solved in time $O(|w|^{2-\epsilon})$ for all $\epsilon > 0$, even when w has 2 threads.

The formal proof of Theorem 3.4 is presented in Appendix A.3. It is established via a fine-grained reduction from the Orthogonal Vector Conjecture, which also admits a quadratic lower bound under SETH [Williams 2005]. The input to the Orthogonal Vectors (OV) problem is a pair of sequences $A, B \subseteq \{0,1\}^d$ of d-dimensional vectors, each of length n (i.e., $|A| = |B| = n$), and the output is YES iff there are two vectors $a \in A, b \in B$ such that their inner product is 0 (i.e., $a \cdot b = 0$). The OV hypothesis states that for every $\epsilon > 0$, there is no algorithm that solves the OV problem (with input instances of length n and dimension d, with $d \in \Omega(\log n)$) in $O(n^{2-\epsilon}\,\mathrm{poly}(d))$ time.

3.3 Upper Bounds for Checking Concurrency

In this section, we study the precise complexity of reasoning about concurrency under $\equiv_{\mathrm{rf}}$ and establish time and space upper bounds. First, we observe that both Problems 3.1 and 3.2 can be solved using an algorithm whose running time is a polynomial expression whose degree varies with the number of threads:

Theorem 3.5 (Polynomial time algorithm)**.** Let $w \in \Sigma$ be a run with $|w| = n$ and let e and f be two events in w. The problem of determining if e and f are causally concurrent under $\equiv_{\mathrm{rf}}$ can be solved in time $O(|\mathcal{T}| \cdot n^{|\mathcal{T}|+1})$. Similarly, given $c, d \in \Sigma$, the problem of determining if c and d are causally concurrent under $\equiv_{\mathrm{rf}}$ can be solved in time $O(|\mathcal{T}| \cdot n^{|\mathcal{T}|+2})$.

The proof (see Appendix A.4) is based on constructing a 'frontier graph' [Gibbons and Korach 1994], whose vertices represent subsets of Events_w which are downward closed with respect to po_w and rf_w, while edges represent valid extensions obtained by adding single events to the subsets.

The problem can also be solved using a linearly bounded Turing machine, which also implies that the language of runs that exhibit concurrency of two given events is a context sensitive language.

Theorem 3.6 (Linear Space Upper Bound)**.** Let $w \in \Sigma$ be a run with $|w| = n$ and let e and f be two events in w. The problem of determining if e and f are causally concurrent under $\equiv_{\mathrm{rf}}$ can be solved in deterministic space $O(n)$. Similarly, given $c, d \in \Sigma$, the problem of determining if c and d are causally concurrent under $\equiv_{\mathrm{rf}}$ can be solved in time deterministic space $O(n)$.

We present the formal proof of Theorem 3.6 in Appendix A.5. It operates by successively generating permutations of the given run and checking if they are equivalent to the input run and also invert the order of the given events, all in deterministic linear space.

4 GRAIN COMMUTATIVITY

In this section, we present a new stronger and more syntactic definition of equivalence for concurrent runs that can overcome the hardness results previously discussed. We build on the theory of traces, where equivalence is defined based on a commutativity relation on the alphabet of program actions. The new equivalence relation is defined based on an extended commutativity relation that additionally allows commutating some pairs of *words* over the same underlying alphabet, and strictly weakens trace equivalence. First, let us briefly discuss that unchecked generalization in this direction can very quickly result in hardness.

Theorem 4.1. Let $\Sigma = \{a, b, c\}$ and let $\mathbb{I} = \{(a, bc), (bc, a), (b, c), (c, b)\}$. There exists no constant space monitor that given an input word $w \in \Sigma^*$ can decide whether the first occurrence of a and the last occurrence of c in w are ordered.

The idea of the proof is to focus on words of the form ab^nc^m, and a causal concurrency query that involves the a and the last occurrence of c. It can be argued that the two are causally concurrent if and only if $n \geq m$. A detailed proof is presented in Appendix B.

Note that here, $\mathbb{I}$ is a generalization of commutativity relation in trace equivalence ($\mathbb{I}_{\mathcal{M}}$) by what seems to be the smallest increment to a classic commutativity relation over letters: the commutativity of one word of length 2 (the shortest possible word that is not a letter) against one single letter. Yet, we immediately loose the constant-space checkability of concurrency/orderedness of pairs of events. Therefore, to maintain the property of constant-space checkability, one has to be careful with the generalization of $\mathbb{I}_{\mathcal{M}}$.

4.1 Partially-Commutative Grain Monoids

We define an equivalence relation based on commutativity of words.

Definition 4.1 (Grains). A *grain* is simply a non-empty word in Σ^*.

For a grain g, let *letters*(g) be the set of letters (from Σ) that appear in g. We can define a partially commutative monoid in the same style as *trace monoids* [Mazurkiewicz 1987] as follows:

Definition 4.2 (Grain Monoids). Given a trace monoid $(\Sigma, \mathbb{I})$, a set of grains G induces the (partially commutative) *grain* monoid $(\Sigma_G \cup \Sigma, \widehat{\mathbb{I}}_G)$ where $\Sigma_G = \{a_g | \; g \in G\}$ and

$$\begin{aligned}\widehat{\mathbb{I}}_G &= \mathbb{I} \cup \mathbb{I}_G \cup \{(a, a_g), (a_g, a) | \; a \in \Sigma \wedge a_g \in \Sigma_G \wedge \{a\} \times \mathit{letters}(g) \subseteq \mathbb{I}\} \\ &\cup \{(a_g, a_{g'}), (a_{g'}, a_g) | \; a_g, a_{g'} \in \Sigma_G \wedge \mathit{letters}(g) \times \mathit{letters}(g') \subseteq \mathbb{I}\}\end{aligned}$$

where $\mathbb{I}_G \subseteq \Sigma_G \times \Sigma_G$, the *grain commutativity* relation, is an arbitrary symmetric independence relation defined on the grains.

If $G = \Sigma$ and $\mathbb{I}_G = \emptyset$, then the induced grain monoid coincides with the trace monoid $(\Sigma, \mathbb{I})$. Otherwise, it can be viewed as a classic trace monoid on a new alphabet $\Sigma_G \cup \Sigma$. For this reason, it induces an equivalence relation $\equiv_{\widehat{\mathbb{I}}_G}$ on the set of words in $(\Sigma_G \cup \Sigma)^*$.

Recall that $\mathbb{I}_{\mathcal{M}}$ is defined in the context of the alphabet Σ to be *sound*, precisely in the sense that the induced $\equiv_{\mathcal{M}}$ preserves rf-equivalence. We need to determine when $\equiv_G$ is considered sound.

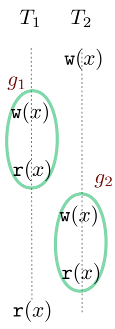
Definition 4.3 (Strict Soundness). We call a grain commutativity relation $\mathbb{I}_G$ *strictly sound* if, $(g, g') \in \mathbb{I}_G$ iff for all $\alpha, \beta \in \Sigma^*$, we have $\alpha g g' \beta \equiv_{\mathsf{rf}} \alpha g' g \beta$.

It is straightforward to see that if we let $G = \Sigma$, then the independence relation that defines trace equivalence is strictly sound according to this definition. The less straightforward fact is that trace equivalence defines the largest such sound relation. To be precise,

Proposition 4.1. If g and g' strictly soundly commute, then $gg' \equiv_{\mathcal{M}} g'g$.

On the one hand, strict soundness seems reasonable because it decouples commutativity from its *context*. On the other hand, it makes the grains seem pointless in the sense that they do not offer any additional commutativity compared to classical trace monoids. The motivation behind forcing the soundness to be independent of the *context* (i.e. the choice of α and β in Definition 4.3) is that as one swaps two commutating grains, the *context* may change, and it would complicate reasoning substantially if the commutativity status of two other grains were to change as a result. In [Sassone et al. 1993], a generalized version of trace monoids are formulated that account for context. These have a sophisticated set of coherence and consistency conditions and have not been studied in any algorithmic contexts beyond being defined.

Fundamentally, we would like a commutativity relation that is sound in the sense that it maintains rf-equivalence and defines an equivalence class in which the commutativity relation does not change from one member to another to keep things simple for formulating algorithms.



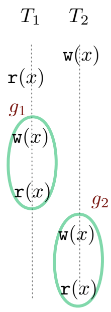
Example 4.1. Consider the program run on the right, with 2 threads. Two grains g_1 and g_2 have been marked. In isolation (as in if everything else from this run is ignored), these two grains soundly commute. But, the run illustrates that in the presence of the last $\texttt{r}(x)$, the two grains do not soundly commute, and therefore they do not strictly soundly commute. Observe that the left context is irrelevant to the commutativity status of these two grains. Only the right context can violate soundness. If unsoundness is the result of the program order being broken, one would observe it by looking only at the two grains. Therefore, any violations to soundness related to the context have to be related to the reads-from relation. Specifically, a write event $\texttt{w}$ belongs to a grain, but there is at least one read event $\texttt{r}$, which *reads* from it (i.e., $\texttt{w} = \texttt{rf}(\texttt{r})$), that does not belong to the same grain. By formally disallowing any such bad right contexts for a pair of grains, one can define a more permissive commutativity relation. □

Example 4.1 illustrates why we put the fault in the definition of strict soundness mainly with the *right* context. To rule these scenarios out, one can restrict the right context from all possible contexts to those that cannot adversarially affect the commutativity status of g and g'.

Definition 4.4 (Sound Grain Commutativity)**.** We call a grain commutativity relation $\mathbb{I}_G$ *sound* if for all $g, g' \in G$, for all $x \in \texttt{var}(g) \cap \texttt{var}(g')$ where at least one $\texttt{w}(x)$ appears in gg', and for all $\alpha, \beta \in \Sigma^*$ such that $\beta|_x \in L(\texttt{w}(x)\Sigma^*)$, we have: $(g, g') \in \mathbb{I}_G \iff \alpha g g' \beta \equiv_{\texttt{rf}} \alpha g' g \beta$

Definition 4.4 strictly weakens Definition 4.3 by limiting the (right) contexts in which commuting the actions must be sound. In other words, every strictly sound grain commutativity relation is sound, but not all sound grain commutativity relations are strictly sound. In particular, if g and g' do not strictly soundly commute, it is fairly straightforward to construct a right context β in which they do not soundly commute.

Remark 2. Given a run w and another run u that can be acquired from w through a sequence of swaps defined by a *strictly* sound commutativity relation $\widehat{\mathbb{I}}_G$ (Definition 4.3), we have $w \equiv_{\texttt{rf}} v$. This is not true for a sound commutativity relation (Definition 4.4).



Example 4.2. Recall the run illustrated in Example 4.1. According to Definition 4.4 and as discussed in Example 4.1, the two grains g_1 and g_2 soundly commute. But, clearly, the equivalence class of the run from Example 4.1 up to this commutativity relation is not sound. In contrast, the same two grains appear in the run illustrated on the right and the equivalence runs inferred by their commutativity are sound. The difference is that the run on the right does not violate the condition about right contexts (from Definition 4.4) but the run from Example 4.1 does. □

It feels like we took one step forward by weakening the definition of soundness and then one step backward since it is not guaranteed to provide soundness in all contexts. There are, however, two key observations that make this definition of soundness a good fit in the context of our main goal, that is checking the status of concurrency of two given events. First, we are solely interested in the set of words equivalent to a single reference program run. Second, we may not have control over the choice of right contexts, but we do have control over the choice of grains and the commutativity relation $\mathbb{I}_G$. We can limit these choices based on the input run so that the equivalence class of the input run induced by the corresponding grain monoid is indeed sound. Therefore, we next focus on the equivalence class $[w]_G$ of a given word w and the choices for G (the set of grains) and the grain commutativity relation $\mathbb{I}_G$ that make $[w]_G$ sound.

4.2 Sound Grain Equivalence

First, observe that the same exact grain (which is a word) can appear several times as a subword in a particular word. In our formalism so far, we had no need of distinguishing the multiple instances, but since their right contexts may differ, we must do so now to attach different commutativity attributes to them.

Definition 4.5. A *valid* set of grains for a run $w \in \Sigma^*$ is set of indexed words of the form $g@i$ where g is a contiguous subword of w that appears at position i and no two grains overlap in w.

Consider a valid set of (indexed) grains G in a program run w. Since the indexed set G may only be a valid set for the run w in consideration, we cannot cleanly define the equivalence induced by G on the set of all runs Σ^*. However, we can still precisely define the class of runs that can be inferred by successively swapping adjacent grains from G. For ease of presentation, we will abuse the notation $[w]_G$ to denote this set, and will take the liberty to call it the *equivalence class of* w, when G is known from the context.

We can formalize $[w]_G$ as follows. Define $h_G : \Sigma^* \to (\Sigma_G \cup \Sigma)^*$ as the homomorphism that maps each indexed grain $g@i$ to the corresponding letter $a_g \in \Sigma_G$ and each letter in Σ, that does not belong to a grain, to itself. Let h_G^{-1} be the inverse homomorphism that replaces letters in Σ_G with the corresponding grain words. Then,

$$u \in [w]_G \iff \exists u' \in (\Sigma_G \cup \Sigma)^* : \left(u = h_G^{-1}(u') \wedge u' \equiv_{\widehat{\mathbb{I}}_G} h_G(w)\right)$$

In short, the set of words that are considered equivalent to w are determined by those that are equivalent to its corresponding word in the grain monoid, for the specific choice of valid grains G. In prose, we say u is equivalent to w when $u \in [w]_G$ for some valid choice of grains G.

We lift the commutativity relation $\mathbb{I}_G$ to relate indexed words of the form $g@i$ as well. This will enable us to say that $(g@i, g'@j) \in \mathbb{I}_G$ while $(g@k, g'@j) \notin \mathbb{I}_G$. Note that corresponding grain monoid is defined as before, each new grain $g@i$ is mapped to a designated letter $a_{g@i}$.

Definition 4.6. For a word w and a set of valid grains G in w, $\mathbb{I}_G$ is sound if $[w]_G$ is sound (i.e. $[w]_G \subseteq [w]_{\mathsf{rf}}$).

Next we give necessary and sufficient conditions for soundness of $\mathbb{I}_G$. For an event $e = \mathsf{w}(x)$, let *reads*(e) denote all events e' of the form $\mathsf{r}(x)$ that read from e. To lighten the notation whenever possible, we may refer to a grain only by its word g whenever the position is not of importance, or implied from the context. We only specifically mention the position $g@i$ when it really matters. Define $\mathsf{op}(g, x)$, for a grain g to be the set of operations ($\{\mathsf{r}, \mathsf{w}\}$) of variable x that appear in g.

Theorem 4.2. In the context of a word w and a *valid* set of grains G, a commutativity relation $\mathbb{I}_G$ is sound iff for all $(g, g') \in \mathbb{I}_G$, $gg' \equiv_{\mathsf{rf}} g'g$ and for all $x \in \mathsf{var}(g) \cap \mathsf{var}(g')$ the following holds:

$$\big(\mathsf{op}(g,x) \cup \mathsf{op}(g',x) = \{\mathsf{r}\}\big) \vee \big(\forall e, f \cdot (e = \mathsf{w}(x), f = \mathsf{r}(x), f = \mathsf{rf}_w(e)) \implies (e \in g \iff f \in g)\big)$$

It is clear that if $gg' \equiv_{\mathsf{rf}} g'g$ does not hold, the resulting commutativity relation is unsound. Example 4.1 captures the idea of why the violation of the additional conditions also leads to unsoundness. Intuitively, these conditions express that if the grains share a variable and at least one writes to that shared variable, then once a write action is included in one grain then all reads that read from it also must be included in that grain. The proof of the other direction is a tedious case analysis and given in Appendix B

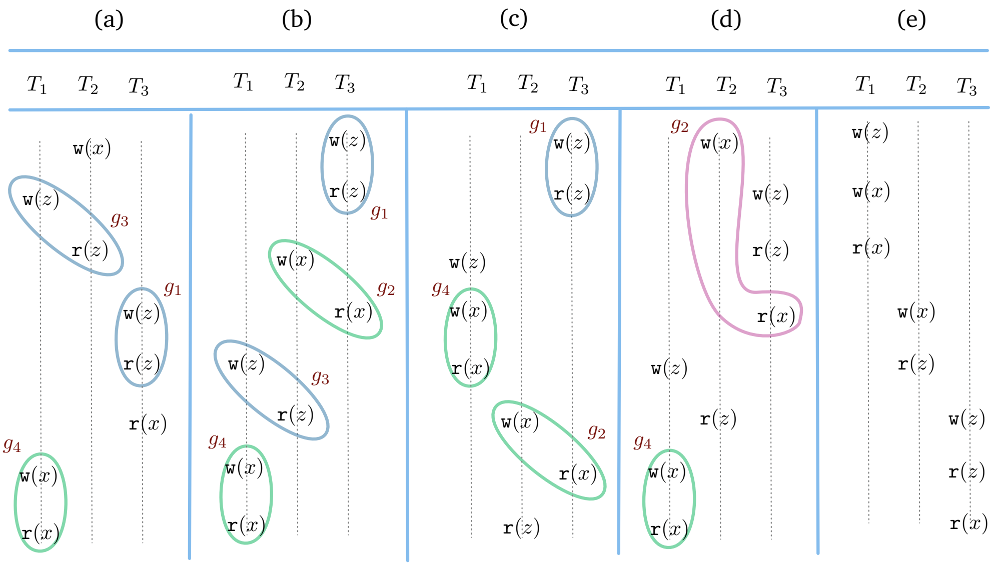


Fig. 3. Examples 4.3, 6.1, and 6.2.

Recall the program runs in Examples 4.1 and 4.2. Observe that even though the pairs of grains are identical, if we assume the commutativity of the pair of grains, then the choice in Example 4.2 satisfies the conditions of the above theorem, and the one in Example 4.1 does not; in particular, they violate the part that demands every read that reads from the same w(x) operation must belong to the grain.

4.3 The Expressive Power of $[w]_G$

We use an extended example to highlight how $[w]_G$ soundly enlarges the trace equivalence class of w, induced by the trace equivalence relation $\equiv_{\mathcal{M}}$.

Example 4.3. Consider the program runs illustrated in Fig. 3. They are all rf-equivalent. We observe that different choices of grains witness the equivalence of different pairs of the run from the figure. Independent of which grain is present in which subfigure, the only sound commutativity between grains, in addition to classic trace theory commutativity, is $\mathbb{I}_G = \{(g_1, g_3), (g_3, g_1), (g_2, g_4), (g_4, g_2)\}$.

First, focus on Fig. 3(a), and observe that g_1 and g_3 soundly commute (in the sense of Theorem 4.2). Yet g_4 does not commute with anything that it would not otherwise under the classic trace monoid through the commutativity of its individual events. For example, taking the single w(x) of thread T_2 or the single r(x) of thread T_3 as grains, one would violate the conditions of Theorem 4.2) if one were to declare either event commutative against g_4. With the grains marked in Fig. 3(a), the run illustrated in Fig. 3(a) is equivalent to the one in Fig. 3(b) — starting from (a), we can swap g_1 and g_3 first, and then g_1 against the w(x) of thread T_2 and g_3 against the r(x) of thread T_3.

Now consider the set of grains in Fig. 3(b). It is a superset of grains marked in Fig. 3(a), and yet, in this configuration we have less freedom of movement. $\mathbb{I}_G$ is still sound, but since (g_1, g_2), (g_2, g_3), and (g_3, g_4) do not commute, this run effectively belongs to an equivalence class of size 1. Specifically, we cannot conclude that it is equivalent to the run illustrated in Fig. 3(a). If we remove grain g_2, then the equivalence class gets larger, and includes the run illustrated in Fig. 3(a). Observe that, therefore:

Having more grains does not necessarily *imply having a larger equivalence classes.*

Similarly, if we take the set of grains in Fig. 3(c), then the run in Fig. 3(c) is equivalent to the one in Fig. 3(b). But, note that there is no possible choice of grains in Fig. 3(c) that would make it equivalent to the run in Fig. 3(a), and vice versa. The two runs are clearly rf-equivalent. They are grain-equivalent to the run in Fig. 3(b). Yet, for no choice of grains they can be made grain-equivalent to each other.

If $v \in [u]_G$ and $w \in [v]_{G'}$, there may not exist a sound G'' such that $w \in [u]_{G''}$.

Example 4.3 illustrates that depending on the input run w, or the choice of events for a query of concurrency, a different choice of grains G may be suitable to define the appropriate $[w]_G$.

Definition 4.7 (Grain Concurrency). Consider a program run w and two events e and e' that appear in w. We call the pair of events e and e' to be *grain concurrent* iff there exists a sound set of grains G and a commutativity relation $\mathbb{I}_G$ in the context of w such that, there exists $u \in [w]_G$ in which e and e' are reordered with respect to their order of appearance in w.

Note that for any given program run w and any choice of valid grains G and a sound commutativity relation $\mathbb{I}_G$, $[w]_G$ is always, by definition, a superset of the trace equivalence class of w. Moreover, if we let $G = \Sigma$, then $[w]_G$ coincides with the trace equivalence class of w, since the $\mathbb{I}_{\mathcal{M}}$ is by definition sound. As such, any two events that are concurrent according to trace equivalent are also grain concurrent.

In Section 5, we formally argue that grain concurrency can be checked in constant space by giving a construction for a monitor that is strictly more expressive than a constant-space monitor based on $\equiv_{\mathcal{M}}$. For example, our monitor would declare that both the pair of $\mathtt{w}(x)$ and the pair of $\mathtt{w}(z)$ operations in the run illustrated in Fig. 3(b) can be soundly reordered, while they are strictly ordered according to $\equiv_{\mathcal{M}}$.

5 GRAIN CONCURRENCY MONITOR

Grain monoids are closely related to trace monoids. Therefore, we begin by defining a monitor that in constant space checks whether two events in an input run are concurrent according to $\equiv_{\mathcal{M}}$, and then present the grain concurrency monitor as an extension of this monitor.

To have a simple setup, we augment our alphabet Σ with two new symbols $\diamond_1$ and $\diamond_2$ that are assumed to appear precisely once each in any input word, marking the two events that are meant to be checked for concurrency; these would be the events that immediately succeed $\diamond_1$ and $\diamond_2$. The regular expression $\Sigma^* \diamond_1 \Sigma^+ \diamond_2 \Sigma^+$ captures that the the two $\diamond$'s are properly placed in an input run. Therefore, in the description of our main monitor, we assume that the input run is already well-formed in this sense.

The high level idea behind the monitor is simple. The monitor idles until it sees the first marker $\diamond_1$ and the event e_1 that it marks. Afterwards, it maintains a summary of the set of operations, reads or writes to specific variables by specific threads, seen so far that are *ordered* wrt to e_1. When the monitor comes across $\diamond_2$ and therefore identifies the second event e_2, it can use the summary to determine if e_1 and e_2 are causally concurrent or ordered.

The key to constant-space implementability is that the monitor does not have to remember all individual events, but rather what variables and threads are involved. This is based on the observation

State	Event	State Update
$\langle -, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle -, \mathrm{C}\rangle$
$\langle -, \mathrm{C}\rangle$	$\diamond_1$	$\langle \diamond_1, \mathrm{C}\rangle$
$\langle \diamond_1, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \diamond_1 -, \{e\}\rangle$
$\langle \diamond_1 -, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \diamond_1 -, \mathrm{C} \odot e\rangle$
$\langle \diamond_1 -, \mathrm{C}\rangle$	$\diamond_2$	$\langle \diamond_1 - \diamond_2, \mathrm{C}\rangle$
$\langle \diamond_1 - \diamond_2, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \mathit{Ord}(\mathrm{C}, e), \mathrm{C}\rangle$
$\langle \mathit{false}, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \mathit{false}, \mathrm{C}\rangle$

$$\mathrm{C} \odot e = \begin{cases} \mathrm{C} \cup \{e\} & \text{if } \exists e' \in \mathrm{C} : (e, e') \in \mathbb{D}_{\mathcal{M}} \\ \mathrm{C} & \text{owise} \end{cases}$$

$$\mathit{Ord}(\mathrm{C}, e) \iff \exists e' \in \mathrm{C} : (e, e') \in \mathbb{D}_{\mathcal{M}}$$

Fig. 4. Trace Concurrency Monitor: The monitor starts in state $\langle -, \varnothing\rangle$ and accepts if in a state $\langle \mathit{false}, \mathrm{C}\rangle$ (for any C) once the input is read. The operation $\odot$ updates C based on a new event.

that to determine if two events commute, it suffices to know what variables are being accessed, what the nature of the access is, and to what threads the two events belong. The summary can be maintained in the most compact manner if the list of operations $\mathtt{op}(x)$ and the list of thread identifies are kept separately. But, we present the less compact version that maintains the summary as a set of events, since it is easier to generalize this version of the monitor to *grains*.

Formally, the monitor's state is pair $\langle d, \mathrm{C}\rangle$ where $\mathrm{C} \subseteq \Sigma$ is a set of labels. The first element of the pair d is used to track whether the monitor has seen events e_1 and e_2 yet. It encodes the six distinct stages: $-$: before the first diamond, $\diamond_1$: right after the first diamond, $\diamond -$: after e_1 has been recorded, $\diamond_1 - \diamond_2$: right after the second diamond is seen, and *true*/*false*: depending on the monitors decision to accept/reject after reading e_2. Fig. 4 lists the transitions of the monitor and the functions used. Since the result is folklore, we forgo giving a proof for the correctness of this monitor.

5.1 A Monitor for a Fixed Set of Grains G

We introduce our monitor based on grains in two stages, for the simplicity of presentation. First, we assume a set of grains is pre-decided and pre-marked in an input word, and present the core idea behind monitoring causal concurrency in this setup. Then, in Section 5.2, we build the final grain concurrency monitor as a generalization of this one.

Assume that Σ is further extended with a pair of symbols $\rhd$ and $\lhd$, which are used as delimiters to mark grain boundaries. Any letter that appears outside the range of these delimiters is treated as a standalone event. For example, the program run form Fig. 3(a) with the marked grains becomes:

$$\langle T_2, \mathtt{w}(x)\rangle \rhd \langle T_1, \mathtt{w}(z)\rangle\langle T_2, \mathtt{r}(z)\rangle \lhd \rhd\langle T_3, \mathtt{w}(z)\rangle\langle T_3, \mathtt{r}(z)\rangle \lhd \langle T_3, \mathtt{r}(x)\rangle \rhd \langle T_1, \mathtt{w}(x)\rangle\langle T_1, \mathtt{r}(x)\rangle\lhd$$

The two events of interest are marked with $\diamond$'s, as before. Except that if either event belongs to a grain, then the diamond must mark the entire grain. For example, if we want to determine whether the two $\mathtt{w}(x)$ events are ordered in the program run above, the diamonds would mark the first one as before, and the second one behind the left grain delimiter like this:

$$\diamond_1\langle T_2, \mathtt{w}(x)\rangle \rhd \langle T_1, \mathtt{w}(z)\rangle\langle T_2, \mathtt{r}(z)\rangle \lhd \rhd\langle T_3, \mathtt{w}(z)\rangle\langle T_3, \mathtt{r}(z)\rangle \lhd \langle T_3, \mathtt{r}(x)\rangle \diamond_2 \rhd\langle T_1, \mathtt{w}(x)\rangle\langle T_1, \mathtt{r}(x)\rangle\lhd$$

Note that the events in a grain always move together. Therefore, one cannot have a verdict that an event e (e.g. the first $\mathtt{w}(x)$ above) is concurrent with some arbitrary event f of a grain (e.g. the second $\mathtt{w}(x)$ above), while it is ordered wrt another event f' of the same grain (e.g. the second $\mathtt{r}(x)$ above). The following (revised) regular expression captures all the input runs in which grains and $\diamond$'s are marked properly, and therefore, we do not make it part of the design of the monitor:

$$\left((\rhd\Sigma^+\lhd) + \Sigma\right)^* \diamond_1 \left((\rhd\Sigma^+\lhd) + \Sigma\right)^+ \diamond_2 \left((\rhd\Sigma^+\lhd) + \Sigma\right)^+ \qquad \text{(WF)}$$

To generalize the monitor in Fig. 4 to work with grains, we face two sources of complications. First, grains are arbitrary words in Σ^+, and even though Σ is finite, Σ^+ is infinite in size. Second, there are (potentially) infinitely many sound commutativity relations that can be inferred over the unbounded set of potential grains.

A simple observation helps overcome the first problem. Fundamentally, we are interested in grains because we are interested in the commutativity properties of these grains. Theorem 4.2 captures what information is relevant to make a sound decision about the commutativity of two grains. According to Theorem 4.2, g and g' soundly commute if $gg' \equiv_{\mathsf{rf}} g'g$ **and** for every variable x accessed in g (respectively g'), where x is written in at least one of the two grains, the following predicate is true:

$$\mathsf{complete}(g, x) \overset{\text{def}}{=} \big(\forall e, f \cdot (e = \mathsf{w}(x), f = \mathsf{r}(x), e = \mathsf{rf}_w(f)) \implies (e \in g \Longleftrightarrow f \in g)\big) \tag{2}$$

In other words, two grains commute, if they commute according to $\equiv_{\mathsf{rf}}$ in isolation and if the share a variable that is written by at least one of them, then both grains must be *complete* wrt to that variable. We now introduce the *signature* of a grain as a pair of a set of letters that appear in the grain and a set of variables wrt which the grain is complete:

$$\forall w \in \Sigma^+ : \; \mathit{grain}(w) \overset{\text{def}}{=} \langle \mathit{letters}(w), \{x \in \mathcal{X} \mid \mathsf{complete}(w, x)\}\rangle$$

which contains all the information required for deciding commutativity of two given grains. More importantly, observe that there are boundedly many different grain signatures, $2^{|\Sigma|+|\mathcal{X}|}$ to be exact. Therefore, in the summary (C in Fig. 4), rather than keep track of the grain as an arbitrary size word, we maintain a set of grain signatures, which is very much bounded.

Let us turn our attention to the second problem. For any pair of grains in a word, one can look at their signatures and soundly decide whether they commute or not. Yet, there are unboundedly many such grains, and therefore, unboundedly many such commutativity relations to enumerate for the definition of *grain concurrency* (Definition 4.7).

Observe that commutativity and causal concurrency are monotonically related: the larger the grain commutativity relation, the larger the set of pairs of grain concurrent events. Therefore, rather than enumerate all possible commutativity relations, one can conservatively choose the largest one. In this largest relation, any two grains, that can soundly commute, are assumed to be commuting. This is determined based on their signatures alone, and therefore, the number of possible choices for the *largest* commutativity relation is bounded because the number of distinct signatures is bounded.

The Monitor. One can conceptually think about the monitor combining the following two passes into a single pass through nondeterminism:

- Pass 1: From right to left, analyze the grains and replace each with a fresh letter (corresponding to its signature), and learn the bounded maximal commutativity relations for these new letters. Intuitively, this pass finds out which grains are complete wrt which variables, constructs their signature, and replaces the grains with a new letter that encodes the information from the signature. Constructing signatures is straightforward in a left-to-right or right-to-left pass, but it is more straightforward to see why completeness (condition 2) can be checked and encoded for each grain in a right-to-left pass: a violation of this condition manifests as a pending read's matching write appearing in a grain.
- Pass 2: In the style of the trace concurrency monitor (Fig. 4)), in a left to right pass, decide the causal concurrency of the two entities marked by $\diamond$'s based on the original letters and the new letters computed during pass 1.

Classic ideas from automata theory provide the recipe to combine these passes, through nondeterminism, into a single-pass (left to right) constant space monitor that decides causal concurrency between any two events based on the largest sound grain commutativity relation:

Theorem 5.1. For a fixed set of valid grains G and a largest sound commutativity relation $\mathbb{I}_G$, the monitor sketched above accepts a program run $u \diamond_1 e\ v \diamond_2 e'w$ iff e and e' are reordered in some member of $[u \diamond_1 e\ v \diamond_2 e'w]_G$.

The proof together with the details of the monitor are presented in Appendix C.1.

5.2 Grain Concurrency Monitor

We are now ready to construct the monitor that precisely captures Definition 4.7. This monitor nondeterministically guesses the $\triangleright$ and $\triangleleft$ symbols and therefore the grain boundaries, and for each guess runs the monitor in Fig. 9. It has to maintain a state to make sure that the grains are well-formed (non-overlapping) and non-empty. Therefore it effectively makes a guess and checks that its guess belongs to the language of the regular expression WF. Note that this guessing must account for the $\diamond$'s. The monitor makes a guess that an event of interest will be in a grain that it just nondeterministically opened, and therefore marks it with a diamond. Naturally, all wrong guesses are refuted later when the grain closes without seeing an event of interest.

Theorem 5.2. There exists a monitor that can decide, in constant space, the (causal) grain concurrency (respectively orderedness) of two events in a given program run.

6 SCATTERED GRAINS

So far, we have formally defined grains as subwords of a concurrent program run. Let us revisit our examples from Fig. 3 to motivate expanding the definition to include grains that do not appear as contiguous subwords; we call these *scattered grains*.

Example 6.1. To argue for the equivalence of the run illustrated in Fig. 3(d) to the one in Fig. 3(c), we need the $\mathtt{w}(x)$ of T_2 to first commute as an individual event (from (d) to (b)), and then move as part of the grain that is marked in Fig. 3(c). If we are permitted to consider the *scattered grain* g_2 as a grain in Fig. 3(d) (marked in pink), then we can argue that $\mathtt{w}(x)$ of thread T_2 can be reordered against $\mathtt{w}(x)$ of thread T_1 by (eventually) swapping the corresponding grains g_2 and g_4.

The grain monitor we present in Section 5 cannot keep track of these dual roles. Starting from the run illustrated in Fig. 3(d), it cannot see the potential of the grain including the $\mathtt{w}(x)$ of T_2 and $\mathtt{r}(x)$ of T_3 forming after a few sound swaps, and therefore cannot reason that $\mathtt{w}(x)$ of T_2 can ultimately be soundly reordered against $\mathtt{w}(x)$ of T_1. □

Formally, we say $\mathbf{i}$ is subsequence of the sequence of the range $[1..n]$ if it is strictly increasing, and all its elements belong to $[1..n]$. For convenience, we treat subsequences as sets of their elements (without order) when appropriate.

Definition 6.1 (Scattered Grains)**.** A *scattered grain* of program run w is a subsequence of w. To distinguish identical scattered grains from each other, we denote them as $g@\mathbf{i}$, where $\mathbf{i}$ is a subsequence of $[1..|w|]$ and identifies the position of g. A set of scattered grains G for a word $w \in \Sigma^*$ is said to be *valid* if no two distinct grains overlap, that is, $g_1@\mathbf{i}_1 \neq g_2@\mathbf{i}_2 \in G \implies \mathbf{i}_1 \cap \mathbf{i}_2 = \varnothing$.

Observe that scattered grains generalize the definition of grains when the subsequences happen to be contiguous. We may refer to a scattered grain simply as g rather than $g@\mathbf{i}$ whenever the position is unimportant or clear from the context. Sound commutativity relations over scattered grains are defined identically to contiguous grains, therefore all definitions and theorems from

Section 4 hold. Moreover, we assume that single events form grains of size one and therefore every event belongs to some grain in what follows.

Definition 6.2 (Grain Graph of a Run)**.** Let G be a valid set of scattered grains for a program run w and $\mathbb{I}_G$ be the largest sound commutativity relation over G in the context of w. The grain graph $\mathcal{G}_{w,G} = (V, E)$ is a directed graph defined with the set of nodes $V = G$ and the set of edges

$$E = V \times V - \{(v_1, v_2) \mid v_1 = v_2 \vee (v_1, v_2) \in \widehat{\mathbb{I}}_G \vee w|_{v_1,v_2} \equiv_{\mathcal{M}} v_2 v_1\}$$

where $w_{v_1,v2}$ is the projection of w to the content of the grains v_1 and v_2.

The first two sets of excluded edges correspond to the classic notions of anti-reflexivity and independence. The third condition above determines when there is an edge between two scattered grains that are *entangled.* We want a directed edge only if the second grain cannot be safely commuted to before the first grain.

Grain graphs can be used to define a notion of concurrency based on a set of scattered grains in the following sense:

Definition 6.3 (Grain Graph Concurrency)**.** Let w be a run, G be a valid set of scattered grains in w and $\mathcal{G}_{w,G}$ be the corresponding grain graph. Let e_1 and e_2 be events in w such that e_1 appears before e_2 in w. We say that the events e_1 and e_2 are *grain graph concurrent* under G if there is no path in $\mathcal{G}_{w,G}$ from the node containing e_1 to the node containing e_2.

We call a valid set of scattered grains G and the corresponding commutativity relation $\mathbb{I}_G$ *sound* in the context of a run w if the same conditions listed in Definition 4.4 hold.

Observe that for contiguous grains, soundness of grain concurrency was baked into the definition that $[w]_G$ is sound. With scattered grains, this is no longer the case, and hence we need the following theorem:

Theorem 6.1. (Soundness of Grain Graph Concurrency) Let w be a run, G be a valid set of scattered grains in w. If a pair of events e_1 and e_2 are grain graph concurrent under G, then they appear in a different order in some run u such that $u \equiv_{\mathsf{rf}} w$.

The proof (given in Appendix D) relies on the construction of the *condensation* of $\mathcal{G}_{w,G}$; that is, the directed *acyclic* graph acquired from $\mathcal{G}_{w,G}$ by contracting all its maximal strongly connected components. The proof argues that any linearization of this *condensed* graph is rf-equivalent to w. This, in turn, means that the condensed graph is analogous to a partial order representing a an equivalence class of $\equiv_{\mathcal{M}}$ induced by the trace commutativity relation $\mathbb{I}_{\mathcal{M}}$. However, it differs from it in two important ways: (1) even though every linearization is rf-equivalent to w, it is not guaranteed to be equivalent to w up to a sequence of valid grain and letter swaps, and (2) the set of linearizations does not necessarily include everything that is (grain) equivalent to w, even possibly w itself.

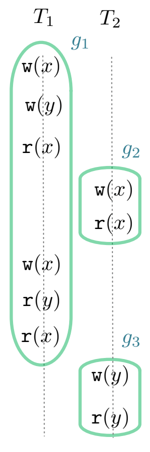


Fig. 5. Entangled Grains

Consider the grains illustrated in Fig. 5. They can be used to argue that the first w(x) and the last w(y) are grain graph concurrent. The grain graph only has one edge between grains g_2 and g_3, since all other pairs commute. However, observe that because g_2 is somewhat *entangled* with g_1, there exists no swap sequence to witness this concurrency. Moreover, the illustrated run itself does not belong to any linearization of the (condensed) grain graph, since no such linearization can reproduce the *entanglement* of the grains of the illustrated run.

Even though contiguous grains are a special case of scattered grains, the way we define *concurrency* in the two cases are fundamentally different in Definitions 4.7 and 6.3. Yet, for a set of contiguous grains, *grain graph concurrency* coincides with *grain concurrency.*

Theorem 6.2. For a program run w and a sound valid set of contiguous grains G, a pair of events e and e' are *grain concurrent* under G iff they are *grain graph concurrent* under G.

This is the consequence of the fact that $\mathcal{G}_{w,G}$ is an acyclic graph for a valid set of *contiguous* grains G, and as such the condensed graph and $\mathcal{G}_{w,G}$ coincide, and are identical to the partial order describing the same equivalence class in the corresponding grain monoid, for which a valid swap sequence can be constructed.

As with Definition 4.7, two events may be graph grain concurrent under one choice of scattered grains G but not under another choice G'. Consequently we define the following more permissive notion of concurrency under scattered grains.

Definition 6.4 (Scattered Grain Concurrency)**.** Consider a program run w and two events e and e' that appear in w. We call the pair of events e and e' to be *scattered grain concurrent* if there exists a valid set of scattered grains G for w such that e and e' are grain graph concurrent under G.

The Theorem 6.2 also implies that *scattered grain concurrency* properly subsumes *grain concurrency.* In Section 7, we demonstrate how *scattered grain concurrency* can be monitored in constant space. With the following example, we make the observation that even though *scattered grain concurrency* is strictly weaker than *grain concurrency*, it strictly under-approximates *sound concurrency* defined based on rf-equivalence.

Example 6.2. There remains a fundamental gap between the notion of concurrency defined based on rf-equivalence and scattered grain concurrency: in the run in Fig. 3(b), no choice of grains would witness the fact that the $\mathtt{w}(z)$ operation of thread T_3 can be soundly reordered against the $\mathtt{w}(x)$ operation of thread T_1. If all 4 grains are present, then the events are ordered. If we take either g_1 or g_3 out, then they become ordered through the conflict dependencies between the x operations. If we take either g_2 or g_4 out, then they become ordered through the conflict dependencies between the z variables. Yet, the rf-equivalent run in Fig. 3(e) witnesses that they are soundly concurrent.

Interestingly, if we focus on the run in Fig. 3(e), and assume all grains g_1, g_2, g_3, and g_4 are present in it as scattered grains, then we can reason using the induced grain graph that the run in Fig. 3(b) is linearization of its (condensed) grain graph and as such $\mathtt{w}(x)$ of thread T_1 is (scattered) grain concurrent with $\mathtt{w}(z)$ of T_3. Therefore, in the non-swap-based notion of *scattered grain concurrency*, one can reason about the implied equivalence and the corresponding notion of the runs in Figures 3(b,e) in one way but the inverse. □

7 MONITORING WITH SCATTERED GRAINS

In this section, we develop a monitor for checking concurrency of two events in the presence of scattered grains. When grains are scattered, they can be interleaved in the run w, and this poses fundamental challenges towards the design of a constant space monitor.

We call a grain *active* in a prefix of a concurrent run, if part of the grain has appeared in the prefix, but it is has not appeared in its entirety. With contiguous grains, at most one grain can be active at any given time. In sharp contrast, the number of scattered grains that may be *active*

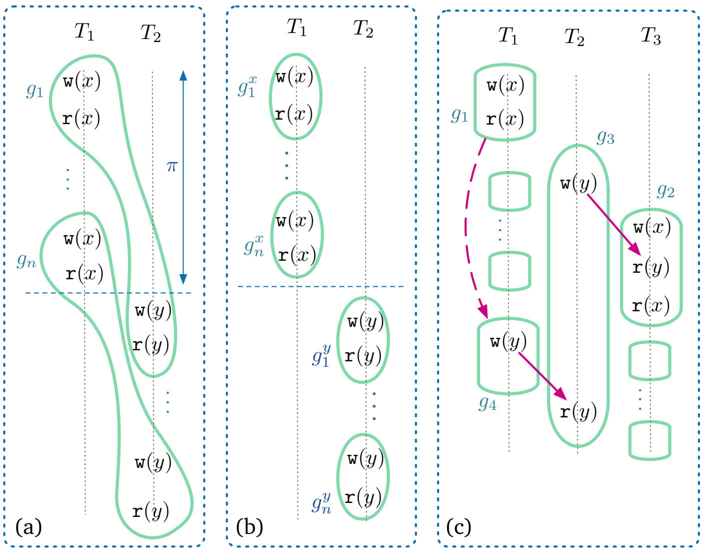


Fig. 6. The challenges of monitoring with scattered grains: (a) Unboundedly many active grains, (b) Minimal grains, and (c) Retroactive paths.

simultaneously, can be unbounded (i.e., not constant). Consider the run and the scattered grains $G = \{g_1, \ldots, g_n\}$ marked in Fig. 6(a). Observe that all grains $g_1, g_2, \ldots g_n$ are active in the prefix π. The unboundedness of the number of active grains is problematic because one expects that a monitoring algorithm for checking concurrency would need to at least keep track of all active grains. This is the first challenge that has to be overcome in designing a constant space monitor for scattered grain concurrency.

The second challenge arises because a single active grain may overlap with many other active grains through its lifetime, even if at a given point only boundedly many grains are active. This means that two grains can be observed as ordered witnessed by a path in the grain graph (Definition 6.2), but this path is completed by a grain that appears long after the lifespan of both the grains have ended. Consider, for example, the run in Fig. 6(c). Before the grain g_4 appears, there is a path, namely the direct edge, from g_3 to g_2, but no path from g_1 to g_2. In fact, right before g_4 appears, g_1 and g_2 have both become inactive. When g_4 appears, a path is formed from g_1 to g_4. Once the $\mathsf{r}(y)$ event in g_3 appears, an edge is formed between grain g_4 and grain g_3, which now completes the path from g_1 to g_2 *retroactively*. Accounting for such retroactive paths is necessary for soundness.

We present solutions to these two challenges in Sections 7.1 and 7.2. We introduce the notion of a *minimal* grain to deal with the problem of unboundedly meany active grains and then demonstrate how a monitor can track retroactive paths by *summarizing* the paths between active grains when an intermediate grain has finished.

7.1 Bounding the number of Active Scattered Grains

We start by defining a class of grains that are *minimal*, with the idea that if all (scattered) grains are minimal, then the number of active grains in any prefix of the program run is bounded.

Definition 7.1 (Minimal Grain). A (scattered) grain g is said to be minimal in program run w if for all $\rho, \sigma \neq \epsilon$ such that $g = \rho\sigma$, there is a read event $\mathsf{r} \in \sigma$ where $\mathsf{rf}_w(\mathsf{r}) \in \rho$. We call a set of grains minimal in w if all grains in it are minimal in w.

Intuitively, a grain is minimal if it cannot be broken into simpler grains, without worsening its commutativity status with respect to other grains. In Fig. 6(a), none of the grains $g_1, \ldots, g_n$ are minimal, because each of them have a prefix (of size 2) where there is no read event whose corresponding write event is not in the prefix grain. In contrast, the following regular expression can produce arbitrarily long minimal grains:

$$\mathtt{w}(x)\mathtt{w}(y)\mathtt{r}(x)\Big(\mathtt{w}(x)\mathtt{r}(y)\mathtt{w}(y)\mathtt{r}(x)\Big)^*$$

For a valid set G of grains in w and a prefix π of w, we use the notation $\mathsf{Active}_{\pi,G}$ to denote the set of grains in G that are active in π. If all grains are minimal, then the set of active grains is bounded in size by the total number of variables:

Lemma 7.1. Let w be a run and let G be a set of minimal grains in w. For a prefix π of w, we have $|\mathsf{Active}_{\pi,G}| \leq |\mathcal{X}|$.

This is implied by the following crucial observation. Assume two minimal grains g and g' are active at given prefix π of w. The minimality g (respectively g') implies that there is a variable x (respectively x') that is written in π and has a corresponding read in the remainder of w. The variables x and x' cannot coincide. Therefore one needs a distinct variable that only belongs to one of the many active grains at any given point, which puts a bound of $|\mathcal{X}|$ on the maximum number of grains that can be active at any given time.

Ideally, we want all grains to be minimal, since this resolves the problem of having unboundedly many active grains. Let us argue why this can be achieved without any compromises to the result of checking causal concurrency. Consider Fig. 6(b) which depicts the same run as in Fig. 6(a) but this time with a different set of grains which are all minimal. Observe that the set of minimal grains in Fig. 6(b) witness more causal concurrency than the set of non-minimal grains in Fig. 6(a). In general, one can argue that for any set of (non-minimal) grains that witnesses the causal concurrency of two events e and f, there exists a set of minimal events that does the same. We give a constructive argument for this claim.

One can argue that any grain g is the concatenation of a sequence of minimal grains. Recall Definition 7.1. For any ρ and σ that witness non-minimality of $g = \rho\sigma$ according to Definition 7.1, add a split point between ρ and σ. Let $g = g_1 \ldots g_n$ where g_i's are precisely marked by these split points. By definition, g_i's are all minimal. Define $\mathsf{split}(G) = \{g_1, \ldots, g_n\}$.

Given a valid set of grains G, we use $\mathsf{split}(G) = \bigcup_{g \in G} \mathsf{split}(g)$ to denote the set of grains obtained by splitting individual grains in G. We need to argue that splitting all grains into minimal grains would result in declaring at least as many pairs of events causally concurrent as before.

Given a commutativity relation $\mathbb{I}_G$ on G, define $\mathsf{split}(\mathbb{I}_G) \subseteq \mathsf{split}(G) \times \mathsf{split}(G)$ as

$$\mathsf{split}(\mathbb{I}_G) = \{(g'_1, g'_2) \mid \exists g_1, g_2 \in G, (g_1, g_2) \in \mathbb{I}_G \text{ and } g'_1 \in \mathsf{split}(g_1), g'_2 \in \mathsf{split}(g_2)\}$$

One can prove that $\mathsf{split}(\mathbb{I}_G)$ is sound. This in turn implies that splitting grains does not add spurious paths in the new grain graph, which did not exist in the original one.

Lemma 7.2. Let w be a run, G be a valid set of scattered grains, and $\mathbb{I}_G$ be a sound independence relation (Definition 4.6). $\mathsf{split}(\mathbb{I}_G)$ is sound, and for every pair of events (e_1, e_2), if e_1 and e_2 are grain graph concurrent under G (using commutativity relation $\mathbb{I}_G$), then they are grain graph concurrent under the grains $\mathsf{split}(G)$ (using commutativity relation $\mathsf{split}(\mathbb{I}_G)$).

Therefore, when checking scattered grain concurrency, it is safe to ignore all sets of grains that include any non-minimal grains, since the same causal concurrency verdicts can be declared by other sets of minimal grains.

7.2 Tracking Retroactive Paths

Let us address our second challenge, that is how to keep track of *retroactive* paths in the grain graph. We fix the set of minimal grains. Since grains containing only a single event are by definition minimal, we can assume, without loss of generality, that every event is part of a grain; standalone events are grains of size one. Like Section 5, the causal concurrency question between events e_1 and e_2 is posed as the causal concurrency between the grains $g^{\diamond_1}$ and $g^{\diamond_2}$ containing these events.

It is clear that to design a constant space monitor, we cannot store the entire grain graph in the memory of the monitor. The idea is to forget all grains that are no longer active and *summarize* their effect instead. Under the assumption that all grains are minimal, the number of active grains are bounded by $|\mathcal{X}|$. This guarantees that the monitor keeps track of constantly many grains. We can annotate events with a finite set of grain identifiers $\{1, 2, \ldots, |\mathcal{X}|\}$. Identifiers are reused for grains that do not overlap.

The monitor maintains a graph where the nodes are precisely the set of active grains. We call this graph the *summarized grain graph.* Recall that the key information a monitor wants from a grain graph is whether two grains are connected by a directed path in the grain graph. Paths from the grain graph are represented as edges in the summarized grain graph. In particular, the edges in the summarized grain graph capture paths in the original grain graph whose intermediate grains have become inactive.

More formally, let π be a prefix of run w and let G be a valid set of grains. For grains (active or otherwise) g, g' that overlap with π, use the notation $g \rightsquigarrow_{\pi,G} g'$ to say that there is a path *through* inactive grains from g to g', i.e., there are grains $g_1, g_2 \ldots, g_k \in G$ (with $k > 1$, $g = g_1$ and $g' = g_k$), such that $g_2, \ldots, g_{k-1} \subseteq \pi$ are inactive (i.e., have started and completed) in π, and (g_i, g_{i+1}) is an edge of the original grain graph, for each $1 \leq i \leq k-1$. In essence, a path in the grain graph can be split into successive paths (through inactive grains) between the active grains.

The summarized grain graph is maintained by the monitor for answering a specific causal concurrency query between two grains $g^{\diamond_1}$ and $g^{\diamond_2}$. Formally:

Definition 7.2 (Summarized Grain Graph)**.** Let w be a run, G be a valid set of scattered grains and let π be a prefix of w. The summarized conflict graph of π is $\mathcal{SG}_{\pi,G} = (V_\pi, E_\pi)$, where

(1) $V_\pi = \text{Active}_{\pi,G} \cup \{g^{\diamond_1}, g^{\diamond_2}\}$ is the set containing the active grains at the end of π as well as the focal grains,

(2) $(g, g') \in E_\pi$ if $g \rightsquigarrow_{\pi,G} g'$,

Summarized grain graphs sufficiently capture the reachability information between the focal grains:

Proposition 7.1. Let w be a run and let G be a valid set of scattered grains in w. There is a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{G}_{w,G}$ iff there is a prefix π of w such that there is a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{SG}_{\pi,G}$.

It remains to argue that a constant-space streaming algorithm (that reads an input run in one pass from left to right) can successfully construct the summarized grain graph for the run. Intuitively, every time an active grain g is about to end, and as such become inactive and disappear, its summary is added to all its predecessors g'; that is, all nodes in the summarized grain graph that have a incoming edge to g. This way, a future active grain g'', that would be a successor of g in the grain graph, will now become the successor of all g''s, since the summary information stored in them will trigger the formation of an edge from them to g''. We make this idea formal in the detailed description of the monitors in the next sections.

For example, recall Fig. 6(c) and assume we want to query causal concurrency between g_1 and g_2. As such, both g_1 and g_2 have dedicated nodes in the summarized grain graph independent of their activeness status. The dashed edge appears in the summarized grain graph in place of the path from g_1 to g_4, while g_4 is active. Once g_4 is finished, as a predecessor of g_4, g_1 would remember the $\mathsf{w}(y)$ access so that it can add an edge to g_3 when its $\mathsf{r}(y)$ appears.

7.3 Monitoring for a Fixed set of Minimal Scattered Grains

The description of the monitor that checks grain graph concurrency (under a given set of minimal grains) is now straightforward as it essentially tracks and updates the summarized grain graph after each prefix of the run that is seen. We assume that the run contains symbols $\triangleright$ and $\triangleleft$ denoting start and end of grains. For ease of presentation, let us assume that the run labels the *focal grains* $g^{\diamond_1}$ and $g^{\diamond_2}$ with fresh identifiers identifiers $\diamond_1$ and $\diamond_2$. Thus, the set of grain identifiers is thus $\mathsf{gIDs} = \{1, 2, \dots, |\mathcal{X}|\} \uplus \{\diamond_1, \diamond_2\}$.

The commutativity status of a grain depends on the *pending* variables of a grain, i.e., for each grain g, we identify the set of variables x such that x is read at some event $e \in g$ but not written to in g (i.e., $\mathsf{rf}_w(e) \notin g$), or x is written at some event $e \in g$ to but is read outside of g i.e., $\exists e', \mathsf{rf}_w(e') = e \land e' \notin g$). While this information can be guessed non-deterministically and checked later on, for simplifying the presentation of our monitor, we will also assume that the alphabet encodes this information as part of the $\triangleright$ marker of grain. Thus, the alphabet of runs can be assumed to be

$$\widehat{\Sigma} = \mathsf{gIDs} \times (\Sigma \uplus \{\triangleright\} \times \mathcal{P}(\mathcal{X}) \uplus \{\triangleleft\}).$$

For the purpose of this section, we will assume that the runs we consider are valid strings over the alphabet $\widehat{\Sigma}$ that are minimal, and use the language $L_{\mathsf{VMG}} = \{w \in \widehat{\Sigma}^* \mid w$ represents a $\underline{\mathbf{v}}$alid $\underline{\mathbf{m}}$inimal $\underline{\mathbf{g}}$rain annotation with focal grains$\}$ to denote the set of all such annotated runs. Below, we present the annotated version of the run in Fig. 3(a) with all grains g_1, g_2, g_3, g_4, assuming the two focal grains are g_2 and g_4 is presented below.

$$\begin{array}{rcl} w & = & (\triangleright, \varnothing)^{\diamond_1} \langle T_2, \mathsf{w}(x)\rangle^{\diamond_1} (\triangleright, \varnothing)^1 \langle T_1, \mathsf{w}(z)\rangle^1 \langle T_2, \mathsf{r}(z)\rangle^1 \triangleleft^1 \; (\triangleright, \varnothing)^1 \langle T_3, \mathsf{w}(z)\rangle^1 \langle T_3, \mathsf{r}(z)\rangle^1 \triangleleft^1 \\ & & \langle T_3, \mathsf{r}(x)\rangle^{\diamond_1} \triangleleft^{\diamond_1} \; (\triangleright, \varnothing)^{\diamond_2} \langle T_1, \mathsf{w}(x)\rangle^{\diamond_2} \langle T_1, \mathsf{r}(x)\rangle^{\diamond_2} \triangleleft^{\diamond_2} \end{array}$$

In the above, we use the notation a^i as shorthand for $(i, a) \in \widehat{\Sigma}$. Observe that the (unique) grains with grain identifier $\diamond_1$ (namely grain g_2) overlaps with both the grains with identifier 1 (i.e., grains g_1 and g_3). Also observe that L_{VMG} is a regular language. Thus, a monitor that works correctly for runs in this language also works for non-annotated runs because the language of a constant space monitor is regular and thus closed under projection.

As discussed in Section 5, it suffices to consider the largest sound commutativity relation on a given set of grains. In fact, the largest commutativity relation also has a succinct representation — the dependence between any two scattered grains in this relation can be checked only using the *signatures* $g_1 = \langle E_1, V_1\rangle$ and $g_2 = \langle E_2, V_2\rangle$ of these grains:

$$\mathit{depend}(g_1, g_2) \iff \exists e_1 \in E_1, e_2 \in E_2 \cdot \left(\begin{array}{rl} & \mathsf{thr}(e_1) = \mathsf{thr}(e_2) \\ \lor & \mathsf{var}(e_1) = \mathsf{var}(e_2) \notin V_1 \cap V_2 \land \mathsf{w} \in \{\mathsf{op}(e_1), \mathsf{op}(e_2)\} \end{array} \right)$$

Recall that the signatures are bounded sized, and so is their dependence relationship. Thus, in order to maintain the summarized graph inductively, states stores additional information to correctly infer commutativity with other grains, including those that have finished.

Our monitor essentially tracks grain signatures of interest to infer edges between relevant grains. Let π be a prefix of w and let g be some grain that is active in π. Then, we use the notation $\mathsf{C}_\pi(g) = \{w[e] \mid e \in g \cap [1..|\pi|]\}$ to denote the $\underline{\mathsf{C}}$*ontents* of G in π. Likewise, we denote the set of $\underline{\mathsf{P}}$*ending* variables of g by $\mathsf{P}(g) = \{x \in \mathcal{X} \mid \exists e \in g, \mathsf{var}(e) = x, \neg\mathsf{complete}(g, x)\}$. For each active grain g that we track in our monitor, we will also maintain summarized information about those grains that are no longer active but can be reached from g, to accurately infer edges in the summarized graph (alternatively *retroactive* paths in the grain graph). The first such information is the $\underline{\mathsf{S}}$*ummarized* $\underline{\mathsf{C}}$*ontents* of the inactive but reachable grains: $\mathsf{SC}_\pi(g) = \bigcup\{\mathsf{C}_\pi(g') \mid \exists g' \subseteq \pi, g \rightsquigarrow_{\pi,G} g'\}$. Likewise, we use $\mathsf{SP}_\pi(g) = \bigcup\{\mathsf{P}_\pi(g') \mid \exists g' \subseteq \pi, g \rightsquigarrow_{\pi,G} g'\}$ to denote the $\underline{\mathsf{S}}$*ummarized* $\underline{\mathsf{P}}$*ending* variables that g must keep track of.

We are now ready to formally describe our **G̲**rain **G̲**raph concurrency monitor $\mathcal{A}_{\mathsf{GG}} = (Q_{\mathsf{GG}}, q_0, \delta_{\mathsf{GG}}, F_{\mathsf{GG}})$. The states Q_{GG} of $\mathcal{A}_{\mathsf{GG}}$ are tuples of the form $\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$ where

- $V \subseteq \mathsf{gIDs}$ represents the set of active and focal grains of the summarized graph.
- $E \subseteq V \times V$ represents the edges of the summarized graph.
- $\mathsf{C} : V \to \mathcal{P}(\Sigma)$ maintains the *contents* of active grains. For each grain g tracked as a vertex u, we will have $\mathsf{C}(u) = \mathsf{C}_\pi(g)$ after having processed the prefix π.
- $\mathsf{P} : V \to \mathcal{P}(\mathcal{X})$ to track the set $\mathsf{P}_\pi(g)$ for each grain g (at the end of prefix π) tracked as some vertex in V.
- $\mathsf{SC} : V \to \mathcal{P}(\Sigma)$ is such that $\mathsf{SC}_\pi(u)$ tracks the set $\mathsf{SC}_\pi(g)$ at the end of prefix π, where u represents the grain g.
- $\mathsf{SP} : V \to \mathcal{P}(\mathcal{X})$ to track the set $\mathsf{SP}_\pi(g)$ for each grain g (at te end of prefix π) tracked as some vertex in V.

The start state of the monitor is $q_0 = \langle \varnothing, \varnothing, \lambda i \cdot \varnothing, \lambda i \cdot \varnothing, \lambda i \cdot \varnothing, \lambda i \cdot \varnothing\rangle$. All states $\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$ in which $\diamond_1, \diamond_2 \in V$ and further $\diamond_2$ is reachable from $\diamond_1$ via a path using edges in E are marked rejecting, and others are accepting (i.e., belong to F_{GG}). The transitions of the monitor are described in Fig. 7. When we see an event $e = (i, (\rhd, Y))$ that demarcates the beginning of a grain with identifier i, we also know upfront the set of pending variables in the grain. At this point, we create a new node labeled i in the graph and also track this set Y. When we see the end of a grain (marked $\lhd$) with identifier i, then we *garbage collect* the node i from the graph. As part of this garbage collection, we add an edge from the immediate predecessors of i to its immediate successors; this is captured by the operation $mrg(\cdot, \cdot)$. Further, the maps C and P reset the entry corresponding to i, and SC and SP entries of the predecessors of i are updated to include $\mathsf{C}(i)$, $\mathsf{SC}(i)$, $\mathsf{P}(i)$ and $\mathsf{SP}(i)$; this is captured using $mrgSm(\cdot, \cdot, \cdot, \cdot)$. When we see a read or a write event $a = \langle T, o, x\rangle$ in grain corresponding to the node i, we add edges from all nodes j to i such that j conflicts with a, i.e., either the contents or the summary of j contains a letter that conflicts with a, or one of j or an inactive grain reachable from j has x as a pending variable.

One can prove that the monitor in Fig. 7 correctly maintains $\rightsquigarrow_{\pi,G}$ between each pair of active/focal grains after every prefix π of w, and as such, it can correctly decide whether the two focal grains are scattered grain concurrent or not:

State	Event	State Update
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, (\triangleright, Y))$	$\langle V \uplus \{i\}, E, \mathsf{C}, \mathsf{P}[i \mapsto Y], \mathsf{SC}, \mathsf{SP}\rangle$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, \triangleleft), i \in \{\diamond_1, \diamond_2\}$	$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, \triangleleft), i \notin \{\diamond_1, \diamond_2\}$	$\langle V', E', \mathsf{C}', \mathsf{P}', \mathsf{SC}', \mathsf{SP}'\rangle$, where, $V' = V - \{i\}, E' = \mathit{mrg}(E, i)$ $\mathsf{C}' = \mathsf{C}[i \mapsto \varnothing], \mathsf{P}' = \mathsf{P}[i \mapsto \varnothing]$, $\mathsf{SC}' = \mathit{mrgSm}(\mathsf{SC}, \mathsf{C}, E, i)$, $\mathsf{SP}' = \mathit{mrgSm}(\mathsf{SP}, \mathsf{P}, E, i)$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, a), a \in \Sigma$	$\langle V, E', \mathsf{C} \cup \{a\}, \mathsf{SC}, \mathsf{SP}\rangle$, where, $E' = E \cup \{(j, i) \mid j \neq i$ and $\mathit{Dep}(\mathsf{C}(j) \cup \mathsf{SC}(j),\ a,\ \mathsf{P}(j) \cup \mathsf{SP}(j) \cup \mathsf{P}(i))\ \}$

$$\mathit{mrg}(E, i) = (E - \{(i, j), (j, i) \mid j \neq i\}) \cup \{(j, k) \mid (j, i) \in E, (i, k) \in E\}$$

$$\mathit{mrgSm}(SM, M, E, i) = \lambda j \cdot \begin{cases} \varnothing & \text{if } j = i \\ SM(j) \cup M(i) \cup SM(i) & \text{if } j \neq i, (j, i) \in E \\ SM(j) & \text{owise} \end{cases}$$

$$\mathit{Dep}(S, a, Z) \iff \exists b \in S \cdot \Big(\mathsf{thr}(a) = \mathsf{thr}(b) \vee \big(\mathsf{var}(a) = \mathsf{var}(b) \in Z \wedge \mathsf{w} \in \{\mathsf{op}(a), \mathsf{op}(b)\}\big)\Big)$$

Fig. 7. Grain Graph Concurrency Monitor for Annotated Runs $\mathcal{A}_{\mathrm{GG}}$: The monitor rejects if it is in a state $(V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP})$ such that $(\diamond_1, \diamond_2) \in E^*$ at the end of the run, and accepts otherwise.

Theorem 7.1. Given a run $w \in L_{\mathrm{VMG}}$ annotated with grains G, w is accepted by $\mathcal{A}_{\mathrm{GG}}$ iff the two focal grains are grain graph concurrent under G.

7.4 Monitoring Scattered Grain Concurrency

Similar to the case for grain concurrency, the monitor for scattered grain concurrency (Definition 6.4) can non-deterministically guess the choice of scattered grains and check, using the monitor in Fig. 7, if any of these guesses is valid and declares the two focal grains to causally concurrent.

Theorem 7.2. There exists a monitor $\mathcal{A}$ that uses $2^{O(|\mathcal{X}| \cdot |\Sigma|)}$ space and accepts the word $w \in \Sigma^* \diamond_1 \Sigma^+ \diamond_2 \Sigma^+$ iff events that appear immediately after $\diamond_1$ and $\diamond_2$ are scattered grain concurrent in w. Consequently, scattered grain concurrency can be checked in constant space.

The proof of the above theorem relies on Theorem 7.1 and the observations that given a run $w \in \Sigma$ (with $\diamond_1$ and $\diamond_2$), we can guess the grains on w in constant space, validate whether the guessed grains are minimal and valid, and finally if the resulting annotated run is accepted by $\mathcal{A}_{\mathrm{GG}}$. The number of states in $\mathcal{A}_{\mathrm{GG}}$ is $2^{|\mathcal{X}|^2} \cdot (2^{|\Sigma|})^{|\mathcal{X}|} \cdot (2^{|\mathcal{X}|})^{|\mathcal{X}|} \cdot (2^{|\Sigma|})^{|\mathcal{X}|} \in 2^{O(|\mathcal{X}| \cdot |\Sigma|)}$ can can be obtained by counting the different ways to construct graphs on $|\mathcal{X}|$ vertices and deciding on the contents, pending variables and summaries and summarized pending variables for these vertices. Since the scattered grain concurrency monitors essentially adds non-determinism on top of this DFA, its deterministic version has exponentially many states, and each state thus has size $2^{O(|\mathcal{X}| \cdot |\Sigma|)}$.

8 RELATED WORK

There is little work in the literature in the general area of generalizing commutativity-based analysis of concurrent programs. On the theoretical side, some generalizations of Mazurkiewicz traces have been studied before [Bauget and Gastin 1995; Kleijn et al. 1998; Maarand and Uustalu 2019; Sassone et al. 1993]. The focus has mostly been on incorporating the concept of *contextual* commutativity, that is, when two events can be considered commutative in some contexts but not in all. This is motivated, among other things, by send/receive or producer/consumer models in distributed systems where send and receive actions commute in contexts with non-empty message buffers.

On the practical side, similar ideas were used in [Desai et al. 2014; Farzan et al. 2022; Farzan and Vandikas 2020; Genest et al. 2007] to reason about equivalence classes of concurrent and distributed programs under contextual commutativity. There is a close connection between this notion of contextual commutativity and the concept of *conditional independence* in partial order reduction [Godefroid and Pirottin 1993; Katz and Peled 1992] which is used as a weakening of the independence (commutativity) relation by making it parametric on the current *state* to increase the potential for reduction.

This work points out that, in general, the coarsest notion of equivalence may not yield monitoring-style algorithms for analyzing runs of concurrent programs, and puts forward an alternative notion of equivalence, coarser than trace equivalence, that can be efficiently used in monitoring causal concurrency. Below, we briefly survey some application domains relevant to programming languages research where our proposed equivalences can have an immediate positive impact.

Concurrency Bug Prediction. Dynamic analysis techniques for detecting concurrency bugs such as data races [Flanagan and Freund 2009], deadlocks [Samak and Ramanathan 2014] and atomicity violations [Farzan and Madhusudan 2008; Flanagan et al. 2008; Sorrentino et al. 2010] suffer from poor coverage since their bug detection capability is determined by the precise thread scheduling observed during testing. Predictive techniques [Said et al. 2011; Sen et al. 2005; Wang et al. 2009] such as those for detecting data races [Huang et al. 2014; Kini et al. 2017; Mathur et al. 2021; Pavlogiannis 2019; Roemer et al. 2018; Smaragdakis et al. 2012] or deadlocks [Kalhauge and Palsberg 2018; Tunç et al. 2023] enhance coverage by exploring equivalent runs that might yield a bug. The core algorithmic problems involved in such approaches are akin to checking causal concurrency. Coarser yet tractable equivalence relations can yield better analysis techniques with more accurate predictions. Recent work [Kulkarni et al. 2021] explores hardness results for data race prediction, including Mazurkiewicz-style reasoning, when the alphabet is not assumed to be of constant size.

Dynamic Partial Order Reduction for Stateless Model Checking. There has been a rising interest in developing dynamic partial order based [Flanagan and Godefroid 2005] stateless model checking techniques [Godefroid 1997] that are *optimal*, in that they explore as few program runs as possible from the underlying program. Coming up with increasingly coarser equivalences is the go-to approach for this. The notion of reads-from equivalence we study has received a lot of attention in this context [Abdulla et al. 2019; Chalupa et al. 2017; Kokologiannakis et al. 2022, 2019]. Recent works also consider an even coarser reads-value-from [Agarwal et al. 2021; Chatterjee et al. 2019] equivalence. Events encode variable values and two runs are equivalent if every read observes the same value across the two runs. The problem of verifying sequential consistency, which is one of the key algorithmic questions underlying such approaches, was proved to be intractable in general by Gibbons and Korach [Gibbons and Korach 1997].

9 CONCLUSION AND FUTURE WORK

This paper demonstrates that reads-from equivalence, the most relaxed sound notion of equivalence on concurrent program runs, does not share the nice algorithmic properties of (Mazurkiewicz) trace equivalence. This poses the following research questions: "Are there other notions of equivalence, which remain sound, relax trace equivalence, and yet maintain its key desirable algorithmic properties? And, what design principles bring about algorithmic simplicity?". We propose two new notions of equivalence in this paper under which causal concurrency can be decided by a streaming algorithm in constant space. Equivalence based on contiguous grains shares the characteristic with trace equivalence, in that it is definable purely in terms of commutativity. Remarkably, while this characteristic is lost with scattered grains, the algorithmic simplicity remains. The point of commonality between the two notions is that each individual event can only *move* in one role: either individually, or as part of a grain. As we demonstrate in Theorem 4.1, algorithmic hardness kicks in when this rule is broken in the simplest of syntactic settings. It would be interesting to investigate whether one can further relax the notion of *grain equivalence* by breaking this barrier.

As we note in Section 7, the straightforward determinization of the (scattered) grain concurrency monitor can result in an exponential blowup on the number of threads and shared variables. We treat these parameters as constants, in a manner similar to trace theory's reliance on a finite (constant-sized) alphabet of actions. There seems to be a tradeoff between how coarse the equivalence relation is and how efficiently causal concurrency can be monitored. The research question of how to devise a practical monitor when the number of threads or variables is not very small remains an interesting direction for future research.

Finally, this paper studies coarser equivalences in the context of a *causal concurrency query*. In several application domains, the standard oracle for causal concurrency based on trace equivalence can be replaced with the (scattered) grain concurrency from this paper. Yet, there remain other application domains in which trace equivalence is used in completely different ways: for example, proof simplification [Farzan 2023] by verifying a commutativity-based reduction of a concurrent program. The notion of soundness used in this paper suffices when the object of study is a single program run. In proof simplification, however, a set of program runs must be considered together, and, as we argued, different grain commutativity relations may be sound in different program runs. Moreover, the alphabet of actions for proof simplification typically includes atomic program statements rather than shared variable reads and writes. As such, the theoretical results presented in this paper do not immediately offer a solution in such domains. It will be interesting to explore how similar coarse equivalences, based on commutativity of words (rather than symbols), can be designed to be exploited for proof simplification.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (oct 2019), 29 pages. https://doi.org/10.1145/3360576

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I.* Springer-Verlag, Berlin, Heidelberg, 341–366. https://doi.org/10.1007/978-3-030-81685-8_16

Serge Bauget and Paul Gastin. 1995. On congruences and partial orders. In *Mathematical Foundations of Computer Science 1995*, Jiří Wiedermann and Petr Hájek (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 434–443.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (dec 2017), 30 pages. https://doi.org/10.1145/3158119

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 124:1–124:29. https://doi.org/10.1145/3360550

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014.* 709–725.

Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces.* World Scientific.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

Azadeh Farzan. 2023. Commutativity in Automated Verification. In *LICS*. 1–7. https://doi.org/10.1109/LICS56636.2023.10175734

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound sequentialization for concurrent program verification. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 506–521. https://doi.org/10.1145/3519939.3523727

Azadeh Farzan and P. Madhusudan. 2006. Causal Atomicity. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–328.

Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–65.

Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. 2009. Meta-analysis for Atomicity Violations Under Nested Locking. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 248–262. https://doi.org/10.1007/978-3-642-02658-4_21

Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

Azadeh Farzan and Anthony Vandikas. 2020. Reductions for safety proofs. *Proc. ACM Program. Lang.* 4, POPL (2020), 13:1–13:28. https://doi.org/10.1145/3371081

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

Blaise Genest, Dietrich Kuske, and Anca Muscholl. 2007. On Communicating Automata with Bounded Channels. *Fundam. Inform.* 80, 1-3 (2007), 147–167.

Phillip B. Gibbons and Ephraim Korach. 1994. On Testing Cache-Coherent Shared Memories. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures* (Cape May, New Jersey, USA) *(SPAA '94)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/181014.181328

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614 arXiv:https://doi.org/10.1137/S0097539794279614

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid and Didier Pirottin. 1993. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings.* 438–449.

Michel Hack. 1976. *Petri net language.* Massachusetts Institute of Technology.

Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Russell Impagliazzo and Ramamohan Paturi. 2001. On the complexity of k-SAT. *J. Comput. System Sci.* 62, 2 (2001), 367–375.

Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. 1999. Toward Integration of Data Race Detection in DSM Systems. *J. Parallel Distrib. Comput.* 59, 2 (Nov. 1999), 180–203. https://doi.org/10.1006/jpdc.1999.1574

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

Shmuel Katz and Doron A. Peled. 1992. Defining Conditional Independence Using Collapses. *Theor. Comput. Sci.* 101, 2 (1992), 337–359.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

H.C.M. Kleijn, R. Morin, and B. Rozoy. 1998. Event Structures for Local Traces. *Electronic Notes in Theoretical Computer Science* 16, 2 (1998), 98–113. https://doi.org/10.1016/S1571-0661(04)00120-3 EXPRESS '98, Fifth International Workshop on Expressiveness in Concurrency (Satellite Workshop of CONCUR '98).

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 6, POPL, Article 49 (jan 2022), 28 pages. https://doi.org/10.1145/3498711

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory (CONCUR 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:23. https://doi.org/10.4230/LIPIcs.CONCUR.2021.16

Hendrik Maarand and Tarmo Uustalu. 2019. Certified normalization of generalized traces. *Innovations in Systems and Software Engineering* (2019).

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434317

Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time Using Vector Clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 183–199. https://doi.org/10.1145/3373376.3378475

A Mazurkiewicz. 1987. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag New York, Inc., 279–324.

Robin Milner. 1980. *A calculus of communicating systems*. Springer.

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

Grigore Roşu and Mahesh Viswanathan. 2003. Testing Extended Regular Language Membership Incrementally by Rewriting. In *Rewriting Techniques and Applications*, Robert Nieuwenhuis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 499–514.

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327. http://dl.acm.org/citation.cfm?id=1986308.1986334

Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2555243.2555262

Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. 1993. Deterministic behavioural models for concurrency. In *Mathematical Foundations of Computer Science 1993*, Andrzej M. Borzyszkowski and Stefan Sokołowski (Eds.).

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Formal Methods for Open Object-Based Distributed Systems*, Martin Steffen and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–226.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1733–1758. https://doi.org/10.1145/3591291

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. https://doi.org/10.1145/3591291

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *Proceedings of the 2Nd World Congress on Formal Methods* (Eindhoven, The Netherlands) *(FM '09)*. Springer-Verlag, Berlin, Heidelberg, 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science* 348, 2-3 (2005), 357–365.

Glynn Winskel. 1987. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–392.

A PROOFS FROM SECTION 3

In the following we will use the notation $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, i, j)$ to denote that the events i and j are causally ordered in w under $\equiv_{\mathsf{rf}}$

A.1 Proof of Theorem 3.1 and Theorem 3.2

We first state some properties of reads-from equivalence. We first define additional notation. For an execution $w \in \Sigma^*$, we define the relation $<^{w}_{\equiv_{\mathsf{rf}}}$ defined as follows:

$$<^{w}_{\equiv_{\mathsf{rf}}} = \{(i, j) \mid i, j \in [1..|w|], \mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, i, j)\}$$

Proposition A.1. Let $w \in \Sigma^*$ be an execution. The relation $<^{w}_{\equiv_{\mathsf{rf}}}$ defined above satisfies the following properties.

(Partial order). $<^{w}_{\equiv_{\mathsf{rf}}}$ is a partial order. That is, $<^{w}_{\equiv_{\mathsf{rf}}}$ is irreflexive and transitive.

(Intra-thread order). $<^{w}_{\equiv_{\mathsf{rf}}}$ orders events of w in the same thread. That is, for every $i < j \in [1..|w|]$, if $\mathsf{thr}(w[i]) = \mathsf{thr}(w[j])$, then $i <^{w}_{\equiv_{\mathsf{rf}}} j$.

(Reads-from). $<^{w}_{\equiv_{\mathsf{rf}}}$ w orders events if there is a reads-from dependency between them. That is, for every $i < j \in [1..|w|]$, if $\mathsf{rf}_w(j) = i$, then $i <^{w}_{\equiv_{\mathsf{rf}}} j$.

(Implied orders). Let $i, j, k \in [1..|w|]$ be distinct indices such that such that $\mathsf{var}(i) = \mathsf{var}(j) = \mathsf{var}(k)$, $\mathsf{op}(i) = \mathsf{op}(k) = \mathtt{w}$ and $\mathsf{op}(j) = \mathtt{r}$ and $\mathsf{rf}_w(j) = i$.

- If $i <^{w}_{\equiv_{\mathsf{rf}}} k$, then $j <^{w}_{\equiv_{\mathsf{rf}}} k$.
- If $k <^{w}_{\equiv_{\mathsf{rf}}} j$, then $k <^{w}_{\equiv_{\mathsf{rf}}} i$.

Proof. Follows from the definition of $<^{w}_{\equiv_{\mathsf{rf}}}$. □

The proof of Theorem 3.1 relies on a reduction from the following language, parametrized by $n \in \mathbb{N}_{>0}$:

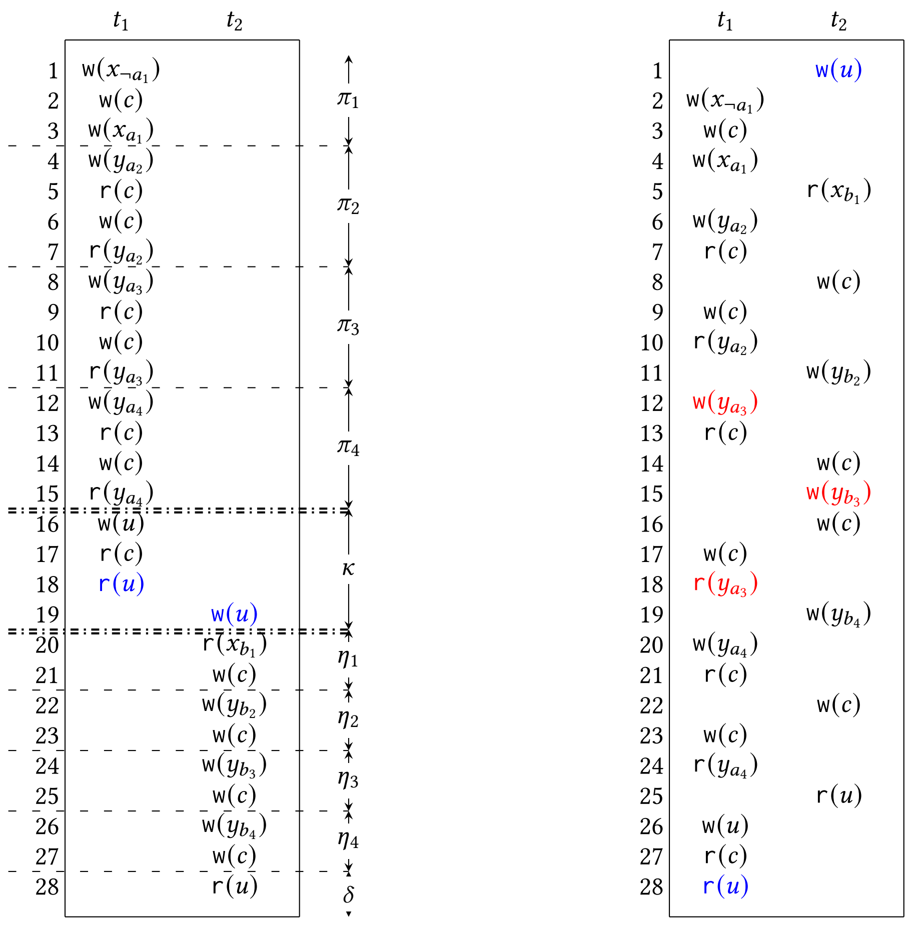
$$L^{=}_{n} = \{\overline{a}\#\overline{b} \mid \overline{a}, \overline{b} \in \{0,1\}^n \text{ and } \overline{a} = \overline{b}\}$$

We first observe that there is a linear space lowerbound for the problem of recognition of this language.

Lemma A.1. Any streaming algorithm that recognizes $L^{=}_{n}$ uses $\Omega(n)$ space.

Proof. Assume towards contradiction otherwise, i.e., there is a streaming algorithm that uses $o(n)$ space. Hence the state space of the algorithm is $o(2^n)$. Then, there exist two distinct n-bit strings $a \neq a'$, such that the streaming algorithm is in the same state after parsing $\overline{a}$ and $\overline{a'}$. Hence, for any n-bit string $\overline{b}$, the algorithm gives the same answer on inputs $\overline{a}\#\overline{b}$ and $\overline{a'}\#\overline{b}$. Since the algorithm is correct, it reports that $a\#a$ belongs to L_n. But then the algorithm reports that $\overline{a'}\#\overline{a}$ also belongs to L_n, a contradiction. The desired result follows. □

Proof of Theorem 3.1. We will now show that there is a linear-space lower bound for Problem 3.2 by showing a reduction from $\{0,1\}^n\#\{0,1\}^n$ to Σ^* in one pass streaming fashion using constant space. The constructed string will be such that there are unique indices i and j whose labels correspond to the symbols c an d in the causal concurrency problem.



(a) Run constructed for $n = 4$. (b) Reordering for $a = 11\underline{0}0$ and $b = 11\underline{1}0$.

Fig. 8. Reduction from $L_4^= = \{a_1a_2a_3a_4\#b_1b_2b_3b_4 \mid \forall i, a_i = b_i \in \{0,1\}\}$. Trace construction (on left) and example of equivalent trace when $\overline{a} \neq \overline{b}$.

Consider the language L_n for some n. We describe a transducer $\mathcal{M}_n$ such that, on input a string $v = \overline{a}\#\overline{b}$, the output $\mathcal{M}_n(v)$ is an execution w with 2 threads t_1 and t_2, $O(n)$ events and 6 variables such that w is of the form

$$w = \pi\kappa\eta\delta$$

We fix the letters c and d for which we want to check for causl concurrency to be $c = \langle t_1, \mathsf{r}, u\rangle$ and $d = \langle t_2, \mathsf{w}, u\rangle$, where $u \in \mathcal{X}$. The fragments π and η do not contain any access to the variable u. The fragments κ and δ contain accesses to u. Fig. 8a describes our construction for $n = 4$.

Our reduction ensures the following:

(1) If $v \in L_n^=$, then there are $\theta_1, \theta_2 \in [1..|w|]$ such that $\mathsf{thr}(w[\theta_1]) = t_1$, $\mathsf{thr}(w[\theta_2]) = t_2$, $\mathsf{var}(w[\theta_1]) = \mathsf{var}(w[\theta_2]) = u$ and $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \theta_1, \theta_2)$.

(2) If $v \notin L_n^=$, then there are no such θ_1 and θ_2.

Moreover, $\mathcal{M}_n$ will use $O(1)$ working space.

Let us now describe the construction of w. At a high level, the sub-execution π corresponds to the prefix $\overline{a} = a_1a_2 \cdots a_n$ of v and the sub-execution η corresponds to the suffix $\overline{b} = b_1b_2 \cdots b_n$ of v. Both these sub-executions have $O(n)$ events. The sub-executions κ and δ have $O(1)$ (respectively 4 and 1) events. The sequences of events in the two sub-executions π and η ensure that there is a 'chain' of conditional dependencies, that are incrementally met until the point when the two substrings $\overline{a}$ and $\overline{b}$ match. If $\overline{a} = \overline{b}$ (i.e., full match), then the dependency chain further ensures that there are events, at indices θ_1 and θ_2, both in κ (accessing the variable u in threads t_3 and t_2 respectively) such that $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \theta_1, \theta_2)$.

The execution fragment π is of the form

$$\pi = \pi_1\pi_2 \cdots \pi_n.$$

Here, the fragment π_i corresponds to the i^{th} bit a_i of $\overline{a}$, and only contains events performed by the thread t_1. The fragment η is of the form

$$\eta = \eta_1\eta_2 \cdots \eta_n$$

Here, η_i corresponds to b_i, the i^{th} bit in $\overline{b}$ and only contains events of thread t_2. The variables used in the construction are $\{c, x_0, x_1, y_0, y_1, u\}$, where u is the special variable whose events will be ordered or not based on the input. In the rest of the construction, we will use the notation $\langle t, o, v, \gamma\rangle$ to denote the unique index α occuring within the fragment γ for which $w[\alpha] = \langle t, o, v\rangle$. We next describe each of the fragments $\pi_1, \ldots, \pi_n$, $\eta_1, \ldots, \eta_n$ and the fragments κ and δ.

Fragment π_1. The first fragment π_1 of π is as follows:

$$\pi_1 = \langle t_1, \mathsf{w}, x_{\neg a_1}\rangle\langle t_1, \mathsf{w}, c\rangle\langle t_1, \mathsf{w}, x_{a_1}\rangle$$

That is, the last (resp. first) event writes to x_0 (resp. x_1) if $a_1 = 0$, otherwise it writes to the variable x_1 (resp. x_0).

Fragment η_1. The first fragment η_1 of η is as follows:

$$\eta_1 = \langle t_2, \mathsf{r}, x_{b_1}\rangle\langle t_2, \mathsf{w}, c\rangle$$

In the entire construction, the variables x_0 and x_1 are being written-to only in fragment π_1, and (potentially) read only in η_1. This means that, either $\mathsf{rf}_w(\langle t_2, \mathsf{r}, x_{b_1}, \eta_1\rangle) = \langle t_1, \mathsf{r}, x_{a_1}, \pi_1\rangle$ (if $a_1 = b_1$) or $\mathsf{rf}_w(\langle t_2, \mathsf{r}, x_{b_1}, \eta_1\rangle) = \langle t_1, \mathsf{r}, x_{\neg a_1}, \pi_1\rangle$ (if $a_1 \neq b_1$). In summary,

$$a_1 = b_1 \implies \mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{w}, c, \pi_1\rangle, \langle t_2, \mathsf{w}, c, \eta_1\rangle) \tag{3}$$

Fragment π_i ($i \geq 2$). For each $i \geq 2$, the fragment π_i is the following

$$\pi_i = \langle t_1, \mathsf{w}, y_{a_i}\rangle\langle t_1, \mathsf{r}, c\rangle\langle t_1, \mathsf{w}, c\rangle\langle t_1, \mathsf{w}, r_{a_i}\rangle.$$

Let us list some reads-from dependencies introduced due to π_i ($i \geq 2$):

$$\mathsf{rf}_w(\langle t_1, \mathsf{r}, y_{a_2}, \pi_i\rangle) = \langle t_1, \mathsf{w}, y_{a_2}, \pi_i\rangle \qquad \text{and} \qquad \mathsf{rf}_w(\langle t_3, \mathsf{r}, c, \pi_i\rangle) = \langle t_3, \mathsf{w}, c, \pi_{i-1}\rangle$$

The reads-from mapping $\mathsf{rf}_w(\langle t_1, \mathsf{r}, c, \pi_2\rangle) = \langle t_1, \mathsf{w}, c, \pi_1\rangle$, together with Equation (3), implies that $\langle t_1, \mathsf{r}, c, \pi_2\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, c, \eta_1\rangle$ (see Proposition A.1), in the case $a_1 = b_1$. Finally, the read-from dependency from t_1 to t_3 due to f gives us:

$$a_1 = b_1 \implies \mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{w}, y_{a_2}, \pi_2\rangle, \langle t_2, \mathsf{w}, c, \eta_1\rangle) \tag{4}$$

Fragment η_i ($i \geq 2$). For each $i \geq 2$, the fragment η_i is the following

$$\eta_i = \langle t_2, \mathsf{w}, y_{b_i}\rangle\langle t_2, \mathsf{w}, c\rangle.$$

It is easy to see from Equation (4) that if $a_1 = b_1$ then the intra-thread dependency between η_1 and η_2 further implies that $\langle t_1, \mathsf{w}, y_{a_2}, \pi_2\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, y_{b_2}, \eta_2\rangle$. Now, if additionally $a_2 = b_2$, we get:

$$(\forall i \leq 2, a_i = b_i) \implies \mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{r}, y_{a_2}, \pi_2\rangle, \langle t_2, \mathsf{w}, y_{b_2}, \eta_2\rangle) \quad (5)$$

.

In fact, the same reasoning can be inductively extended for any $2 \leq k \leq n$:

$$(\forall i \leq k, a_i = b_i) \implies \mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{r}, y_{a_k}, \pi_k\rangle, \langle t_2, \mathsf{w}, y_{b_k}, \eta_k\rangle). \quad (6)$$

The base case of $k = 2$ follows from Equation (5). For the inductive case, assume that the statement holds for some $k < n$, and that $(\forall 2 \leq i \leq k, a_i = b_i)$. Together with intra-thread dependencies, we have $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{w}, c, \pi_k\rangle, \langle t_2, \mathsf{w}, c, \eta_k\rangle)$. It follows from Proposition A.1 that $\langle t_1, \mathsf{w}, c, \pi_{k+1}\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, c, \eta_k\rangle$ This, in turn implies that $\langle t_1, \mathsf{w}, y_{a_{k+1}}, \pi_{k+1}\rangle$ is causally ordered before $\langle t_2, \mathsf{w}, y_{b_{k+1}}, \eta_{k+1}\rangle$. Thus, $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_1, \mathsf{r}, y_{a_{k+1}}, \pi_{k+1}\rangle, \langle t_2, \mathsf{w}, y_{b_{k+1}}, \eta_{k+1}\rangle)$ if $a_{k+1} = b_{k+1}$.

Fragments κ and δ. The sequence κ and δ are:

$$\kappa = \langle t_1, \mathsf{w}, u\rangle\langle t_1, \mathsf{r}, c\rangle\langle t_1, \mathsf{r}, u\rangle\langle t_2, \mathsf{w}, u\rangle \quad \text{and} \quad \delta = \langle t_2, \mathsf{r}, u\rangle$$

The reads-from dependencies induces due to κ and δ are:

$$\mathsf{rf}_w(\langle t_1, \mathsf{r}, c, \kappa\rangle) = \langle t_1, \mathsf{w}, c, \pi_n\rangle, \quad \mathsf{rf}_w(\langle t_1, \mathsf{r}, u, \kappa\rangle) = \langle t_1, \mathsf{w}, u, \kappa\rangle, \quad \mathsf{rf}_w(\langle t_2, \mathsf{r}, u, \delta\rangle) = \langle t_2, \mathsf{w}, u, \kappa\rangle$$

Correctness. Let us make some simple observations. First, every read event has a write event on the same variable prior to it, and thus, $\mathsf{rf}_w(i)$ is well defined for every $i \in \mathsf{Reads}_w$ Second, it is easy to see that the construction can be performed by a transducer $\mathcal{M}_n$ in $O(1)$ space.

($\Rightarrow$) Let us first prove that if $\forall i \in [n], a_i = b_i$, then $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \theta_1, \theta_2)$, where θ_1 is the index of the $\langle t_1, \mathsf{r}, u\rangle$ event in κ and θ_2 is the index of the $\langle t_2, \mathsf{w}, u\rangle$ event in κ. Recall that if $a_i = b_i$ for every $i \leq n$, then $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \langle t_3, \mathsf{r}, y_{a_n}, \pi_n\rangle, \langle t_2, \mathsf{w}, y_{b_n}, \eta_n\rangle)$ (see Equation (6)). As a result, $\langle t_1, \mathsf{w}, c, \pi_n\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, c, \eta_n\rangle$. Next, due to Proposition A.1, we get $\langle t_1, \mathsf{r}, c, \kappa\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, c, \eta_n\rangle$. This, together with intra-thread dependency further gives $\langle t_1, \mathsf{w}, u, \kappa\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{r}, u, \delta\rangle$. If we next apply Proposition A.1, we get $\langle t_1, \mathsf{w}, u, \kappa\rangle <^{w}_{\equiv_{\mathsf{rf}}} \langle t_2, \mathsf{w}, u, \kappa\rangle$. Applying Proposition A.1 once again, we conclude that $\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \theta_1, \theta_2)$

($\Leftarrow$) Let us now prove that if there is an index i for which $a_i \neq b_i$, then $\theta_1 \not<^{w}_{\equiv_{\mathsf{rf}}} \theta_2$ and $\theta_2 \not<^{w}_{\equiv_{\mathsf{rf}}} \theta_1$. To show this, we will construct an execution w' such that $w' \equiv_{\mathsf{rf}} w$ and $\theta_2 \leq^{w'}_{\mathsf{tr}} \theta_1$. In the rest, we assume i is the least such index.

First, consider the case when $i = 1$. In this case, $\neg a_1 = b_1$. Then, w' is the following concatenated sequence (κ' is the second largest prefix of κ):

$$w' = \langle t_2, \mathsf{w}, u\rangle\langle t_1, \mathsf{w}, x_{\neg a_1}\rangle\eta\delta\langle t_1, \mathsf{w}, x\rangle\langle t_1, \mathsf{w}, x_{a_1}\rangle\pi_2 \cdots \pi_n\kappa'$$

First, observe that, the thread-wise projections of w and w' are the same. Next, we note that the reads-from dependencies of each read access to either y_0, y_1, c or u are the same in both w and w':

- The only read accesses to y_0 or y_1 are in thread t_1, and so are their corresponding write events (as per w). In the new sequence w', we ensure that all write access to y_0 or y_1 in t_2 occur before any access to y_0 or y_1 in thread t_1.

- The same reasoning as above applies to the accesses to c.
- The read access to u in t_2 appears before the write access to u in t_1 in the new sequencd w'.

Finally, the reads-from dependency of $\langle t_2, \mathsf{r}, x_{b_1} \rangle$ is also preserved. Hence, $w' \equiv_{\mathsf{rf}} w$. Finally, observe that the corresponding events $\theta_2 = \langle t_2, \mathsf{r}, u, \kappa \rangle$ and $\theta_1 = \langle t_1, \mathsf{w}, u, \kappa \rangle$ appear in inverted order.

Next, consider the case of $i > 1$. Then, we construct w' as the following interleaved sequence:

$$\gamma_1 \gamma_2 \cdots \gamma_n \kappa'$$

where:

$$\begin{array}{rcll}
\gamma_1 & = & \langle t_2, \mathsf{w}, u \rangle \langle t_1, \mathsf{w}, x_{\neg a_1} \rangle \langle t_1, \mathsf{w}, c \rangle \langle t_1, \mathsf{w}, x_{a_1} \rangle \langle t_2, \mathsf{r}, x_{b_1} \rangle & \\
\gamma_j & = & \langle t_1, \mathsf{w}, y_{a_j} \rangle \langle t_1, \mathsf{r}, c \rangle \langle t_2, \mathsf{w}, c \rangle \langle t_1, \mathsf{w}, c \rangle \langle t_1, \mathsf{r}, y_{a_j} \rangle \langle t_2, \mathsf{w}, y_{b_j} \rangle & (2 \leq j < i) \\
\gamma_i & = & \langle t_1, \mathsf{w}, y_{a_i} \rangle \langle t_1, \mathsf{r}, c \rangle \langle t_2, \mathsf{w}, c \rangle \langle t_2, \mathsf{w}, y_{b_i} \rangle \langle t_2, \mathsf{w}, c \rangle \langle t_1, \mathsf{w}, c \rangle \langle t_1, \mathsf{r}, y_{a_i} \rangle & \\
\gamma_j & = & \langle t_2, \mathsf{w}, y_{b_j} \rangle \langle t_1, \mathsf{w}, y_{a_j} \rangle \langle t_1, \mathsf{r}, c \rangle \langle t_2, \mathsf{w}, c \rangle \langle t_1, \mathsf{w}, c \rangle \langle t_1, \mathsf{r}, y_{a_j} \rangle & (j > i) \\
\kappa' & = & \langle t_2, \mathsf{r}, u \rangle \langle t_1, \mathsf{w}, u \rangle \langle t_1, \mathsf{r}, c \rangle \langle t_1, \mathsf{r}, u \rangle &
\end{array}$$

Here again, we observe that the new trace w' is such that its thread-wise projections are the same as in w. Further, every read event reads-from the same corresponding events in both w and w'. That is, $w \equiv_{\mathsf{rf}} w'$. Further, observe that the positions of θ_1 and θ_2 have been inverted. This trace w' for a special case is shown in Fig. 8b.

This finishes our proof of Theorem 3.1. □

Let us now turn to the proof of Theorem 3.2. This time, we focus on the following language.

$$L^{=}_{\text{pad},n} = \{\overline{a}\#^n\overline{b} \mid \overline{a}, \overline{b} \in \{0,1\}^n \text{ and } \overline{a} = \overline{b}\}$$

Lemma A.2. For any streaming algorithm that recognizes $L^{=}_{\text{pad},n}$ in time $T(n)$ and space $S(n)$, we have $T(n) \cdot S(n) \in \Omega(n^2)$.

Proof. We first observe that the communication complexity of checking equality between two n-bit strings is $\Omega(n)$. Consider a Turing Machine M that recognizes $L^{=}_{\text{pad},n}$ in time $T(n)$ and space $S(n)$, by possibly going back and forth on the input tape. Since M takes $T(n)$ time, it must only traverse 'across' the central padding '$\#^n$' atmost $\frac{T(n)}{n}$ times. Since the space usage is $S(n)$, each time M crosses the padding completely, it communicates at most $S(n)$ bits across the padding. Thus, the total number of bits that can be communicated is atmost $\frac{T(n)\cdot S(n)}{n}$ and thus we have $\frac{T(n)\cdot S(n)}{n} \in \Omega(n)$, giving us $T(n) \cdot S(n) \in \Omega(n^2)$. □

Proof Sketch for Theorem 3.2. The reduction in the proof of Theorem 3.1 can be modified to prove the time-space tradeoff. This time, we can show a reduction from $T(n) \cdot S(n) \in \Omega(n^2)$ as we did in the proof of Theorem 3.1. The only difference will be to add extra events in the run, corresponding to the padding string $\#^n$, for which we can use n write events $\mathsf{w}(d)$ in the fragment κ in thread t_1 after the $\mathsf{r}(u)$ event of t_1; here d is a fresh memory location. Observe that none of the $\mathsf{w}(d)$ events are read by any read events. Thus, the answer of $\text{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, \theta_1, \theta_2)$ does not get affected. Besides the reduction itself is a streaming one pass reduction that takes $O(n)$ time and $O(1)$ space. □

A.2 Proof of Theorem 3.3

We will use the pumping lemma to establish Theorem 3.3. Recall that, a context-free language L satisfies the following property:

$$\begin{array}{c}\text{Pumpable}(L) \equiv \text{There is a } n \geq 1 \text{ s.t. forall } s \in L \text{ with } |s| > n, \text{ there are strings } u, v, w, x, y \text{ s.t.} \\ s = uvwxy, |vwx| \leq n, |vx| \geq 1, \text{ and for all } i \geq 0, uv^iwx^iy \in L\end{array}$$

We will show that $\text{Pumpable}(L^{c,d}_{\text{ordered}})$ does not hold:

$$L^{c,d}_{\text{ordered}} = \{w \in \Sigma^* c \Sigma^* d \Sigma^* \mid c \text{ and } d \text{ are not causally concurrent in } w \text{ under } \equiv_{\text{rf}}\}$$

To argue this, we will construct a class of strings $\{s_n\}_{n \geq 1}$ such that $s_n \in L^{c,d}_{\text{ordered}}$ (for all $p \geq 1$) and satisfies $|s_n| > n$, but every way of splitting s_n can be pumped down to a language that is not in $L^{c,d}_{\text{ordered}}$.

Construction. Our construction of the string s_n essentially mimics the construction for the proof of Theorem 3.1. Thus, we will have 2 threads t_1 and t_2 in s_n, and the same set of variables as in this proof. Without loss of generality, we will take $c = \langle t_1, \mathsf{r}, u\rangle$ and $d = \langle t_2, \mathsf{w}, u\rangle$.

More concretely, s_n will be the run corresponding to the instance $0^n\#0^n \in L^{=}_n$. The idea behind the construction stems from the observation that the construction is tight, and removing one or more events from the run results into one of: (a) non well-formed run, (b) the two focal events $e_c = \langle t_1, \mathsf{r}, u\rangle$ and $e_d = \langle t_2, \mathsf{w}, u\rangle$ being unordered, or (c) the focal events to completely vanish. Any of these cases result in a string outside of $L^{c,d}_{\text{ordered}}$.

Formally, given $n \geq 1$, we construct the run $s_n = \pi\kappa\eta\delta$ as shown in Fig. 8a, where $\pi = \pi'_1\pi_2 \dots \pi_n$ and $\eta = \eta_1\eta_2 \dots \eta_n$ as described. Here, π'_1 is the same string as π_1 but does not contain the event $\langle t_1, \mathsf{w}, x_{\neg a_1}\rangle$. Now, we make the following observations; we use the notation $\sigma \setminus e$ and $\sigma \setminus X$ to denote the sequence obtained by removing from σ, the event e and the set of events X respectively.

Claim A.1. Let $X \subseteq \text{Events}_{s_n}$ be a set of events such that $X \neq \varnothing$ and $|X| \leq n$ and X contains events only from the segment $\pi\kappa$. The sequence $s_n \setminus X$ does not belong to $L^{c,d}_{\text{ordered}}$.

Proof. First, if X contains a write event but not an event that reads from it, then clearly $s_n \setminus X$ is not well-formed, and thus cannot belong to $L^{c,d}_{\text{ordered}}$. In the rest of the proof, we assume that for every write event in X, the corresponding read event is also in X.

If X contains the event $\langle t_1, \mathsf{w}, x_{a_1}\rangle$, then X must also contain $\langle t_2, \mathsf{r}, x_{b_1}\rangle$ which is not in $\pi\kappa$; so X cannot contain this event. If X contains the event $\langle t, \mathsf{w}, y_{a_i}\rangle$ for some $i \geq 2$, then it must also contain $\langle t_, \mathsf{r}, y_{a_i}\rangle$. More importantly, the proof of Theorem 3.1 argues that in fact in this case the write events in t_2 can be carefully reordered so that the resulting reordering is $\equiv_{\text{rf}}$ equivalent to $s_n \setminus X$ and thus c and d can be reordered. For example, if X contains $\langle t_1, \mathsf{w}, y_{a_1}\rangle$, then the reordering that places $\langle t_2, \mathsf{r}, x_{b_1}\rangle$ after $\langle t_1, \mathsf{w}, x_{a_1}\rangle$ but places $\langle t_2, \mathsf{w}, c\rangle\langle t_2, \mathsf{w}, y_{b_2}\rangle\langle t_2, \mathsf{w}, c\rangle$ after $\langle t_2, \mathsf{r}, x_{b_1}\rangle$ but before $\langle t_1, \mathsf{w}, y_{a_3}\rangle$ is the correct reordering that witnesses reordering of c and d. The same argument also shows why X cannot contain any $\langle t_1, \mathsf{r}, y_{a_i}\rangle$, $\mathsf{w}t_1, \mathsf{w}, c$ or $\mathsf{w}t_1, \mathsf{r}, c$. Also, $\langle t_1, \mathsf{w}, u\rangle$ cannot be in X; if so, then $\langle t_1, \mathsf{r}, u\rangle \in X$ and thus $s_n \setminus X$ does not contain c. ◻

Claim A.2. Let $X \subseteq \mathsf{Events}_{s_n}$ be a set of events such that $X \neq \varnothing$ and $|X| \leq n$ and X contains events only from the segment $\eta\delta$. The sequence $s_n \setminus X$ does not belong to $L^{c,d}_{\text{ordered}}$.

Proof. If X contains $\langle t_2, \mathsf{w}, u\rangle = d$, then $s_n \setminus X$ trivially is not in $L^{c,d}_{\text{ordered}}$. If X contains $\langle t_2, \mathsf{r}, x_{b_1}\rangle$, then there is no ordering from the event $\langle t_1, \mathsf{w}, x_{a_1}\rangle$ to an event of thread t_2, as a result of which d can be reordered before c as show in Fig. 8b. If X contains any $\langle t_2, \mathsf{w}, c\rangle$ of κ_i, then we do not get an ordering from $\langle t_1, \mathsf{r}, y_{a_{i+1}}\rangle$ to $\langle t_2, \mathsf{w}, y_{b_{i+1}}\rangle$. As a result, we can reorder c and d. The same reasoning can be used to argue why X cannot contain $\langle t_2, \mathsf{w}, y_{b_i}\rangle$ for any $i \geq 2$. □

Now, we consider a case-by-case analysis of how the substrings u, v, w, x, y of s_n are picked subject to the constraints $s_n = uvwxy$, $|vwx| \leq n$, $|vx| \geq 1$.

Case $vwx \subseteq \pi\kappa$ or $vwx \subseteq \eta\delta$. In this case, we can pump down (choose $i = 0$) and the resulting string $s'_n = uwy \notin L^{c,d}_{\text{ordered}}$ due to Claim A.1 and Claim A.2.

Case vwx spans $\pi\kappa\eta$. In this case, observe that there is no j such that vwx intersects with both π_j and η_j because of the restriction that $|vwx| \leq n$. Hence, choosing $i = 0$ (pumping down) again leads to a run $s'_n = uwy \notin L^{c,d}_{\text{ordered}}$, and this can be established using arguments similar to the proofs of Claim A.1 and Claim A.2.

A.3 Proof of Theorem 3.4

The proof of this theorem can be proved in a similar manner as an analogous result of [Mathur et al. 2020] in the context of data race detection. Given a set of events $X \subseteq \mathsf{Events}_w$, a partial order $P \subseteq X \times X$ which totally orders events of each thread, and a reads-from relation $RF : X \hookrightarrow X$ that maps each read event in X to a write event in X with the same variable, the RF-Poset realizability problem for (X, P, RF) asks if there is a linearization of P whose reads-from function matches RF. The following is the statement of the analogous result in [Mathur et al. 2020]:

Theorem A.1 (Lemma 5.6 in [Mathur et al. 2020]). Assuming SETH holds. RF-Poset realizability for posets with n events cannot be solved in time $O(n^{2-\epsilon})$ for every $\epsilon > 0$, even for inputs with 2 threads and 7 variables.

The proof of the above statement in fact constructs a simple RF-Poset with only two threads, and is such that the RF-Poset can be translated into a simple linearizartion (that first linearizes the events of the first thread, followed by the events of the second thread). This means that the RF-Poset realizability holds iff the linearization can be reordered so that the two focal events (read and write of z) can be flipped. Since our proof is not a direct reduction from RF-Poset realizability, we need to prove our result separately. However, since most parts of the proof are identical, we skip the entire construction and only outline the high level details, and highlight the low level details about where our construction differs.

Consider the sequence of threads τ_A and τ_B constructed by Theorem A.1 (dependin upon the two sequence of Boolean vectors A and B given as part of the OV instance). We will use two fresh variables z and u. Let τ'_A be the sequence obtained by (a) inserting a $\mathsf{w}(z)$ event after the event $\mathsf{w}^{a_1}_1(x_1)$, and (b) inserting a $\mathsf{w}(u)$ event before the event $\mathsf{r}^{a_{n/2}}_1(x_2)$ in τ_A. Likewise, τ'_B be the sequence obtained by (a) inserting the event $\mathsf{r}(z)$ before the event $\mathsf{r}^{b_1}_1(x_1)$, and (b) inserting $\mathsf{w}(u)$ event after $\mathsf{w}^{b_{n/2}}_1(x_2)$ in τ_B. Observe that when the entire τ'_A appears after τ'_B, we have $(\mathsf{w}^{a_1}_1(x_1), \mathsf{r}^{b_1}_1(x_1)) \in \mathsf{rf}$ and this imposes one of the desired orderings of the construction in [Mathur et al. 2020]. However the other ordering is not imposed, but this can be imposed when we instead ask the check order question.

Formally, let $\sigma = \tau'_A\tau'_B$. We claim that in σ, the two events $e_1 = \langle \tau_A, \mathsf{w}(u)\rangle$ and $e_2 = \langle \tau_B, \mathsf{w}(u)\rangle$ are reorderable under $\equiv_{\mathsf{rf}}$ iff the partial order P constructed in [Mathur et al. 2020] is realizable. For the forward direction, consider the witness reordering σ' in which e_1 and e_2 are reordered. Consider the run ρ obtained by removing from σ' the events of variables z and u. Observe that this is a witness to the realizability of the RF poset constructed in [Mathur et al. 2020] because σ' is rf-equivalent to σ and further e_2 appears before e_1 in σ', and thus σ' satisfies all the constraints of the RF poset instance. For the reverse direction, suppose that the RF poset instance of [Mathur et al. 2020] is realizable, then, consider the trace that realizes the poset, say ρ. In ρ, we can add the two write events on u and the read-write pair on z as described above. We remark that the resulting trace σ' is rf-equivalent to σ — the order between all thread-wise events is preserved as it was also a constraint in the poset. Further, the reads-from of all the events on $x_1, x_2, \ldots, x_7$ is preserved since this ρ is an instance of RF-poset and adding events on z and u doesnt change any other variables' reads-from. Since the ordering on $f_1 = \mathsf{w}_1^{a_1}(x_1)$ and $f_2 = \mathsf{r}_1^{a_{n/2}}(x_1)$ was preserved (as $f_1 <^{\rho} f_2$) because of the constraint in the poset, it implies that the event immediately preceding f_1 (namely $g_1 = \langle \tau_A, \mathsf{w}(z)\rangle$) and immediately succeeding f_2 (namely $g_1 = \langle \tau_B, \mathsf{r}(z)\rangle$)) must also be preserved as $g_1 <^{\sigma'} g_2$, and since these two are the only two events on z, they are also a RF pair. Finally, the two focal events get reversed because of the other ordering in the poset. his means σ' thus obtained is a witness of the reoderability of e_1 and e_2 in σ.

A.4 Proof of Theorem 3.5

Overview. Our proof is derived from similar statements in [Gibbons and Korach 1994] and [Mathur et al. 2020]. The idea behind the algorithm is to search for a path over an abstract representation of the search space. The search space will be represented as a graph, also called the 'frontier graph' [Gibbons and Korach 1994]. The nodes of this graph are subsets of Events_w which are downward closed subsets of $(\mathsf{po}_w \cup \mathsf{rf}_w \cup (e, f))^*$, assuming this is a partial order (if not, we can directly say NO). The edges represent 'extensions' by one event. The existence of a witnessing rf-equivalent run w' can then be checked by checking if there is a path from the node representing the empty subset to the (unique) node corresponding to the set Events_w. The proof of correctness will argue the correctness of this abstraction.

Definition A.1 (Ideal). Let P be a partial order on Events_w such that $(\mathsf{po}_w \cup \mathsf{rf}_w) \subseteq P$. An ideal X of P is a subset of Events_w such that for every pair of events $e, f \in \mathsf{Events}_w$ such that $(e, f) \in P$, if $f \in X$, then $e \in X$.

Remark 3. Observe that an ideal contains no information about relative ordering (linearization) of events, but just the set of events. Further, the empty set $\varnothing$ is also an ideal of P.

Definition A.2 (Extension). Given two ideals X_1, X_2 of a partial order P, we say that X_2 extends X_1 if there is an event e such that

(1) $X_2 = X_1 \uplus \{e\}$,

(2) if e is a write event (on variable x), and if there is another write event $e' \neq e$ on variable x in X_1, then we must have $r \in X_1$ for every r such that $\mathsf{rf}_w(r) = e'$.

We also say that the event e extends X_1 to X_2.

Proposition A.2. Observe that for an ideal X, there are at most $|\mathcal{T}|$ ideals that are extensions of X.

Remark 4. The notion of extension above is 'conservative' in the sense that it is aware of the future. In particular, our goal is to incrementally construct extensions, all the way until we reach the set Events_w. As a result, some write e' is not allowed to be added to an ideal because that will disable a future read r. If on the other hand, our goal was to not reach the ideal Events_w but only to reach a subset that does not contain the read r, then e' could be added.

Definition A.3 (Frontier Graph)**.** Let P be a partial order on Events_w. The frontier graph $G_P = (V_P, E_P)$ of P is a directed graph such that:

(1) V_P is the set of ideals of P

(2) $(X_1, X_2) \in E_P$ if X_2 is an extension of X_1.

Edges in this graph are implicitly labeled by events. For the edge $(X_1, X_2) \in E_P$ such that $\{e\} = X_2 - X_1$, the label is e and we will denote this as $X_1 \rightarrow_e X_2$.

Remark 5. The size of G_P is $O(|\mathcal{T}| \cdot |w|^{|\mathcal{T}|})$ because there are at most $|w|^{|\mathcal{T}|}$ nodes, each having at most $|\mathcal{T}|$ outgoing edges. The time to construct it is $O(|\mathcal{T}| \cdot |w|^{|\mathcal{T}|})$ because for every edge you spend time $O(|w|)$ to check if the edge corresponds to an extension.

Definition A.4 (Respecting a partial order)**.** Let S be a set and let P be a partial order over S. Let $S' \subseteq S$ and let ρ be a permutation of elements of S'. We say that ρ respects P if for every $(e, f) \in P$, if $f \in S'$ then $e \in S'$ and further, e appears before f in ρ.

Definition A.5 ($(\mathsf{po}, \mathsf{rf})$-preserving subrun)**.** Given a run ρ, we say that ρ is $(\mathsf{po}, \mathsf{rf})$-preserving subrun of w if

(1) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_w$,

(2) Events_ρ respects $\mathsf{po}_w \cup \mathsf{rf}_w$

(3) For every read event $r \in \mathsf{Events}_w$, with $w = \mathsf{rf}_w(r)$, there is no other write event $w' \neq w$ on the same variable as r that appears between w and r in ρ.

Remark 6. If ρ is a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w, and further if $\mathsf{Events}_\rho = \mathsf{Events}_w$, then ρ is rf-equivalent to w.

Remark 7. Not every $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w can be extended to an rf-equivalent run. In other words, there is a w and a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun ρ of w for which every extension $\rho \cdot \gamma$ is not rf-equivalent to w

Lemma A.3. *Let P be a partial order over Events_w and let $G_P = (V_P, E_P)$ be the frontier graph of P. For every ideal $X \in V_w$ such that X is reachable from $\varnothing \in V_P$, there is a run ρ such that ρ is a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w and ρ respects P (i.e., for every two events $e, f \in \mathsf{Events}_\rho$ such that $(e, f) \in P$, we also have that e appears before f in the run ρ).*

Proof. We prove the following stronger statement that, in fact, every path $X_0, X_1, \ldots, X_n$ from $\varnothing$ to X labeled $\sigma_X = e_1 e_2 \ldots e_n$ is indeed a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w that respects P. Here, where e_i is the i^{th} event in the path, i.e., $X_i \rightarrow_{e_{i+1}} X_{i+1}$. Observe that $X = \{e_1, \ldots, e_n\}$.

We prove this by inducting on the length of w_X (alternatively on the size of X).

Base case: In this case $X = \varnothing$ and $w_X = \epsilon$ and the statement holds vacuously.

Inductive case: Suppose we have that for every labeled path of length $\leq i$, the statement holds true. Consider now a labeled path w_X of length $i+1$. Let e be the last event in this path. We consider two cases based on the type of e.

$e = \langle t, \mathtt{w}(x)\rangle$. That is, e is a write event. Suppose on the contrary that w_X is not $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w that respects P. Consider the penultimate prefix w_Y such that $w_X = w_Y \circ e$; we will also use $Y = X \setminus \{e\}$. By inductive hypothesis, w_Y is a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w that also respects P. T hen, there is an event $f \in Y$ such that $(e, f) \in \mathsf{po}_w \cup \mathsf{rf}_w \cup P$. But this means that Y is not an ideal of P, a contradiction.

$e = \langle t, \mathtt{r}(x)\rangle$. That is, e is a read event. Suppose on the contrary that w_X is not $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w that respects P. Then either we have that the penultimate set $Y = X \setminus \{e\}$ is not an ideal, as in the previous case giving a contradiction, or we have the more interesting case where w_X is not a $(\mathsf{po}, \mathsf{rf})$-preserving-subrun of w becuase there is a write $f' \neq f$ (where $f = \mathsf{rf}_w(e)$) such that f' appears later than f in w_Y. Consider the largest prefix w_Z of w_Y that does not contain f'. Also, let w_U be the immediately longer path, i.e., $w_U = w_Z \circ f'$. We will use Z and U to denote the ideals r eached after taking the paths w_Z and w_U respectively. Observe that $U = Z \uplus \{f'\}$, $e \notin Z$, $f = \mathsf{rf}_w(e) \in Z$ and $Z \longrightarrow_{f'} U$. This contradicts that U is an extension of Z.

□

Remark 8. The graph G_P defined above does not capture all $(\mathsf{po}, \mathsf{rf})$-preserving-subruns of w. It only captures those that are prefixes of some run that is rf-equivalent to w. Thus the converse of Lemma A.3, as stated, is not true. The following lemma is true though (since it talks about rf-equivalent executions).

Lemma A.4. Let P be a partial order over Events_w and let $G_P = (V_P, E_P)$ be the frontier graph of P. Let $\rho = f_1 f_2 \ldots f_N$ be a run that is rf-equivalent to w such that e_2 appears before e_1 in ρ. Then there is a path in G_P that is labeled with ρ.

Proof. We establish that ρ is a path in the graph G_P by inductively establishing that X_i is a node of G_P and $X_{i-1} \longrightarrow_{f_i} X_i$ is an edge of G_P for every $i \geq 1$.

Base Case (i=1). Since ρ is rf-equivalent to w, f_1 must be the first event of its own thread and further it must be a write event. Also, f_1 cannot be the focal event e_1. Thus, the set $X_1 = \{f_1\}$ is indeed an ideal of P and thus $X_1 \in V_P$ is a node of the graph G_P. Next, since there is no other write event in $X_0 = \varnothing$, f_1 trivially extends X_0 to X_1. Thus, $X_0 \longrightarrow_{f_1} X_1$ is an edge in G_P.

Inductive Case. Suppose that X_i is an ideal of G_P. Consider the set $X_i = X_{i-1} \uplus \{f_i\}$. Since ρ is rf-equivalent to w, all events po-ordered before f_i must be in the prefix $f_1, \ldots, f_{i-1}$ and thus in the set X_{i-1}. Likewise, if f_i is a read event, then $\mathsf{rf}_w(f_i)$ is also in X_{i-1}. Finally since ρ does not execute e_1 before executing e_1, if $f_i = e_1$, then $e_1 \in X_{i-1}$. In other words, X_i is an ideal of P and thus a node in G_P. Next, consider the case when f_i and suppose on thae contrary that there is a write event $w \neq f_i$ and a read event r with $\mathsf{rf}_w(r) = w$ such that $w \in X_{i-1}, r \notin X_{i-1}$. This will contradict that ρ is rf-equivalent to w as f_i appears between w and r in ρ. Hence, f_i extends X_{i-1} to X_i. As a result, $X_{i-1} \longrightarrow_{f_i} X_i$ is an edge in G_P □

A.4.1 Algorithm for solving causallyOrdered$_{\equiv_{\mathsf{rf}}}(\cdot,\cdot,\cdot)$.

Theorem A.2. Algorithm 1 runs in time $O(|\mathcal{T}| \cdot |w|^{|\mathcal{T}|+1})$ for an input run w. Further, Algorithm 1 returns YES iff $\neg\mathsf{causallyOrdered}_{\equiv_{\mathsf{rf}}}(w, e_1, e_2)$

Algorithm 1: Polynomial time algorithm for Checking Causal Orderedness under $\equiv_{\mathsf{rf}}$

```
Input: Run $w \in \Sigma^*$, events $e_1, e_2 \in \mathsf{Events}_w$ such that $e_1$ appears before $e_2$ in $w$
Output: YES iff there is a run $w'$ which is rf-equivalent to $w$ in which $e_2$ appears before $e_1$.
1 Construct the transitive reduction of the quasi order $P = (\mathsf{po}_w \cup \mathsf{rf}_w \cup \{e_2, e_1\})$;
2 if $P$ has a cycle then return NO;
3 Construct the frontier graph $G_P = (V_P, E_P)$ as in Definition A.3 ;
4 if $(\varnothing, \mathsf{Events}_w) \in E_P^+$ then
5     return YES
6 else
7     return NO
```

Proof. The time spent in constructing P and checking cycles in it is $O(|w|)$. The time taken to build G_P is $O(|\mathcal{T}| \cdot |w|^{|\mathcal{T}|+1})$ and the time taken to check reachability in G_P is $O(|\mathcal{T}| \cdot |w|^{|\mathcal{T}|})$

Let us now argue correcntess.
($\Rightarrow$) Suppose the algorithm says YES. Then P is a partial order. Further the node $X = \mathsf{Events}_w$ is reachable from $\varnothing$ in G_P. Then by Lemma A.3, there is a run ρ that is rf-equivalent to w and flips e_1 and e_2.

($\Leftarrow$) Suppose there is an rf-equivalent run $\rho = f_1 f_2 \ldots f_{|w|}$ of w such that e_2 appears before e_1 in ρ. First, observe that ρ respects $\mathsf{po}_w \cup \mathsf{rf}_w \cup (e_2, e_1)$ and thus P must be acyclic. By Lemma A.4, we must have that $X = \mathsf{Events}_w = \{f_1, \ldots, f_{|w|}\}$ is reachable from $\varnothing$ in G_P. Thus the algorithm returns YES. □

A.5 Proof of Theorem 3.6

Proof. At a high level, the algorithm enumerates permutations of the given run w, one at a time, and checks if each such permutation is rf-equivalent to w and contains e and f appearing in the reverse order. Further, this task can be performed in total additional space (besides the input w) that is linear in $|w|$. In the following, we explain how these two tasks can be performed by a deterministic Turing machine M with a linearly bounded work tape.

Generating permutations in linear space. Given the run w, M first stores the lexicographically minimum permutation w_{min} of w into a separate worktape. This can be done by maintaining a counter that counts the number of occurences of each symbol in Σ and copying the required copies of each letter to the work tape. Next, given the current contents u of the work tape, M identifies the longest non-increasing suffix v of u, identifies the symbol a right before it (and thus $u = u_1 \cdot a \cdot v$ for some u). Next, M identifies the rightmost symbol b of v that is lexicographically larger than a (and thus $v = v_1 \cdot f \cdot v_2$), swaps a and b to obtain the string $x = u_1 \cdot b \cdot v_1 \cdot a \cdot v_2$, and finally reverses the suffix $v_1 \cdot a \cdot v_2$ to obtain the next permutation $u' = u_1 \cdot b \cdot v_2^{\mathsf{rev}} \cdot a \cdot v_1^{\mathsf{rev}}$. Clearly, all this operations can be performed in place on the worktape.

Checking each permutation. Given a string u on the worktape, M also needs to check if $u \equiv_{\mathsf{rf}} w$ and whether the order of occurence of e and f is different in u and w. This can also be checked using no additional space as follows. First, M checks if u and w have the same program order (i.e., $\mathsf{po}_w = \mathsf{po}_u$) by making $|\mathcal{T}|$ many passes over u and w, one for each $t \in \mathcal{T}$, checking if the projection of u and w to t match. Next, M checks if $\mathsf{rf}_w = \mathsf{rf}_u$ by making a separate pass for each read event r, and verifying if the corresponding last write event w on the same variable as r and occuring before r is performed by the same thread, and at the same local index in that thread.

Finally, checking if e and f appear reversed corresponds to checking if the i^{th} event of thread t appears after the j^{th} event of thread t' in u, where t, t' are respectively the threads performing e, f, and i, j are their respective local indices in t, t'. The check of whether some two occurences of c and d can be reversed is analgous. M performs each of these checks in place on the worktape, using only $\log |w|$ many extra bits to keep track of counters and local indices. □

B PROOFS FROM SECTION SECTION 4

B.1 Proof of Theorem 4.1

Proof. Let us focus only on words w where w is of the form ab^nc^m. Given the relation I, we claim that a and last c are concurrent iff $n < m$, and ordered otherwise. It is clear that if $m \leq n$, then c can be reordered to before a using a swap sequence like this:

$$ab^nc^m \rightarrow abcb^{n-1}c^{m-1} \rightarrow \cdots \rightarrow a(bc)^m b^{n-m} \rightarrow bca(bc)^{m-1}b^{m-1} \rightarrow \cdots \rightarrow (bc)^m ab^{m-1}$$

where each arrow in the first group corresponds to a (b, bc) swap and each arrow in the second group corresponds to a an (a, bc) swap. Let us argue why otherwise, the last c is ordered wrt a. The only way that the last occurrence of c can be reordered against a is that all occurrences of c have already been moved behind a; since occurrences of c do not commute against each other. Every time an occurrence of c is reordered with a, it must be through a swap $\ldots abc \rightarrow bca$, because those are the only elements of I that involve an a and a c on the two sides of a swap. This swap *consumes* one b, and this b cannot move back, unless it moves back together with a c, which would be counterproductive. Therefore, we need at least as many b's as c's to swap all the c's behind the a.

A monitor, therefore, should be able to distinguish the two sets of strings ab^nc^m where $n < m$ and where $n \geq m$ from each other. But, this involves counting and therefore is not regular. □

B.2 Proof of Theorem 4.2

Proof. $\Leftarrow$ direction: If $gg' \not\equiv_{\mathsf{rf}} g'g$ for some pair of grains, then it is straightforward to see that if they are consecutive and swapped, the soundness will be violated. For the additional conditions, the grains from the run in Example 4.1 satisfy $gg' \equiv_{\mathsf{rf}} g'g$ but violate the extra condition on the grains for the theorem and as argued the corresponding commutativity relation is not sound in the context of the run.

$\Rightarrow$ direction: We have to show that if $[w]_G \not\subseteq [w]_{\mathsf{rf}}$, then at least one of the conditions of the theorem is violated. Assume there exists $u \in [w]_G$ where $u \notin [w]_{\mathsf{rf}}$. The latter can happen only if in u, either the program order or the reads-from relation are changed compared to w. Since $gg' \equiv_G g'g$ implies that program order is never changed (and any standard Mazurkiewicz swap preserves program order), the only point of change can be in the reads-from relation.

We prove, by contradiction, that the reads-from relation cannot change. Let us consider the sequence of swaps that would get us from w to u:

$$w \xrightarrow{s_1} v_1 \xrightarrow{s_2} v_2 \xrightarrow{s_3} \ldots \xrightarrow{s_n} u$$

Let us assume v_i is the first word in the sequence such that $v_i \not\equiv_{\mathsf{rf}} w$, and as such $v_i \not\equiv_{\mathsf{rf}} v_{i-1}$. Therefore, the swap s_i is to blame. By definition, this cannot be a Mazurkiewicz swap. Therefore, it is a swap of two grains g and g' where $(g, g') \in I_G$. Note that any pair of $\mathtt{w}(x)$ and $\mathtt{r}(x)$ that are both on one side of this swap will be unaffected by the swap. Therefore, a change in the reads-from relation must be of one of the following forms, for a given variable x:

- $\mathtt{w}(x)$ is before gg' in v_{i-1} and the corresponding $\mathtt{r}(x)$ is somewhere in gg'. A swap can brake such a relation only if $\mathtt{r}(x)$ is in g, there are no other writes to x before $\mathtt{r}(x)$ in g, and g' contains a write to x. This is however in violation of the condition stated in the theorem.
- $\mathtt{w}(x)$ is before gg' in v_{i-1} and the corresponding $\mathtt{r}(x)$ is after gg': This means that gg' contains no write to variable x, and therefore, the swap cannot affect the fact that $\mathtt{r}(x)$ reads from $\mathtt{w}(x)$.
- $\mathtt{w}(x)$ is in gg' in v_{i-1} and the corresponding $\mathtt{r}(x)$ is also in gg': This would contradict $gg' \equiv_{\mathsf{rf}} g'g$.
- $\mathtt{w}(x)$ is in gg' in v_{i-1} and the corresponding $\mathtt{r}(x)$ is after gg'. This is in violation of the condition stated in the theorem.

□

C PROOFS OF SECTION 5

C.1 Proof of Theorem 5.1

The proof sketch we provided in the text, through a two-pass construction is formal enough to convince a reader with a good command of automata theory. Alternatively, one can give the full construction to this monitor as follows.

The state of the grain monitor is conceptually the same as the trace concurrency monitor, except that E is no longer a set of *events*, but rather a set of *grain signatures*. Additionally, the state includes one grain signature g that maintains information about the current grain, a flag $i : Bool$ that is true whenever the monitor is in the middle of reading a grain, and a flag $C : \mathcal{X} \to Bool$ which is the set of variables wrt which some previously closed grain was (nondeterministically) assumed to satisfy the part of condition 2 which assumes all the reads, that correspond to a write in some grain, also belong to the grain; therefore, the monitor does not expect to see any reads on these variables before it sees a write. The monitor makes a nondeterministic guess in the second component of a grain signature, for each variable on whether all the reads of a given write are included in the grain, and later checks the (right) context of the grain to verify this guess. In effect, the monitor is able to make both choices about the grain and therefore, try its luck with both versions of the (sound) commutativity relations that are implied by the choice. A wrong choice will later result in a halt. Fig. 9 lists the transitions of the monitor, and the definitions of the operators used.

Proof. The full proof of why the detailed monitor is correct is long and tedious, through case analysis. We sketch the high level idea behind the most interesting case of the proof. The proof is by induction on the length of the portion of the input run so far processed by the monitor. We assume that the monitor has processed the prefix σ (which includes e but not e') and is about to read the next entity a, and assume that it satisfies the following induction hypothesis:

There exists a resolution of nondeterministic choices such that:

- *For any entity b (event or grain) in σ:*
$$(grain(b) \subseteq E \iff b \text{ is ordered wrt } e)$$
- $E \subseteq \{grain(d) |\ d \in \sigma\}$
- *For every entity in E, the monitored has correctly guessed the status of condition 2.*

and any error in underestimating E will eventually result in a halt.

We split the proof on whether a is concurrent or ordered wrt e:

State	Event	State Update
$\langle -, E, g_\emptyset, i, C\rangle$	$e \in \Sigma$	$\langle -, E, g_\emptyset, i, C\rangle$
$\langle -, E, g_\emptyset, i, C\rangle$	$\diamond_1$	$\langle \diamond_1, E, g_\emptyset, i, C\rangle$
$\langle \diamond_1, E, g, i, C\rangle$	$e = \langle i, \mathsf{w}(x)\rangle$	$\langle \diamond_1 -, \{grain(e)\}, g, false, C[x \mapsto nondet]\rangle$
$\langle \diamond_1, E, g, i, C\rangle$	$e = \langle i, \mathsf{r}(x)\rangle$	$\langle \diamond_1 -, \{grain(e)\}, g, false, C\rangle$
$\langle \diamond_1, E, g, i, C\rangle$	$\rhd$	$\langle \diamond_1 -, \emptyset, g_\emptyset, true, C\rangle$
$\langle \diamond_1 -, E, g, true, C\rangle$	$e = \langle i, \mathsf{w}(x)\rangle$	$\langle \diamond_1 -, E, update(g, e), true, C\rangle$
$\langle \diamond_1 -, E, g, true, C\rangle$	$e = \langle i, \mathsf{r}(x)\rangle$	$\langle \diamond_1 -, E, update(g, e), true, C\rangle$
$\langle \diamond_1 -, E, g, true, C\rangle$	$\lhd$	$\langle \diamond_1 -, E \odot ND(g), g_\emptyset, false, C \otimes ND(g)\rangle$
$\langle \diamond_1 -, E, g, false, C\rangle$	$e = \langle i, \mathsf{w}(x)\rangle$	$\langle \diamond_1 -, E \odot grain(e), g, false, C[x \mapsto nondet]\rangle$
$\langle \diamond_1 -, E, g, false, C[x \mapsto false]\rangle$	$e = \langle i, \mathsf{r}(x)\rangle$	$\langle \diamond_1 -, E \odot grain(e), g, false, C\rangle$
$\langle \diamond_1 -, E, g, i, C\rangle$	$\diamond_2$	$\langle \diamond_1 - \diamond_2, E, g, i, C\rangle$
$\langle \diamond_1 - \diamond_2, E, g, i, C\rangle$	$e \in \Sigma$	$\langle Ord(E, ND(grain(e))), E, g, i, C \otimes ND(grain(e))\rangle$
$\langle \diamond_1 - \diamond_2, E, g, i, C\rangle$	$\rhd$	$\langle \diamond_1 - \diamond_2 \rhd, E, g_\emptyset, true, C\rangle$
$\langle \diamond_1 - \diamond_2 \rhd, E, g, i, C\rangle$	$e = \langle i, \mathsf{w}(x)\rangle$	$\langle \diamond_1 - \diamond_2 \rhd, E, update(g, e), true, C\rangle$
$\langle \diamond_1 - \diamond_2 \rhd, E, g, i, C[x \mapsto false]\rangle$	$e = \langle i, \mathsf{r}(x)\rangle$	$\langle \diamond_1 - \diamond_2 \rhd, E, update(g, e), true, C\rangle$
$\langle \diamond_1 - \diamond_2 \rhd, E, g, i, C\rangle$	$\lhd$	$\langle Ord(E, ND(g)), E, g_\emptyset, false, C \otimes ND(g)\rangle$
$\langle false, E, g, i, C\rangle$	$e = \langle i, \mathsf{w}(x)\rangle$	$\langle false, E, g, i, C\rangle$
$\langle false, E, g, i, C[x \mapsto false]\rangle$	$e = \langle i, \mathsf{r}(x)\rangle$	$\langle false, E, g, i, C\rangle$

$$update(g, \langle i, \mathsf{op}(x)\rangle) = \begin{cases} g.E = g.E \cup \{e\}, g.V = g.V \cup \{x\} & \text{if } (\mathsf{op} = \mathsf{r} \land \mathsf{w}(x) \notin g.E) \\ g.E = g.E \cup \{e\} & \text{owise} \end{cases}$$

$$Ord(E, g) \iff \exists g' \in E : \ depend(g, g') \qquad C \otimes g = C - \{x | \langle i, \mathsf{w}(x)\rangle \in g.E\} \cup g.V$$

$$E \odot g = \begin{cases} E \cup g & \text{if } Ord(E, g) \\ E & \text{owise} \end{cases} \qquad ND(g) = \langle g.E, g.V \cup \{x | \ \langle i, \mathsf{w}(x)\rangle \in g.E \land \ \text{nondet}\}\rangle$$

Fig. 9. Grain Concurrency Monitor: The monitor starts in $\langle -, \emptyset, g_\emptyset, false, \emptyset\rangle$ and accepts if it is in a state $\langle false, \dots\rangle$ once the input run is read. $g_\emptyset$ corresponds to an empty grain signature. $grain(e)$ is syntactic sugar for $update(g_\emptyset, e)$. With ND, the monitor nondeterministically decides for which of the writes that appear in the grain, all the read operations are also assumed to be in the grain.

- Ordered: Consider the definition of $[\sigma a]_G$. It is simpler to move to the corresponding grain monoid and consider the equivalence class of $[h_G(\sigma a)]_{\widehat{I_G}}$. By definition, e and a are always ordered the same way in $[h_G(\sigma a)]_{\widehat{I_G}}$.

  In the grain monoid, this implies the existence of a path in the partial order representation of the class. Let this path be:

$$e \longrightarrow a_1 \longrightarrow \cdots \longrightarrow a_m \longrightarrow a$$

  where each a_i is either in Σ or in Σ_G. The induction hypothesis implies that:

$$\{e\} \cup events(a_1) \cup \dots events(a_m) \subseteq E$$

  Since $a_m \longrightarrow a$, we know the two items do not commute. The reasons for this could be that:

  – They share a thread: then the monitor correctly decides that the new event a is ordered and adds its signature to E.

  – They share a variable x, at least one writes to x, and condition 2 does not hold. Since the grain summaries of a_m in E is correctly computed (and guessed) by the induction hypothesis, and the correct guess for a is an option, we correctly see a as ordered with respect to the summary E.

  As such, the monitor adds a to E and re-established all the hypotheses.

  If as result of a wrong guess (with *ND*), we incorrectly establish that a and a_m commute, then we have under-estimated E and have to argue that the computation will eventually halt.

  The wrong guess implies that there exists a $\mathtt{r}(x)$ in the remainder of the concurrent run that is matched with a $\mathtt{w}(x)$ in a. The monitor made the mistake of not including x in the V component of he grain signature of a, and therefore *commute* issued the wrong verdict. The same choice (with *ND*) is reflected in the C component of the monitor's state as $C[x \mapsto \mathit{true}]$. Therefore, when the monitor eventually reaches this $\mathtt{r}(x)$, it halts because it does not have any transitions defined for this configuration.

- Concurrent: By definition, a cannot be ordered against any entity b outlined in the induction hypothesis. Let us assume that the monitor correctly guesses all the relevant components of the grain for a. If the monitor incorrectly decides that a is ordered wrt E, then it means that it does not commute with at least one grain signature in E. By induction hypothesis, this signature must belong to some entity (grain or event) that has already been observed in σ and is ordered wrt e. Therefore, by definition, a is also ordered wrt e.

  If the monitored correctly sees a as concurrent, then it does not update E and therefore maintains the invariants. If a is concurrent, but the monitor incorrectly sees it dependent on E, then this is an over-estimation of E, which can be dismissed. There exists another nondeterministic choice (as outlined above) that will get E right.

□

C.2 Proof of Theorem 5.2

Let L_{WF} be the set of words described by the regular expression in Equation (WF). Conside the set:

$$L_{\text{concurrent-marked-grains}} = \{w \in L_{\mathsf{WF}} \mid \text{ The two focal events demarcated by } \diamond_1 \text{ and } \diamond_2 \text{ are concurrent under the marked grains in } w\}.$$

From Theorem 5.1, we have that $L_{\text{concurrent-marked-grains}}$ is a regular set.

We will now focus on the set of words of the form $L_{\text{WF-unmarked-grains}} = \Sigma^* \diamond_1 \Sigma^* \diamond_2 \Sigma^*$ and the following subset of it:

$$L_{\text{concurrent-unmarked-grains}} = \{w \in L_{\text{WF-unmarked-grains}} \mid \text{ The two focal events demarcated by } \diamond_1 \text{ and } \diamond_2 \text{ are concurrent under any choice of grains in } w\}.$$

Observe that the set $L_{\text{concurrent-unmarked-grains}}$ is obtained by projecting the language $L_{\text{concurrent-unmarked-grains}}$ from the alphabet $\Sigma \uplus \{\diamond_1, \diamond_2, \rhd, \lhd\}$ to the sub-alphabet $\Sigma \uplus \{\diamond_1, \diamond_2\}$. Since $L_{\text{concurrent-marked-grains}}$ is regular, and since regular languages are closed under projection, we immediately get that $L_{\text{concurrent-unmarked-grains}}$ is also regular and hence there is a constant space monitor that recognizes it.

D PROOFS OF SECTION 6

D.1 Proof of Theorem 6.1

Proof. The high level idea is to try to linearize the graph $\mathbb{G}_{w,G}$, with the addition of an extra directed edge from the node whose grain contains e_2 to the one whose node contains e_1. Since there is no path from e_1 to e_2 in the graph, the addition of the edge cannot put the pair of events in the same strongly connected component. Moreover, the addition of the edge ensures that in every linearization of the graph e_2 appears before e_1.

The claim is that the step-by-step linearization succeeds and produces an rf-equivalent run.

At each step, let σ be the linearization so far and let R be the set of residual vertices left in the graph. Let $\mathbb{G}|_R$ be the graph induced on the vertices in R through the edges of $\mathbb{G}$.

Induction Hypothesis:

(i) σ contains everything in a maximal strongly connected component completely, or not at all, never partially. As such, the same is true about each grain, which is the smallest strongly connected component in the absence of a larger one.

(ii) Every event from w is either in σ or in some grain in R.

(iii) σ preserves po and rf of w.

(iv) There does not exist an edge between a node in R to a note containing any element in σ in graph $\mathbb{G}$.

Base case: $\sigma = \epsilon$, and all invariants hold.

Induction step: depending on the composition of R, we extend σ and reprove the induction hypothesis. Consider $\mathbb{G}|_R$, and the condensation of $\widehat{\mathbb{G}}|_R$ in which every edge in every maximally strongly connected component has been contracted. Observe that $\widehat{\mathbb{G}}|_R$ is acyclic. Therefore, there exists a node in it with no predecessors. Let us call this node v:

- Case 1: v is not a contracted node, and it corresponds to a node in $\mathbb{G}|_R$. We let $\sigma = \sigma v$, with the understanding that v is a word that represents the corresponding grain; the grain can correspond to a single letter and the word can be that letter. Remove the node v from $\mathbb{G}|_R$ (and all its adjacent edges).

  Let us reprove the induction hypothesis:

  (i) By definition of $\mathbb{G}|_R$, v is in a strongly connected component of size 1, and therefore this assumption holds again.

  (ii) Trivially true, because we just shift events from one side to the other.

  (iii) Any predecessors (through $\mathsf{po} \cup \mathsf{rf}$) of the events in v would have a path to v. If v is a minimal element, that means that all predecessors should be in σ or in v . Inside the word in v, we do not change the order of any events, and therefore po and rf cannot be broken inside v. By induction hypothesis, inside σ, we have maintained po and rf. Therefore, it remains to argue that po and rf are not broken when we concatenate the two.

  If po is broken, this means that some element in σ must have been po ordered after some element in p. But, this is a contradiction to induction hypothesis (iv).

It remains to argue that any dangling reads in v (i.e. reads whose matching writes do not belong to v) are matched with the correct write as a result of concatenating σ and v. Assume that is not the case; dangling read $\mathtt{r}(x)$ is matched with $\mathtt{w}_1(x)$ in σv, while it was matched with $\mathtt{w}_2(x)$ in w. We argued that $\mathtt{w}_2(x)$ is in σ. Therefore, it must be the case that $\mathtt{w}_2(x)$ appears before $\mathtt{w}_1(x)$ in σ. There are two possibilities for the original arrangement of the three events in w:

∗ $\mathtt{w}_1(x) \dots \mathtt{w}_2(x) \dots \mathtt{r}(x)$: This means that as a result of our linearization of σ so far, we ended up reordering the two writes. This is only possible if their corresponding grains commute. Otherwise, there would be an edge between them that would prevent us from linearizing them in the new order. But, since the dangling $\mathtt{r}(x)$ is not part of the grain of $\mathtt{w}_2(x)$, by definition, it cannot commute against any other grain with which it shares the variable; specifically, not the grain that includes $\mathtt{w}_1(x)$. Contradiction!

∗ $\mathtt{w}_2(x) \dots \mathtt{r}(x) \dots \mathtt{w}_1(x)$: This means that as a result of our linearization of σ so far, we ended up reordering $\mathtt{r}(x)$ and $\mathtt{w}_1(x)$. This is only possible if their corresponding grains commute. Otherwise, there would be an edge between them that would prevent us from linearizing them in the new order. But, since the dangling $\mathtt{r}(x)$ is not part of the grain of the write operation that it reads from (i.e. $\mathtt{w}_2(x)$), by definition, it cannot soundly commute against another grain with which it shares x and the other grain writes to x; specifically, not the grain that includes $\mathtt{w}_1(x)$. Contradiction!

(iv) It is implied by the choice of v as a node with no predecessors in the remaining graph.

- Case 2: v is a contracted node, and therefore corresponds to a set of nodes V_v from $\mathbb{G}|_R$. We reference induction hypothesis (i) to state the fact that all nodes in V_v belong in $\mathbb{G}|_R$. Let u be the subsequence of w that includes precisely all the events from the grains in V_v. Let $\sigma = \sigma u$. Remove the node v from $\mathbb{G}|_R$.

  Let us reprove the induction hypothesis:

(i) By definition of $\mathbb{G}|_R$, this holds

(ii) Trivially true, because we just shift events from one side to the other.

(iii) Any predecessors (through $\mathsf{po} \cup \mathsf{rf}$) of the events in u would have a path to v. If v is a minimal element, then all such predecessors are either in σ or inside u.

Inside the word in u, we do not change the order of any events (compared to how they appear in w), and therefore po and rf cannot be broken inside u.

It remains to argue that po and rf are not broken when we concatenate σ and u. The argument for po is similar to the previous case.

We must argue that any dangling reads in u are matched to the correct write as a result of concatenating σ and u. Assume that is not the case; dangling read $\mathtt{r}(x)$ is matched with $\mathtt{w}_1(x)$ in σu, while it was matched with $\mathtt{w}_2(x)$ in w.

$\mathtt{w}_1(x)$ and $\mathtt{w}_2(x)$ must both be in σ, because we are not reordering anything in u and $\mathtt{r}(x)$ would not otherwise be a dangling read. Therefore, it must be the case that $\mathtt{w}_2(x)$ appears before $\mathtt{w}_1(x)$ in σ and neither belongs to u. There are two possibilities for the original arrangement of the three events in w:

- $\mathtt{w}_1(x)\ldots\mathtt{w}_2(x)\ldots\mathtt{r}(x)$: This means that as a result of our linearization of σ so far, we ended up reordering the two writes. This is only possible if their corresponding grains commute. Otherwise, there would be an edge between them that would prevent us from linearizing them in the new order. But, since the dangling $\mathtt{r}(x)$ is not part of the grain of $\mathtt{w}_2(x)$, by definition, it cannot commute against any other grain with which it shares the variable; specifically, not the grain that includes $\mathtt{w}_1(x)$. Contradiction!
- $\mathtt{w}_2(x)\ldots\mathtt{r}(x)\ldots\mathtt{w}_1(x)$: This means that as a result of our linearization of σ so far, we ended up reordering $\mathtt{r}(x)$ and $\mathtt{w}_1(x)$. This is only possible if their corresponding grains commute. Otherwise, there would be an edge between them that would prevent us from linearizing them in the new order. But, since the dangling $\mathtt{r}(x)$ is not part of the grain of the write operation that it reads from (i.e. $\mathtt{w}_2(x)$), by definition, it cannot soundly commute against another grain with which it shares x and the other grain writes to x; specifically, not the grain that includes $\mathtt{w}_1(x)$. Contradiction!

(iv) It is implied by the choice of v as a node with no predecessors in the remaining graph.

□

D.2 Proof of Theorem 6.2

Proof. Observe that $\mathbb{G}_{w,G}$ is an acyclic graph for a valid set of *contiguous* grains G, and as such the condensed graph $\widehat{\mathbb{G}}$ and $\mathbb{G}_{w,G}$ coincide. Therefore, the proof of the theorem says that any linearization of $\mathbb{G}_{w,G}$, in which the grains appear as contiguous subwords is rf-equivalent to the original run w. It remains to argue that any such linearization can be acquired through a sequence of swaps.

The graph $\mathbb{G}_{w,G}$ coincides with the partial order that represents the the equivalence class of w under the grain monoid induced by G. More precisely, $[H_G(w)]_{\widehat{I_G}}$ is an equivalence class in a classic partially commutative monoid, and therefore precisely coincides with all the linearization of the partial order induced on the elements of $H_G(w)$ by $\widehat{I_G}$. Observe that the edges of $\mathbb{G}_{w,G}$ coincide with the constraints in this partial order, by construction. Therefore, any linearization $u \in (\Sigma \cup \Sigma_G)^*$ of the graph, with the view of noting down every grain as a_g (its corresponding alphabet symbol in Σ_G) can be obtained from $H_G(w)$ through a sequence of swaps from $\widehat{I_G}$. Since every such swap has a corresponding swap at the level of Σ^*, one can mirror the same swap sequence and transform $H^-1_G(u)$ to w. □

E PROOFS FROM SECTION 7

E.1 Proof of Proposition ??

Proof Sketch. A straightforward induction on i can be used to establish that $g^{(i)}$ is minimal. □

E.2 Proof of Lemma 7.2

Proof. First, let's establish soundess of $\mathbb{I}_{G'}$ using the characterization of Theorem 4.2. Suppose on the contrary that $\mathbb{I}_{G'}$ is not sound. This means, by Theorem 4.2, we have a pair of grains $(g'_1, g'_2) \in \mathbb{I}_{G'}$ that violate one of the conditions of Theorem 4.2; without loss of generality, we assume that g'_1 appears before g'_2 in w. Let $g_1, g_2 \in$ be the larger grains in G such that $g'_i \in \mathsf{split}(g_i)$ for $i \in \{1, 2\}$. The violations witnessed by (g'_1, g'_2) can be one of the following.

- $g'_1 g'_2 \not\equiv_{\mathsf{rf}} g'_2 g'_1$. In this case, there must be a variable x and events $e = \mathsf{r}(x)$, $f = \mathsf{rf}_e$ and $f' = \mathsf{w}(x)$ ($f' \neq f$) such that one of the following holds: (a) $e \in g'_1$ but $f \notin g_1$ and $f' \in g'_2$, (b) $e \in g'_1$ but $f \in g_1 - g'_1$ and $f' \in g'_2$, (c) $e \in g'_2$ but $f \notin g_2$ and $f' \in g'_1$, or (d) $e \in g'_2$ but $f \in g_2 - g'_2$ and $f' \in g'_1$,. In cases (a) and (c), we have $g_1 g_2 \not\equiv_{\mathsf{rf}} g_2 g_2$, contradicting soundness of $\mathbb{I}_G$ In case (b) (resp. case (d)), we have that $g'_1 \notin \mathsf{split}(g_1)$ (resp. $g'_2 \notin \mathsf{split}(g_2)$) since the minimal grain containing f must also contain e, contradicting minimality of splity grains (Proposition **??**).
- There is a variable $x \in \mathsf{var}(g'_1) \cap \mathsf{var}(g'_2)$ such that $\mathsf{w} \in \mathsf{op}(g'_1, x) \cup \mathsf{op}(g'_2, x)$ and an event $e = \mathsf{r}(x)$ (with $f = \mathsf{rf}_e$) such that one of the following holds: (a) $f \in g'_1$ and $e \in g_1 - g'_1$, (b) $f \in g'_1$ and $e \notin g_1$, (c) $e \in g'_1$ and $f \notin g_1 - g'_1$, (d) $e \in g'_1$ and $f \notin g_1$, (e) $f \in g'_2$ and $e \in g_2 - g'_2$, (f) $f \in g'_2$ and $e \notin g_2$, (g) $e \in g'_2$ and $f \notin g_2 - g'_2$, or (h) $e \in g'_2$ and $f \notin g_2$. In cases (a), (c), (e) and (g), minimality of either g'_1 or g'_2 is violated. In cases (b), (d), (f) and (h), soundness of $\mathbb{I}_G$ is violated.

Let us now prove that any two events that are declared grain graph concurrent using G will also be declared so using G'. Towards this, let $\mathcal{G}_{w,G} = (V, E)$ and $\mathcal{G}_{w,G'} = (V', E')$ be the respective grain graphs. Observe that for any two grains $g'_1, g'_2 \in G'$, if $(g'_1, g'_2) \in E'$, then $(g'_1, g'_2) \notin \widehat{\mathbb{I}_{G'}}$ and there exist events $e_1 \in g'_1$ and $e_2 \in g'_2$ such that e_1 appears before e_2 and $(e_1, e_2) \notin \mathbb{I}_{\mathcal{M}}$. This means $(g'_1, g'_2) \notin \mathbb{I}_{G'}$ and thus $(g_1, g_2) \notin \mathbb{I}_G$, where g_1 and g_2 are the corresponding grains of G from which $g'_1 \in g_1$ and $g'_2 \in g_2$ are obtained after splitting. Since we have $e_1 \in g_1$ and $e_2 \in g_2$, we can conclude that $(g_1, g_2) \in E$. As a result, if there is a path $(h'_1, h'_2), (h'_2, h'_3) \dots (h'_k, h'_{k+1})$ in $\mathcal{G}_{w,G'}$, then the following is a path in $\mathcal{G}_{w,G}$: $(h_1, h_2), (h_2, h_3) \dots (h_k, h_{k+1})$, where $h'_i \in h_i$ is the corresponding larger grain. Hence, if e_1 and e_2 are declared grain graph concurrent by G, they will also be declared so using G' □

E.3 Proof of Lemma 7.1

Proof. Let π be a prefix. For a grain g that is active at the end of π, we let $\mathcal{X}_g$ be the set of variables x such that there is a read of x which is not in g, but its corresponding write event is in g. Observe that since each of the grains in G is minimal, we must have $\mathcal{X}_g \neq \emptyset$ for each $g \in \mathsf{Active}_{\pi,G}$. Now, consider two distinct grains $g, g' \in \mathsf{Active}_{\pi,G}$. We must have $\mathcal{X}_g \cap \mathcal{X}_{g'} = \emptyset$, as otherwise two different write events on the same variable will be pending in π which is impossible. This gives us $|\mathsf{Active}_{\pi,G}| \leq |\mathcal{X}|$. □

E.4 Proof of Proposition 7.1

($\Leftarrow$). It is easy to see that if there is a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{SG}_{\pi,G}$, then there must also be a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{G}_{\pi,G}$ and thus a paths in $\mathcal{G}_{w,G}$. This is because edges of $\mathcal{SG}_{\pi,G}$ are paths of $\mathcal{G}_{\pi,G}$.

($\Rightarrow$). Consider a path ρ from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{G}_{w,G}$. This path can be expressed as $\rho = \rho_1 \rho_2 \dots \rho_k$, where for each sub-path ρ_i, the start and the end vertices are vertices of $\mathcal{SG}_{\pi,G}$, and each of the intermediate vertices are not. In other words, $\mathsf{source}(\rho_i) \rightsquigarrow_{\pi,G} \mathsf{target}(\rho_i)$. As a result, $(\mathsf{source}(\rho_i), \mathsf{target}(\rho_i))$ is an edge in $\mathcal{SG}_{\pi,G}$. This gives a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{SG}_{w,G}$.

E.5 Proof of Theorem 7.1

We will prove this theorem by proving that after having processed prefix π of the input word w, the state q of the automaton $\mathcal{A}_{\mathrm{GG}}$ reflects the summarized graph $\mathcal{SG}_{\pi,G}$. Towards this, let us define the notation $\mathsf{proj}(\pi, G)$ to denote the finite projection of $\mathcal{SG}_{\pi,G} = (V_\pi, E_\pi)$, i.e., $\mathsf{proj}(\pi, G) = \langle V, E, C, P, SC, SP \rangle$ such that the following holds. In the following, we use $\mathsf{ID}(g)$ to denote the grain identifer of a grain g. Also, we use $g_{\pi,u}$ to denote the grain identifier of the latest grain in π with identifier u.

- $V = \{\mathsf{ID}(g) \mid g \in V_\pi\}$.
- $E = \{(\mathsf{ID}(g_1), \mathsf{ID}(g_2)) \mid (g_1, g_2) \in E_\pi\}$.
- For each $u \in V$, $\mathsf{C}(u) = \mathsf{C}_\pi(g_{u,\pi})$.
- For each $u \in V$, $\mathsf{P}(u) = \mathsf{P}_\pi(g_{u,\pi})$.
- For each $u \in V$, $\mathsf{SC}(u) = \mathsf{SC}_\pi(g_{u,\pi})$.
- For each $u \in V$, $\mathsf{SP}(u) = \mathsf{SP}_\pi(g_{u,\pi})$.

The invariant that the automaton maintains is the following:

Claim E.1. Let $w \in L_{\mathsf{VMG}}$ and let G be the corresponding grains. Let π be some prefix of w. Let q be the state of $\mathcal{A}_{\mathsf{GG}}$ after processing π. We have, $q = \mathsf{proj}(\pi, G)$.

Proof. We establish this by inducting on the length of π. In the base case, $\pi = \epsilon$ is the empty trace and thus $\mathcal{SG}_{\pi,G}$ is the empty graph and the corresponding state of the automaton reached, namely q_0 also matches $q_0 = \mathsf{proj}(\pi, G)$. Let us now consider the run $\pi = \rho \cdot e$, where $e \in \widehat{\Sigma}$ and let q_ρ be the state of $\mathcal{A}_{\mathsf{GG}}$ after having processed ρ. By the inductive hypothesis, we have $q_\rho = \mathsf{proj}(\rho, G)$ We can now establish the invariant about $q_\pi = \delta_{\mathsf{GG}}(q_\rho, e)$ by doing a case-by-case analysis on e. In the following we will use the notation $q_\rho = \langle V_\rho, E_\rho, \mathsf{C}_\rho, \mathsf{P}_\rho, \mathsf{SC}_\rho, \mathsf{SP}_\pi \rangle$.

Case $e = (i, (\triangleright, Y))$. Since w belongs to L_{VMG}, we know that at the end of ρ, there is no active grain with identifier i, and hence $i \notin V$. Clearly, $\mathcal{SG}_{\pi,G}.V = \mathcal{SG}_{\rho,G}.V \uplus \{i\}$. Also, no new edges must be added in $\mathcal{SG}_{\pi,G}.E$ over $\mathcal{SG}_{\rho,G}.E$. Likewise, since the only new event added is the begin event of a new transaction, the contents of each tracked grain stays the same. The pending variables of the grain with ID i is captured by the set Y. Finally, since there is no path in $\mathcal{G}_{\pi,G}$ that originate at i, neither the summaries nor the pending vars of reachable grains change. Thus, $q_\pi = \mathsf{proj}(\pi, G)$.

Case $e = (i, \triangleleft)$, $i \in \{\diamond_1, \diamond_2\}$. No change in the summarized graph happens and this is also reflected in q_π which is the same as q_ρ.

Case $e = (i, \triangleleft)$, $i \notin \{\diamond_1, \diamond_2\}$. In this case, the grain with identifier i is not present in $\mathcal{SG}_{\pi,G}$ since it is no longer active. Indeed, we have $i \notin V_\pi = V_\rho - \{i\}$, since i is a dead grain at the end of π. For the same reason, $\mathsf{P}_\pi(i) = \varnothing$ and $\mathsf{C}_\pi(i) = \varnothing$. Now, the set of dead paths are as follows: $g \rightsquigarrow_{\pi,G} g'$ iff $g_{i,\pi} \notin g, g'$ and either (a) $g \rightsquigarrow_{\rho,G} g'$ or (b) ($g \rightsquigarrow_{\rho,G} g_{i,\pi}$ and $g_{i,\pi} \rightsquigarrow_{\rho,G} g'$). Thus, $\mathcal{SG}_{\pi,G}.E = (\mathcal{SG}_{\rho,G}.E - \{(g_{i,\pi}, g), (g, g_{i,\pi}) \mid g \neq g_{i,\pi}\}) \cup \{(g, g') \mid (g, g_{i,\pi}) \in \mathcal{SG}_{\rho,G}.E, (g_{i,\pi}, g') \in \mathcal{SG}_{\rho,G}.E\}$. Indeed, the function $mrg(E, i)$ in the monitor captures this accurately. Finally, the summaries of a grain g in the summarized graph change based on the new dead paths. In particular, for a grain g, the set of grains that are reachable from g using a dead path are either those that were already reachable in ρ, or those that were reachable from $g_{i,\pi}$, given $g \rightsquigarrow_{\rho,G} g_{i,\pi}$. This means that $\mathcal{SG}_{\pi,G}.\mathsf{SC}(g) = \mathcal{SG}_{\rho,G}.\mathsf{SC}(g) \cup \mathcal{SG}_{\rho,G}.\mathsf{C}(g_{\pi,i}) \cup \mathcal{SG}_{\rho,G}.\mathsf{SC}(g_{\pi,i})$ if $(g, g_{i,\pi}) \in \mathcal{SG}_{\rho,G}.E$, and $\mathcal{SG}_{\pi,G}.\mathsf{SC}(g) = \mathcal{SG}_{\rho,G}.\mathsf{SC}(g)$ for other active grains. Indeed this is accurately reflected in SC_π using $mrgSm(SM, M, E, i)$. The reasoning for SP_π follows similar reasoning.

Case $e = (i, a)$, $a \in \Sigma$. In this case, no new active grain is seen. However, new edges may be formed between active grains. First note that no new edge between grains g and g' can be added at this point if $g_{i,\pi} \notin \{g, g'\}$ since no new dead paths can be formed at this point. Now consider a grain g in $\mathcal{SG}_{\rho,G}.V = \mathcal{SG}_{\pi,G}.V$. An edge from g to $g_{i,\pi}$ may be inferred if a new dead path $g \rightsquigarrow_{\pi,G} g_{i,\pi}$ may be inferred. This can happen if there is an existing dead path $g \rightsquigarrow_{\rho,G} g'$ for some g' (active or dead) such that there is an event $e \in g'$ (on variable x) that is dependent with a and x is either pending in g' or in $g_{i,\pi}$. If g' is dead, then inductively we have $e \in \mathcal{SG}_{\rho,G}.\mathsf{SC}(g)$ and $x \in \mathcal{SG}_{\rho,G}.\mathsf{SP}(g)$. Otherwise we would have $g' = g$ and thus inductively $e \in \mathcal{SG}_{\rho,G}.\mathsf{C}(g)$ and $x \in \mathcal{SG}_{\rho,G}.\mathsf{P}(g)$. This is captured using $Dep(\mathsf{C}(j) \cup \mathsf{SC}(j),\ a,\ \mathsf{P}(j) \cup \mathsf{SP}(j) \cup \mathsf{P}(i))$ in the monitor.

□

As a corollary of Claim E.1, it is easy to see that for a run in $w \in L_{\mathsf{VMG}}$ annotated with grains G, if w is accepted, then there is a path from $g^{\diamond_1}, g^{\diamond_2}$ in $\mathcal{SG}_{w,G}$, and further if $g^{\diamond_1}, g^{\diamond_2}$ in $\mathcal{SG}_{w,G}$, then the automaton reaches a state $q \in F_{\mathsf{GG}}$ where there is a path from node $\diamond_1$ to node $\diamond_2$. This proves the theorem.

E.6 Proof of Theorem 7.2

First, we outline the language L_{VMG} and show that it is regular. Observe that $L_{\mathsf{VMG}} = L_{\mathsf{valid}} \cap L_{\mathsf{minimal}}$ where L_{valid} is the set of annotated runs with correctly marked grains and pending variables and L_{minimal}. We will show that both of these languages are regular, and thus their intersection is regular.

First consider the language L_{valid}; it can expressed as $L_{\mathsf{valid}} = L_{\mathsf{valid\text{-}grains}} \cap L_{\diamond}$, where the first language is the collection of all words such that for every grain identifier, if we project the word to that grain identifier, then no two grains overlap, while the second language is the set of all words that contain exactly one occurrence of the begin event corresponding to each of $\{\diamond_1, \diamond_2\}$. For a grain identifier $i \in \mathsf{gIDs}$, let us use L_{i} to denote the set of words that correspond to the contents of a valid grain whose identifier is i. Then, $L_{\mathsf{grains,\ i}} = (\sum_{Y \subseteq X} (i, (\triangleright, Y)) \cdot L_{\mathsf{i,\ Y}} (i, \triangleleft))^*$ is the set of words that correspond to valid sequences of grain i. Then, consider the homomorphism π_i that projects all letters whose grain id is i to themselves, and all other letters to ϵ. Then, $L_{\mathsf{valid\text{-}grains\text{-}non\text{-}pending}} = \bigcap\limits_{i \in \mathsf{gIDs}} (\pi_i^{-1} L_{\mathsf{grains,\ i}})$ is the set of words whose projection to any given grain ID is well formed; this is obtained by intersecting the inverse homomorphic image of regular languages, hence it is regular. Now, a simple automaton can also check if the pending variables are consistent with the annotation; let L_{pending} be the langauge of this automaton. Hence, $L_{\mathsf{valid\text{-}grains}} = L_{\mathsf{valid\text{-}grains\text{-}non\text{-}pending}} \cap L_{\mathsf{pending}}$ and thus regular. Finally, $L_{\diamond}$ is the set of all words that contain exactly one occurence of a letter from $\{(\diamond_1, (\triangleright, Y)) \mid Y \subseteq X\}$ and one occurence of a letter from $\{(\diamond_2, (\triangleright, Y)) \mid Y \subseteq X\}$; clearly this is regular.

Now, we consider the set of runs where the grains are minimal. Minimality can also be checked using an automaton which tracks, for each active grain, whether there is at least one pending read not yet seen, by guessing this read and later validating it (in a left to right pass). Thus, L_{VMG} is regular.

Now, the set of runs in which two given events (marked with $\diamond_1$ and $\diamond_2$) are deemed concurrent are essentially those that are obtained by the homomorphic image of the words in L_{VMG}, ensure that these two events are in the focal grains, and are also accepted by $\mathcal{A}_{\mathsf{GG}}$. Here, the homomorphism we consider maps begin and end letters to ϵ, and for other letters it removes their grain identifiers. This is clearly a regular language.