Dynamic Data-Race Detection through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, USA

Umang Mathur
University of Illinois at Urbana-Champaign, USA

Andreas Pavlogiannis
Aarhus University, Denmark

Abstract

Data races are among the most common bugs in concurrency. The standard approach to data-race detection is via dynamic analyses, which work over executions of concurrent programs, instead of the program source code. The rich literature on the topic has created various notions of dynamic data races, which are known to be detected efficiently when certain parameters (e.g., number of threads) are small. However, the *fine-grained* complexity of all these notions of races has remained elusive, making it impossible to characterize their trade-offs between precision and efficiency.

In this work we establish several fine-grained separations between many popular notions of dynamic data races. The input is an execution trace σ with $\mathcal{N}$ events, $\mathcal{T}$ threads and $\mathcal{L}$ locks. Our main results are as follows. First, we show that *happens-before (HB) races* can be detected in $O(\mathcal{N} \cdot \min(\mathcal{T}, \mathcal{L}))$ time, improving over the standard $O(\mathcal{N} \cdot \mathcal{T})$ bound when $\mathcal{L} = o(\mathcal{T})$. Moreover, we show that even reporting an HB race that involves a read access is hard for 2-orthogonal vectors (2-OV). This is the first rigorous proof of the conjectured quadratic lower-bound in detecting HB races. Second, we show that the recently introduced *synchronization-preserving races* are hard to detect for OV-3 and thus have a cubic lower bound, when $\mathcal{T} = \Omega(\mathcal{N})$. This establishes a complexity separation from HB races which are known to be less expressive. Third, we show that *lock-cover races* are hard for 2-OV, and thus have a quadratic lower-bound, even when $\mathcal{T} = 2$ and $\mathcal{L} = \omega(\log \mathcal{N})$. The similar notion of *lock-set races* is known to be detectable in $O(\mathcal{N} \cdot \mathcal{L})$ time, and thus we achieve a complexity separation between the two. Moreover, we show that lock-set races become hitting-set (HS)-hard when $\mathcal{L} = \Theta(\mathcal{N})$, and thus also have a quadratic lower bound, when the input is sufficiently complex. To our knowledge, this is the first work that characterizes the complexity of well-established dynamic race-detection techniques, allowing for a rigorous comparison between them.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases dynamic analyses, data races, fine-grained complexity

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Concurrent programs that communicate over shared memory are prone to *data races*. Two events are *conflicting* if they access the same memory location and one (at least) modifies that location. Data races occur when conflicting access happen concurrently between different threads, and form one of the most common bugs in concurrency. In particular, data races are often symptomatic of bugs in software like data corruption [5, 20, 26], and they have been deemed *pure evil* [6] due to the problems they have caused in the past [43]. Moreover, many compiler optimizations are unsound in the presence of data races [36, 40], while data-race freeness is often a requirement for assigning well-defined semantics to programs [7].

The importance of data races in concurrency has led to a multitude of techniques for detecting them efficiently [4, 39]. By far the most standard approach is via *dynamic analyses*. Instead of analyzing the full program, dynamic analyzers try to *predict* the existence of data races by observing and analyzing concurrent executions [37, 21, 28]. As full dynamic data race prediction is NP-hard in general [24], researchers have developed several approximate notions of dynamic races, accompanied by efficient techniques for detecting each notion.

Happens-before races. The most common technique for detecting data races dynamically is based on Lamport's *happens-before (HB)* partial order [22]. Two conflicting events form an HB race if they are unordered by HB, as the lack of ordering between them indicates the fact that they may execute concurrently, thereby forming a data race. The standard approach to HB race detection is via the use of vector clocks [19], and has seen wide success in commercial race detectors [35]. As vector clock computation is known to require $\Theta(\mathcal{N} \cdot \mathcal{T})$ time on traces of $\mathcal{N}$ events and $\mathcal{T}$ threads [10], HB race detection is often assumed to suffer the same bound, and has thus been a subject of further practical optimizations [29, 16].

Synchronization preserving races. HB races were recently generalized to sync(hronization)-preserving races [25]. Intuitively, two conflicting events are in a sync-preserving race if the observed trace can be soundly reordered to a witness trace in which the two events are concurrent, but without reordering synchronization events (e.g., locking events). Similar to HB races, sync-preserving races can be detected in linear time when the number of threads is constant. However, the dependence on the number of threads is cubic for sync-preserving races, as opposed to the linear dependence for HB races. On the other hand, sync-preserving races are known to offer better precision in program analysis.

Races based on the locking discipline. The locking discipline dictates that threads that access a common memory location must do so inside *critical sections*, using a common lock, when performing the access [39]. Although this discipline is typically not enforced, it is considered good practice, and hence instances that violate this principle are often considered indicators of erroneous behavior. For this reason, there have been two popular notions of data races based on the locking discipline, namely *lock-cover races* [14] and *lock-set races* [33]. Both notions are detectable in linear time when the number of locks is constant, however, lock-set race detection is typically faster in practice, which also comes at the cost of being less precise.

Observe that, although techniques for all aforementioned notions of races are generally thought to operate in linear time, they only do so assuming certain parameters, such as the number of threads, are constant. However, as these techniques are deployed in runtime, often with extremely long execution traces, they have to be as efficient as absolutely possible, often in scenarios when these parameters are very large. When a data-race detection technique is too slow for a given application, the developers face a dilemma: do they look for a faster algorithm, or for a simpler abstraction (i.e., a different notion of dynamic races)? For these reasons, it is important to understand the *fine-grained* complexity of the problem at hand with respect to such parameters. Fine-grained lower bounds can rule out the possibility of faster algorithms, and thus help the developers focus on new abstractions that are more tractable for the given application. Motivated by such questions, in this work we settle the fine-grained complexity of dynamically detecting several popular notions of data races.

1.1 Our Contributions

Here we give a full account of the main results of this work, while we refer to later sections for precise definitions and proofs. We also refer to Appendix A for relevant notions in fine-grained complexity and popular hypotheses. The input is always a concurrent trace σ of length $\mathcal{N}$, consisting of $\mathcal{T}$ threads, $\mathcal{L}$ locks, and $\mathcal{V}$ variables.

Happens-before races. We first study the fine-grained complexity of HB races, as they form the most popular class of dynamic data races. The task of most techniques is to report all events in σ that participate in an HB race, which is known to take $O(\mathcal{N}\cdot\mathcal{T})$ time [19]. Note that the bound is quadratic when $\mathcal{T}=\Theta(\mathcal{N})$, and multiple heuristics have been developed to address it in practice (see e.g., [16]). Our first result shows that polynomial improvements below this quadratic bound are unlikely.

▶ **Theorem 1.** *For any $\epsilon>0$, there is no algorithm that detects even a single HB race that involves a read in time $O(\mathcal{N}^{2-\epsilon})$, unless the OV hypothesis fails.*

Orthogonal vectors (OV) is a well-studied problem with a long-standing quadratic worst-case upper bound. The associated hypothesis states that there is no sub-quadratic algorithm for the problem [42]. It is also known that the strong exponential time hypothesis (SETH) implies the Orthogonal Vectors hypothesis [41]. Thus, under the OV hypothesis, Theorem 1 establishes a quadratic lower bound for HB race detection.

Note that the hardness of Theorem 1 arises out of the requirement to detect HB races that involve a read. A natural follow-up question is whether detecting if the input contains *any* HB race (i.e., not necessarily involving a read) has a similar lower bound based on SETH. Our next theorem shows that under the non-deterministic SETH (NSETH) [9], there is no fine-grained reduction from SETH that proves any lower bound for this problem above $\mathcal{N}^{3/2}$.

▶ **Theorem 2.** *For any $\epsilon>0$, there is no $(2^{\mathcal{N}},\mathcal{N}^{3/2+\epsilon})$-fine-grained reduction from SAT to the problem of detecting any HB race with bound, unless NSETH fails.*

Given the impossibility of Theorem 2, it would be desirable to at least show a super-linear lower bound for detecting any HB data race. To tackle this question, we show that detecting any HB race is hard for the general problem of model checking first-order formulas quantified by $\forall\exists\exists$ on structures of size n with m relational tuples (denoted FO($\forall\exists\exists$)).

▶ **Theorem 3.** *For any $\epsilon>0$, if there is an algorithm for detecting any HB race in time $O(\mathcal{N}^{1+\epsilon})$, then there is an algorithm for FO($\forall\exists\exists$) formulas in time $O(m^{1+\epsilon})$.*

It is known that FO($\forall\exists\exists$) can be solved in $O(m^{3/2})$ time [17], which yields a bound $O(n^3)$ for dense structures (i.e., when $m=\Theta(n^2)$). Theorem 3 implies that if $m^{3/2}$ is the best possible bound for FO($\forall\exists\exists$), then detecting any HB race cannot take $O(\mathcal{N}^{1+\epsilon})$ time for any $\epsilon<1/2$. Although improvements for FO($\forall\exists\exists$) over the current $O(m^{3/2})$ bound might be possible, we find that a truly linear bound $O(m)$ would require major breakthroughs [1]. Under this hypothesis, Theorem 3 implies a super-linear bound for HB races.

Finally, we give an improved upper bound for this problem when $\mathcal{L}=o(\mathcal{T})$.

▶ **Theorem 4.** *Deciding whether σ has an HB race can be done in time $O(\mathcal{N}\cdot\min(\mathcal{T},\mathcal{L}))$.*

In fact, similar to existing techniques [16], the algorithm behind Theorem 4 detects *all* variables that participate in an HB race (instead of just reporting σ as racy).

Synchronization-preserving races. Next, we turn our attention to the recently introduced

[1] Even the well-studied problem of testing triangle freeness, which is a special case of the similarly flavored FO($\exists\exists\exists$), has the super-linear bound $O(n^{\omega})$.

sync-preserving races [24]. It is known that detecting sync-preserving races takes $O(\mathcal{N}\cdot\mathcal{V}\cdot\mathcal{T}^3)$ time. As sync-preserving races are known to be more expressive than HB races, the natural question is whether sync-preserving races can be detected more efficiently, e.g., by an algorithm that achieves a bound similar to Theorem 4 for HB races. Our next theorem answers this question in negative.

▶ **Theorem 5.** *For any $\epsilon > 0$, there is no algorithm that detects even a single sync-preserving race in time $O(\mathcal{N}^{3-\epsilon})$, unless the 3-OV hypothesis fails. Moreover, the statement holds even for traces over a single variable.*

As HB races take at most quadratic time, Theorem 5 shows that the increased expressiveness of sync-preserving races incurs a complexity overhead that is unavoidable in general.

Races based on the locking discipline. We now turn our attention to data races based on the locking discipline, namely *lock-cover races* and *lock-set races*. It is known that lock-cover races are more expressive than lock-set races. On the other hand, existing algorithms run in $O(\mathcal{N}^2\cdot\mathcal{L})$ time for lock-cover races and in $O(\mathcal{N}\cdot\mathcal{L})$ time for lock-set races, and thus hint that the former are computationally harder to detect. Our first theorem makes this separation formal, by showing that even with just two threads, having slightly more that logarithmically many locks implies a quadratic hardness for lock-cover races.

▶ **Theorem 6.** *For any $\epsilon > 0$, any $\mathcal{T}\geq 2$ and any $\mathcal{L}=\omega(\log\mathcal{N})$, there is no algorithm that detects even a single lock-cover race in time $O(\mathcal{N}^{2-\epsilon})$, unless the OV hypothesis fails.*

Observe that the $O(\mathcal{N}\cdot\mathcal{L})$ bound for lock-set races also becomes quadratic, when the number of locks is unbounded (i.e., $\mathcal{L}=\Theta(\mathcal{N})$). Is there a SETH-based quadratic lower bound similar to Theorem 6 for this case? Our next theorem rules out this possibility, again under NSETH.

▶ **Theorem 7.** *For any $\epsilon > 0$, there is no $(2^{\mathcal{N}}, \mathcal{N}^{1+\epsilon})$-fine-grained reduction from SAT to the problem of detecting any lock-set race, unless NSETH fails.*

Hence, even though we desire a quadratic lower bound, Theorem 7 rules out any super-linear lower-bound based on SETH. Alas, our next theorem shows that a quadratic lower bound for lock-set races does exist, based on the hardness of the hitting set (HS) problem.

▶ **Theorem 8.** *For any $\epsilon > 0$ and any $\mathcal{T}=\omega(\log n)$, there is no algorithm that detects even a single lock-cover race in time $O(\mathcal{N}^{2-\epsilon})$, unless the HS hypothesis fails.*

Hitting set is a problem similar to OV, but has different quantifier structure. Just like the OV hypothesis, the HS hypothesis states that there is no sub-quadratic algorithm for the problem [3]. Although HS implies OV, the opposite is not known, and thus Theorem 8 does not contradict Theorem 7. In conclusion, we have that both lock-cover and lock-set races have (conditional) quadratic lower bounds, though the latter is based on a stronger hypothesis (HS), and requires more threads and locks for hardness to arise.

Finally, on our way to Theorem 7, we obtain the following theorem.

▶ **Theorem 9.** *Deciding whether a trace σ has a lock-set race on a variable x can be performed in $O(\mathcal{N})$ time. Thus, deciding whether σ has a lock-set race can be performed in $O(\mathcal{N}\cdot\min(\mathcal{L},\mathcal{V}))$ time.*

Hence, Theorem 9 strengthens the $O(\mathcal{N}\cdot\mathcal{L})$ upper bound for lock-set races when $\mathcal{V}=o(\mathcal{L})$.

1.2 Related Work

Dynamic data-race detection. There exists a rich literature in dynamic techniques for data race detection. Methods based on vector clocks (DJIT algorithm [19]) using Lamport's Happens Before (HB) [22] and the lock-set principle in Eraser [33] were the first ones to popularize dynamic analysis for detecting data races. Later work attempted to increase the performance of these notions using optimizations as in [29] and FastTrack [16], altogether different algorithms (e.g., the GoldiLocks algorithm [15]), and hybrid techniques [27]. HB and lock-set based race detection are respectively sound (but incomplete) and complete (but unsound) variants of the more general problem of data-race *prediction* [34]. While earlier work on data race prediction focused on explicit [34] or symbolic [31, 32] enumeration, recent efforts have focused on scalability [37, 23, 21, 28, 30, 38]. The more recent notion [25] of sync-preserving races generalizes the notion of HB. As the complexity of race prediction is prohibitive (NP-hard in general [24]), this work characterizes the fine-grained complexity of popular, more relaxed notions of dynamic races that take polynomial time.

Fine-grained complexity. Traditional complexity theory usually shows a problem is intractable by proving it NP-hard, and tractable by showing it is in P. For algorithms with large input sizes, this distinction may be too coarse. It becomes important to understand, even for problems in P, whether algorithms with smaller degree polynomials than the known are possible, or if there are fine-grained lower bounds making this unlikely. Fine-grained complexity involves proving such lower bounds, by showing relationships between problems in P, with an emphasis on the degree of the complexity polynomial, and is nowadays a field of very active study. We refer to [8] for an introductory, and to [42] for a more extensive exposition on the topic. Fine-grained arguments have also been instrumental in characterizing the complexity of various problems in concurrency, such as bounded context-switching [11], safety verification [12], data-race prediction [24] and consistency checking [13].

2 Preliminaries

2.1 Concurrent Program Executions and Data Races

Traces and Events. We consider execution traces (or simply *traces*) generated by concurrent programs, under the sequential consistency memory model. Under this memory model, a trace σ is a sequence of events. Each event e is labeled with a tuple $\mathsf{lab}(e) = \langle t, op\rangle$, where t is the (unique) identifier of the thread that performs the event e, and op is the operation performed in e. We will often abuse notation and write $e = \langle t, op\rangle$ instead of $\mathsf{lab}(e) = \langle t, op\rangle$. For the purpose of this presentation, an operation can be one of (a) read ($\mathtt{r}(x)$) from, or write ($\mathtt{w}(x)$) to, a shared memory variable x, (b) $\mathtt{acq}(\ell)$ or $\mathtt{rel}(\ell)$ of a lock ℓ.

For an event $e = \langle t, op\rangle$, we use $\mathsf{tid}(e)$ and $\mathrm{op}(e)$ to denote respectively the thread identifier t and the operation op. For a trace σ, we use Events_σ to denote the set of events that appear in σ. Similarly, we will use $\mathsf{Threads}_\sigma$, Locks_σ and Vars_σ to denote respectively the set of threads, locks and shared variables that appear in trace σ. We denote by $\mathcal{N} = |\mathsf{Events}_\sigma|$, $\mathcal{T} = |\mathsf{Threads}_\sigma|$, $\mathcal{L} = |\mathsf{Locks}_\sigma|$, and $\mathcal{V} = |\mathsf{Vars}_\sigma|$. The set of read events and write events on variable $x \in \mathsf{Vars}_\sigma$ will be denoted by $\mathsf{Reads}_\sigma(x)$ and $\mathsf{Writes}_\sigma(x)$, and further we let $\mathsf{Accesses}_\sigma(x) = \mathsf{Reads}_\sigma(x) \cup \mathsf{Writes}_\sigma(x)$. Similarly, we let $\mathsf{Acquires}_\sigma(\ell)$ and $\mathsf{Releases}_\sigma(\ell)$ denote the set of lock-acquire and lock-release events, respectively, of σ on lock ℓ. The *trace order* of σ, denoted $\leq^\sigma_{\mathsf{tr}}$, is the total order on Events_σ induced by the sequence σ. Finally, the *thread-order* of σ, denoted $\leq^\sigma_{\mathsf{TO}}$ is the smallest partial order on Events_σ such that for any two events $e_1, e_2 \in \mathsf{Events}_\sigma$, if $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ and $\mathsf{tid}(e_1) = \mathsf{tid}(e_2)$, then $e_1 \leq^\sigma_{\mathsf{TO}} e_2$.

Traces are assumed to be well-formed in that critical sections on the same lock do not overlap. For a lock $\ell \in \mathsf{Locks}_\sigma$, let $\sigma|_\ell$ be the projection of the trace σ on the set of events $\{e \,|\, \mathrm{op}(e) \in \{\mathtt{acq}(\ell), \mathtt{rel}(\ell)\}\}$. Also, let $t_1, \dots t_k$ be the thread identifiers in $\mathsf{Threads}_\sigma$. Well-formedness then entails that for each lock ℓ, the projection $\sigma|_\ell$ is a prefix of some string in the language of the grammar with production rules $S \to \varepsilon | S \cdot S_{t_1} | S \cdot S_{t_2} | \cdots | S \cdot S_{t_k}$ and

$S_{t_i} \to \langle t_i, \texttt{acq}(\ell)\rangle \cdot \langle t_i, \texttt{rel}(\ell)\rangle$ and start symbol S. Thus, every release event e has a unique matching acquire event, which we denote by $\mathsf{match}_\sigma(e)$. Likewise for an acquire event e, $\mathsf{match}_\sigma(e)$ denotes the unique matching release event if one exists. For an acquire event e, the critical section of e is the set of events $\mathsf{CS}_\sigma(e) = \{f \mid e \leq^\sigma_{\mathsf{TO}} f \leq^\sigma_{\mathsf{TO}} \mathsf{match}_\sigma(e)\}$ if $\mathsf{match}_\sigma(e)$ exists, and $\mathsf{CS}_\sigma(e) = \{f \mid e \leq^\sigma_{\mathsf{TO}} f\}$ otherwise.

Data Races. Two events $e_1, e_2 \in \mathsf{Events}_\sigma$ are said to be *conflicting* if they are performed by different threads, they are access events touching the same memory location, and at least one of them is a write access. Formally, we have (i) $\mathsf{tid}(e_1) \neq \mathsf{tid}(e_2)$, (ii) $e_1, e_2 \in \mathsf{Accesses}_\sigma(x)$ for some $x \in \mathsf{Vars}_\sigma$, and (iii) $\{e_1, e_2\} \cap \mathsf{Writes}_\sigma(x) \neq \varnothing$. An event $e \in \mathsf{Events}_\sigma$ is said to be *enabled* in a prefix ρ of σ, if for every event $e' \neq e$ with $e' \leq^\sigma_{\mathsf{TO}} e$, we have $e' \in \mathsf{Events}_\rho$. A data race in σ is a pair of conflicting events (e_1, e_2) such that there is a prefix ρ in which both e_1 and e_2 are simultaneously enabled. Depending on the type of access of e_1 and e_2, we often distinguish between write-write races and write-read races.

2.2 Notions of Dynamic Data Races

As the problem of determining whether a concurrent program has an execution with a data race is undecidable, dynamic techniques observe program traces and report whether certain events indicate the presence of a race. Depending on the technique, such reports can be sound (i.e., they guarantee the presence of a race in the program), Here we describe in detail some popular approaches to dynamic race detection that are the subject of this work.

Happens-Before Races. Given a trace σ, the *happens before* order $\leq^\sigma_{\mathsf{HB}}$ is the smallest partial order on Events_σ such that (a) $\leq^\sigma_{\mathsf{TO}} \subseteq \leq^\sigma_{\mathsf{HB}}$, and (b) for any lock $\ell \in \mathsf{Locks}_\sigma$ and for events $e \in \mathsf{Acquires}_\sigma(\ell)$ and $f \in \mathsf{Releases}_\sigma(\ell)$, if $e \leq^\sigma_{\mathsf{tr}} f$ then $e \leq^\sigma_{\mathsf{HB}} f$. A pair of conflicting events (e_1, e_2) is an HB-race in σ if they are unordered by HB, i.e., $e_1 \not\leq^\sigma_{\mathsf{HB}} e_2$ and $e_2 \not\leq^\sigma_{\mathsf{HB}} e_1$. The associated decision question is, *given a trace σ, determine whether σ has an HB race.* Typically HB race detectors are tasked to report all events that form HB race with an earlier event in the trace [35, 2, 1]). That is, they solve the following function problem:*given a trace σ, determine all events $e_2 \in \mathsf{Events}_\sigma$ for which there exists an event $e_1 \in \mathsf{Events}_\sigma$ such that $e_1 \leq^\sigma_{tr} e_2$, and (e_1, e_2) is an HB race of σ.* The standard algorithm for solving both versions of the problem is a vector-clock algorithm that runs in $O(\mathcal{N} \cdot \mathcal{T})$ time [19].

Synchronization Preserving Races. Next, we present the notion of *sync(hronization)-preserving races* [24]. For a trace σ and a read event e, we use $\mathsf{lw}_\sigma(e)$ to denote the write event observed by e. That is, $e' = \mathsf{lw}_\sigma(e)$ is the last (according to the trace order $\leq^\sigma_{\mathsf{tr}}$) write event e' of σ such that e and e' access the same variable and $e' \leq^\sigma_{\mathsf{tr}} e$; if no such e' exists, then we write $\mathsf{lw}_\sigma(e) = \bot$. A trace ρ is said to be a correct reordering of trace σ, if (a) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$ (b) Events_ρ is downward closed with respect to $\leq^\sigma_{\mathsf{TO}}$, and further $\leq^\rho_{\mathsf{TO}} \subseteq \leq^\sigma_{\mathsf{TO}}$, and (c) for every read event $e \in \mathsf{Events}_\rho$, $\mathsf{lw}_\rho(e) = \mathsf{lw}_\sigma(e)$. We say that ρ is *sync-preserving* with respect to σ if for every lock ℓ and for any two acquire events $e_1, e_2 \in \mathsf{Acquires}_\rho(\ell)$, we have $e_1 \leq^\rho_{\mathsf{tr}} e_2$ iff $e_1 \leq^\sigma_{\mathsf{tr}} e_2$. That is, the order of two critical sections on the same lock is the same in σ and ρ.

A pair of conflicting events (e_1, e_2) is a *sync-preserving race* in σ if σ has a sync-preserving correct reordering ρ such that (e_1, e_2) is a data race of ρ. The associated decision question is, *given a trace σ, determine whether σ has a sync-preserving race.* As with HB races, we are typically interested in reporting all events $e_2 \in \mathsf{Events}_\sigma$ for which there exists an event $e_1 \in \mathsf{Events}_\sigma$ such that $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, and (e_1, e_2) is an HB race of σ. It is known one can report all such events e_2 in time $O(\mathcal{N} \cdot \mathcal{V} \cdot \mathcal{T}^3)$.

Lock-Cover and Lock-Set Races. Lock-cover and lock-set races indicate violations

	t_1	t_2
1	acq(ℓ)	
2	w(x)	
3	rel(ℓ)	
4		w(x)
5		acq(ℓ)
6		rel(ℓ)

(a) HB-race

	t_1	t_2
1	w(x)	
2	acq(ℓ)	
3	rel(ℓ)	
4		acq(ℓ)
5		rel(ℓ)
6		w(x)

(b) Sync-preserving race

	t_1	t_2
1	acq(ℓ)	
2	w(x)	
3	rel(ℓ)	
4		acq(ℓ)
5		r(x)
6		rel(ℓ)
7		w(x)

(c) Lock-cover race

	t_1	t_2
1	acq(ℓ_1)	
2	acq(ℓ_2)	
3	w(x)	
4	rel(ℓ_2)	
5	rel(ℓ_1)	
6		acq(ℓ_2)
7		acq(ℓ_3)
8		w(x)
9		rel(ℓ_3)
10		rel(ℓ_2)
11	acq(ℓ_1)	
12	acq(ℓ_3)	
13	w(x)	
14	rel(ℓ_3)	
15	rel(ℓ_1)	

(d) Lockset race

Figure 1 Types of data races.

of the *locking discipline*. For an event e in a trace σ, let $\mathsf{locksHeld}_\sigma(e) = \{\ell \mid \exists f \in \mathsf{Acquires}_\sigma(\ell)$, such that $e \in \mathsf{CS}_\sigma(f)\}$, i.e., $\mathsf{locksHeld}_\sigma(e)$ is the set of locks held by thread $\mathsf{tid}(e)$ when e is executed. A pair of conflicting events might indicate a data race if $\mathsf{locksHeld}_\sigma(e_1) \cap \mathsf{locksHeld}_\sigma(e_2) = \varnothing$. Although this condition does not guarantee the presence of a race, it constitutes a violation of the locking discipline and can be further investigated.

A pair of conflicting events (e_1, e_2) is a *lock-cover race* if $\mathsf{locksHeld}_\sigma(e_1) \cap \mathsf{locksHeld}_\sigma(e_2) = \varnothing$. The decision question is, *given a trace σ, determine if σ has a lock-cover race.* The problem is solvable in $O(\mathcal{N}^2 \cdot \mathcal{L})$ time, by checking the above condition over all conflicting event pairs.

As the algorithm for lock-cover races takes quadratic time, developers often look for less expensive indications of violations of locking discipline, called lock-set races (as proposed by ERASER race detector [33]). A trace σ has a *lock-set race* on variable $x \in \mathsf{Vars}_\sigma$ if

(a) there exists a pair of conflicting events $(e_1, e_2) \in \mathsf{Writes}_\sigma(x) \times \mathsf{Accesses}_\sigma(x)$, and
(b) $\bigcap_{e \in \mathsf{Accesses}_\sigma(x)} \mathsf{locksHeld}_\sigma(e) = \varnothing$.

The associated decision question is, *given a trace σ, determine if σ has a lock-set race.* Note that a lock-cover race implies a lock-set race, but not the other way around. On the other hand, determining whether σ has a lock-set race is easily performed in $O(\mathcal{N} \cdot \mathcal{L})$ time.

Example. We illustrate the different notions of races in Figure 1. We use e_i to denote the i^{th} event of the trace in consideration. First consider the trace σ_a in Figure 1a. The events e_2 and e_4 are conflicting and unordered by $\leq_{\mathsf{HB}}^{\sigma_a}$, thus (e_2, e_4) is an HB-race. Second, in trace σ_b of Figure 1b, the pair (e_1, e_6) is not an HB-race as $e_1 \leq_{\mathsf{HB}}^{\sigma_b} e_6$. But this is a sync-preserving race witnessed by the correct reordering e_4, e_5, as both e_1 and e_6 are enabled. Third, in trace σ_c of Figure 1c, the pair (e_2, e_7) is neither a sync-preserving race nor an HB race, but is a lock-cover race as $\mathsf{locksHeld}_{\sigma_c}(e_2) \cap \mathsf{locksHeld}_{\sigma_c}(e_7) = \varnothing$. Finally, the trace σ_d in Figure 1d has no HB, sync-preserving or lock-cover race, as all $\mathtt{w}(x)$ are protected by a common lock. But there is a lock-set race on x as there is no single lock that protects all $\mathtt{w}(x)$.

3 Happens-Before Races

In this section we prove the results for detecting HB races, i.e., Theorem 1 to Theorem 4.

3.1 Algorithm for HB Races

In this section, we outline our $O(\mathcal{N}\cdot\mathcal{L})$-time algorithm for checking if a trace σ has an HB-race, thereby proving Theorem 4. As with the standard vector clock algorithm [19], our algorithm is based on computing timestamps for each event. However, unlike the standard algorithm that assigns thread-indexed timestamps, we use *lock-indexed* timestamps, or *lockstamps*, which we formalize next. We fix the input trace σ in the rest of the discussion.

Lockstamps A lockstamp is a mapping from locks to natural numbers (including infinity) $L : \mathsf{Locks}_\sigma \to \mathbb{N} \cup \{\infty\}$. Given lockstamps L, L_1, L_2 and lock ℓ, we use the notation (i) $L[\ell \mapsto c]$ to denote the the lockstamp $\lambda m\cdot$ if $m = \ell$ then c else $L(m)$, (ii) $L_1 \sqcup L_2$ to denote the pointwise maximum, i.e., $(L_1 \sqcup L_2)(\ell) = \max(L_1(\ell), L_2(\ell))$ for every ℓ, (iii) $L_1 \sqcap L_2$ to denote the pointwise minimum, and (iv) $L_1 \sqsubseteq L_2$ to denote the predicate $\forall \ell\cdot L_1(\ell) \leq L_2(\ell)$.

Our algorithm computes *acquire* and *release* lockstamps AcqLS^σ_e and RelLS^σ_e for every event $e \in \mathsf{Events}_\sigma$. Let us formalize these next. For a lock ℓ and acquire event $f \in \mathsf{Acquires}_\sigma(\ell)$ (resp. release event $g \in \mathsf{Releases}_\sigma(\ell)$), let $pos_\sigma(f) = |\{f' \in \mathsf{Acquires}_\sigma(\ell) \mid f' \leq^\sigma_{\mathsf{tr}} f\}|$ (resp. $pos_\sigma(g) = |\{f' \in \mathsf{Releases}_\sigma(\ell) \mid f' \leq^\sigma_{\mathsf{tr}} f\}|$) denote the relative position of f (resp. g) among all acquire events (resp. release events) of ℓ. Then, for an event $e \in \mathsf{Events}_\sigma$ the lockstamps AcqLS^σ_e and RelLS^σ_e are defined as follows (we assume that $\max \varnothing = 0$ and $\min \varnothing = \infty$.)

$$\begin{aligned} \mathsf{AcqLS}^\sigma_e(\ell) &= \lambda\ell \cdot \max\{pos_\sigma(f) \mid f \in \mathsf{Acquires}_\sigma(\ell), f \leq^\sigma_{\mathsf{HB}} e\} \\ \mathsf{RelLS}^\sigma_e(\ell) &= \lambda\ell \cdot \min\{pos_\sigma(f) \mid f \in \mathsf{Releases}_\sigma(\ell), e \leq^\sigma_{\mathsf{HB}} f\} \end{aligned} \tag{1}$$

Our $O(\mathcal{N}\cdot\mathcal{L})$ algorithm now relies on the following observations. First, the HB partial order can be inferred using by comparing lockstamps of events (Lemma 10). Second, there is an $O(\mathcal{N}\cdot\mathcal{L})$ time algorithm that computes the acquire and release lockstamps for each event in the input trace. Third, the existence of an HB race can be determined by examining only $O(\mathcal{N})$ pairs of conflicting events (using their lockstamps), instead of all possible $O(\mathcal{N}^2)$ pairs (Lemma 11). Finally, we can also examine all the $O(\mathcal{N})$ pairs in time $O(\mathcal{N}\cdot\mathcal{L})$ (using $O(\mathcal{N})$ lockstamp comparisons) and thus determine the existence of an HB race in the same asymptotic running time. Let us first state how we use lockstamps to infer the HB relation.

► **Lemma 10.** *Let $e_1 \leq^\sigma_{tr} e_2$ be events in σ such that $\mathsf{tid}(e_1) \neq \mathsf{tid}(e_2)$. We have, $e_1 \leq^\sigma_{HB} e_2 \iff \neg(\mathsf{AcqLS}^\sigma_{e_2} \sqsubseteq \mathsf{RelLS}^\sigma_{e_1})$*

Computing Lockstamps. We now illustrate how to compute the acquire lockstamps for all events, by processing the trace σ in a forward pass. For each thread t and lock ℓ, we maintain lockstamp variables $\mathbb{C}_t$ and $\mathbb{L}_\ell$. We also maintain an integer variable p_ℓ for each lock ℓ that stores the index of the latest $\mathtt{acq}(\ell)$ event in σ. Initially, we set each $\mathbb{C}_t$ and $\mathbb{L}_m$ to the *bottom* map $\lambda\ell \cdot 0$, and p_m to 0, for each thread t and lock m. We traverse σ left to right, and perform updates to the data structures as described in Algorithm 1, by invoking the appropriate *handler* based on the thread and operation of the current event $e = \langle t, op\rangle$. At the end of each handler, we assign the lockstamp AcqLS^σ_e to e. The computation of release lockstamps is similar, albeit in a reverse pass, and presented in Appendix B.1. Observe that each step takes $O(\mathcal{L})$ time giving us a total running time of $O(\mathcal{N}\cdot\mathcal{L})$ to assign lockstamps.

We say that a pair of conflicting access events (e_1, e_2) (with $e_1 \leq^\sigma_{\mathsf{tr}} e_2$) to a variable x is a *consecutive conflicting pair* if there is no event $f \in \mathsf{Writes}_\sigma(x)$ such that $e_1 <^\sigma_{\mathsf{tr}} f <^\sigma_{\mathsf{tr}} e_2$. We make the following observation.

► **Lemma 11.** *A trace σ has an HB-race iff there is pair of consecutive conflicting events in σ that is an HB-race. Moreover, σ has $O(\mathcal{N})$ many consecutive conflicting pairs of events.*

Checking for an HB race. We now describe the algorithm for checking for an HB race in

Algorithm 1 *Assigning acquire lockstamps to events in the trace*

```
acquire(t, ℓ):
    p_ℓ ← p_ℓ + 1
    ℂ_t ← ℂ_t[ℓ ↦ p_ℓ] ⊔ 𝕃_ℓ
    AcqLS^σ_e ← ℂ_t

release(t, ℓ):
    𝕃_ℓ ← ℂ_t
    AcqLS^σ_e ← ℂ_t

read(t, x):
    AcqLS^σ_e ← ℂ_t

write(t, x):
    AcqLS^σ_e ← ℂ_t
```

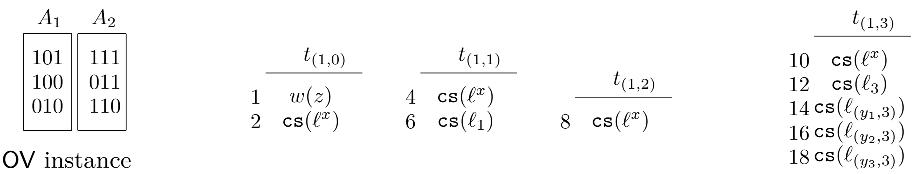

Figure 2 Reducing OV to detecting a write-read HB-races. Illustration of the threads $t(x,i)$, where x is the first vector of A_1. $\texttt{cs}(\ell)$ denotes the sequence $\texttt{acq}(\ell), \texttt{rel}(\ell)$. Event numbers indicate the relative order in which these threads execute in σ.

σ. We perform a forward pass on σ while storing the release lockstamps of some of the earlier events. When processing an access event e, we check if it is in race with an earlier eventby comparing the acquire lockstamp of e with a previously stored release lockstamp. More precisely, we maintain a variable $\mathbb{W}_x$ to store the release lockstamp of the last write event on x, a variable t^w_x to store the thread that performed this write and set S_x to store pairs (t, L) of threads and release lockstamps of all the read events performed since the last write on x was observed. Initially, $t^w_x = \texttt{NIL}$, $\mathbb{W}_x = \lambda\ell \cdot \infty$ and $\mathsf{S}_x = \varnothing$. The update performed at each event $e = \langle t, op \rangle$ are presented in the corresponding handler in Algorithm 2.

Algorithm 2 *Determining the existence of an HB-race using lockstamps*

```
read(t, x):
    if t^w_x ∉ {NIL, t} ∧ AcqLS^σ_e ⊑ 𝕎_x then
        declare 'race' and exit
    S_x ← S_x ∪ {(t, RelLS^σ_e)}

write(t, x):
    if t^w_x ∉ {NIL, t} ∧ AcqLS^σ_e ⊑ 𝕎_x then
        declare 'race' and exit
    if ∃(u, L) ∈ S_x, t ≠ u ∧ AcqLS^σ_e ⊑ L then
        declare 'race' and exit
    t^w_x = t; S_x ← ∅; 𝕎_x ← RelLS^σ_e
```

We refer to Appendix B.1 for the correctness, which concludes the proof of Theorem 4.

3.2 Hardness Results for HB

We now turn our attention to the hardness results for HB race detection. To this end, we prove Theorem 1, Theorem 2, and Theorem 3. We start with defining the graph $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$, which can be thought of as a form of transitive reduction of the HB relation.

The graph $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$. Given a trace σ, the graph $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$ is a graph with node set Events_σ, and we have an edge (e_1, e_2) in $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$ iff (i) e_2 is the immediate successor of e_1 wrt the thread order $\leq^\sigma_{\mathsf{TO}}$, or (ii) e_1 is a $\texttt{rel}(\ell)$ event, e_2 is a $\texttt{acq}(\ell)$ event, $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, and there is no intermediate event in σ that accesses lock ℓ. It follows easily that for any two distinct events e_1, e_2, we have $e_1 \leq^\sigma_{\mathsf{HB}} e_2$ iff e_2 is reachable from e_1 in $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$. Moreover, every node has out-degree ≤ 2 and thus $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$ is sparse, while it can be easily constructed in $O(\mathcal{N})$ time.

OV hardness of write-read HB races. Given a OV instance OV(n,d) on two vector sets A_1, A_2, we create a trace σ as follows. For the part A_1 of OV, we introduce $n \cdot (d+1)$ threads denoted by $t(x,i)$, for $x \in [n], i \in \{0\} \cup [d]$, and d locks, each denoted by l_i, for $i \in [d]$. For

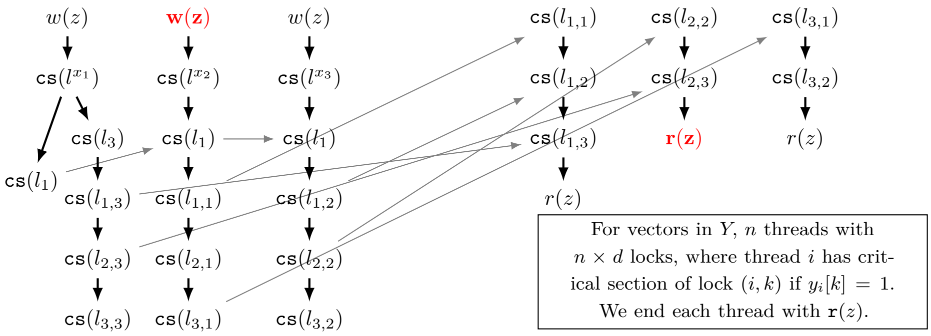

Figure 3 Reducing OV to finding HB races using the instance of Figure 2. For simplicity, we show the graph $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$ instead of the trace σ. The HB race is marked in red, corresponding to the orthogonal pair (x_2, y_2).

the second part A_2 we introduce $n \cdot d$ locks denoted by $l(y,i)$, for $y \in [n], i \in [d]$, and n threads, denoted by t_y, for $y \in [n]$. Finally, we have a single variable z.

We first describe the threads $t(x,i)$. We order the vectors in A_1 arbitrarily. For each vector x, for each $i \in [d]$, we introduce a critical section on the lock l_i. If x is the last vector of A_1 with $x[i] = 1$, we also insert the critical sections $l_{(y,i)}$ for all $y \in [n]$, to $t(x,i)$ after the critical section of l_x. Finally, we construct a thread $t_{x,0}$ which starts with a write event $\mathtt{w}(z)$, followed by a critical section on lock l^x. We also insert a critical section on lock l^x to all threads $t(x,i)$, for $i \in [d]$. Hence the $\mathtt{w}(z)$ event is ordered by HB before all other events of $t(x,i)$. See Figure 2 for an illustration.

Now we describe the threads t_y. For each $i \in [d]$, if $y[i] = 1$, we add a critical section of the lock $l(y,i)$ in t_y. We end the thread with a read event $\mathtt{r}(z)$.

Finally, we construct σ by first executing each thread $t(x,i)$ in the pre-determined order of $x \in A_1$, followed by executing the traces t_y in any order. See Figure 3 for an illustration. We refer to Appendix B for the correctness, which concludes the proof of Theorem 1.

We now turn our attention to the problem of detecting a single HB race (i.e., not necessarily involving a read event). We define a useful multi-connectivity problem on graphs.

▶ **Problem 1.** *[MCONN] Given a directed graph G with n nodes and m edges, and k pairs of nodes $(s_i, t_i), i \in [k]$, decide if there is a path in G from every s_i to the corresponding t_i.*

Due to Lemma 11, detecting whether there is an HB race in σ reduces to testing MCONN between all $O(\mathcal{N})$ pairs of consecutive conflicting events in σ.

Short witnesses for HB races. We now prove Theorem 2. Following [9, Corollary 2], it suffices to show that deciding MCONN can be done in $\mathsf{NTIME}[\mathcal{N}^{3/2}] \cap \mathsf{coNTIME}[\mathcal{N}^{3/2}]$. At a first glance, the bound $\mathsf{NTIME}[\mathcal{N}^{3/2}]$ may seem too optimistic, as there are $\Theta(\mathcal{N})$ paths $P_i \colon s_i \rightsquigarrow t_i$, and each of them can have size $\Theta(\mathcal{N})$. Hence even just guessing these paths appears to take quadratic time. Our proof shows that more succinct witnesses exist.

Proof of Theorem 2. First consider the simpler case where σ has an HB-race. Phrased as a MCONN problem on $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$, it suffices to show that there is a pair (s_i, t_i) such that s_i does not reach t_i. We construct a non-deterministic algorithm for this task that simply guesses the pair (s_i, t_i), and verifies that there is no $s_i \rightsquigarrow t_i$ path. Since $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$ is sparse, this can be easily verified in $O(\mathcal{N})$ time.

Now consider the case when there is no HB-race. Phrased as a MCONN problem on $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$, it suffices to verify that for every pair (s_i, t_i), we have that s_i reaches t_i. We construct a non-deterministic algorithm for this task, as follows. The algorithm operates in two phases, using a set A, initialized as $A = \{(s_i, t_i)\}_{i \in k}$.

1. In the first phase, the algorithm repeatedly guesses a node u that lies on at least $\mathcal{N}^{1/2}$ paths $s_i \rightsquigarrow t_i$, for $(s_i, t_i) \in A$. It verifies this guess via a backward and a forward traversal from u. The algorithm then removes all such (s_i, t_i) from A, and repeats.
2. In the second phase, the algorithm guesses for every remaining $(s_i, t_i) \in A$ a path $P_i : s_i \rightsquigarrow t_i$, and verifies that P_i is a valid path.

Phase 1 can be execute at most $\mathcal{N}^{1/2}$ iterations, while each iteration takes $O(\mathcal{N})$ time since $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$ is sparse. Hence the total time for phase 1 is $O(\mathcal{N}^{3/2})$. Phase 2 takes $O(\mathcal{N}^{3/2})$ time, as every node of $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$ appears in at most $\mathcal{N}^{1/2}$ paths P_i. The desired result follows. ◀

A super-linear lower bound for general HB races. Finally, we turn our attention to Theorem 3. The problem $\mathsf{FO}(\forall\exists\exists)$ takes as input a first-order formula ϕ with quantifier structure $\forall\exists\exists$ and whose atoms are tuples, and the task is to verify whether ϕ has a model on a structure of n elements and m relational tuples. For simplicity, we can think of the structure as a graph G of n nodes and m edges, and ϕ a formula that characterizes the presence/absence of edges (e.g., $\phi = \forall x \exists y \exists z\ e(x, y) \wedge \neg e(y, z)$).

The crux of the proof of Theorem 3 is showing the following lemma.

▶ **Lemma 12.** *$\mathsf{FO}(\forall\exists\exists)$ reduces to MCONN on a graph G with $O(n)$ nodes in $O(n^2)$ time.*

Finally, we arrive at Theorem 3 by constructing in $O(n^2)$ time a trace σ with $\mathcal{N} = \Theta(n^2)$ such that $\mathsf{G}(\leq^{\sigma}_{\mathsf{HB}})$ is similar in structure to the graph G of Lemma 12. In the end, detecting an HB race in σ in $O(\mathcal{N}^{1+\epsilon})$ time yields an algorithm for $\mathsf{FO}(\forall\exists\exists)$ in $\mathcal{N} = \Theta(n^{2+\epsilon})$ time. We refer to Appendix B for the details, which conclude the proof of Theorem 3.

4 Synchronization-Preserving Races

In this section we discuss the dynamic detection of sync-preserving races, and prove Theorem 5.

For notational convenience, we will frequently use the composite *sync* events. A $\mathtt{sync}(\ell)$ event represents the sequence $\mathtt{acq}(\ell), \mathtt{r}(x_\ell), \mathtt{w}(x_\ell), \mathtt{rel}(\ell)$. The key idea behind sync events is as follows. Assume that in a trace σ we have two $\mathtt{sync}(\ell)$ events e_1 and e_2 with $e_1 <^{\sigma}_{\mathsf{tr}} e_2$. Then any correct reordering ρ of σ with $e_2 \in \mathsf{Events}_\rho$ satisfies the following.

(a) We have $e_1 \in \mathsf{Events}_\rho$, as the read event of e_2 must read from the write event of e_1.
(b) For every $e'_1, e'_2 \in \mathsf{Events}_\rho$ such that $e'_1 \leq^{\sigma}_{\mathsf{TO}} e_1$ and $e_2 \leq^{\sigma}_{\mathsf{TO}} e'_2$, we have $e_1 <^{\rho}_{\mathsf{tr}} e_2$.

We hence use sync events to ensure certain orderings in any sync-preserving correct reordering of σ that exposes a sync-preserving data race.

Intuition. Before we proceed with the detailed reduction, we provide a high-level description. The input to 3-OV is three sets of vectors $A_1 = \{x_i\}_{i \in [n]}$, $A_2 = \{y_i\}_{i \in [n]}$, and $A_3 = \{z_i\}_{i \in [n]}$. Every vector $x \in A_1$ is represented by a thread t_x, ending with the critical section $\mathtt{acq}(X), \mathtt{w}(z), \mathtt{rel}(X)$. Similarly, every vector $y \in A_1$ is represented by a thread t_y, ending with the critical section $\mathtt{acq}(Y), \mathtt{r}(z), \mathtt{rel}(Y)$. Notice that we can only have a race between the write event of a thread t_x and the read event of a thread t_y. The search for such a race corresponds to the search of the corresponding vectors $x \in A_1$ and $y \in A_2$ such that there is a vector $z \in A_3$ which makes the triplet x, y, z orthogonal.

To establish this correspondence, we insert in t_x empty critical sections on locks l_k, for $k \in [d]$ that represent the coordinates k for which $x[k] = 1$. We use a similar encoding with locks l'_k for the threads t_y, capturing that $y[k] = 1$. To encode the vectors in A_3, we use k threads t_k, for $k \in [d]$, such that the i^{th} segment of t_k encodes $z_i[k]$: we have two interleaved critical

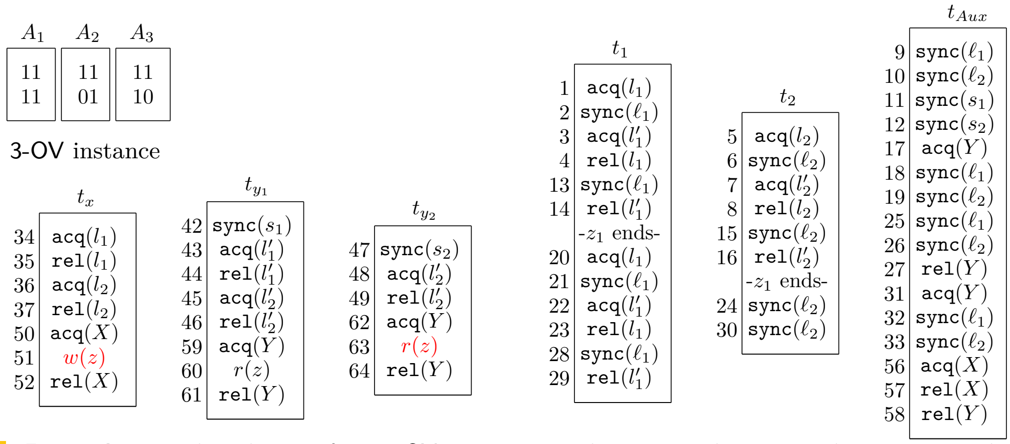

Figure 4 Example reduction from 3-OV to `sync` race detection. The trace orders events as shown by their numbering. We only show one thread t_x, as the two x vectors are identical.

sections on locks l_k and l'_k iff $z_i[k] = 1$.

Finally, we use some `sync` events to force all threads t_k be partially executed whenever we want to execute the write event of any thread t_x. Hence, any correct reordering of σ that exposes a data race in σ, must execute all t_k at least partially. We make all threads t_k execute before all t_x and t_y in σ. The notion of sync-preservation ensures that if we have a correct reordering that exposes a race between two threads t_x and t_y, then the following holds. For every coordinate $k \in [d]$ in which $x[k] = y[k] = 1$, since the corresponding threads t_x and t_y have critical sections on locks l_k and l'_k, the thread t_k must execute up to a point where it does not have critical sections on these locks. This means that we have found a vector z with $z[k] = 0$, and thus the triplet x, y, z is orthogonal on that coordinate.

Reduction. Given an 3-OV instance OV(n, d, 3) on vector sets $A_1 = \{x_i\}_{i\in[n]}$, $A_2 = \{y_i\}_{i\in[n]}$, and $A_3 = \{z_i\}_{i\in[n]}$, we create a trace σ as follows (see Figure 4). We have $\mathcal{T} = 2 \cdot n + d + 1$ threads, while all access events (not counting the sync events) are of the form $\mathtt{w}(z)/\mathtt{r}(z)$ in a single variable z. We first describe the threads, and then how they interleave in σ.

Threads. We introduce a thread t_x for every vector $x \in A_1$ and a lock l_k for every $k \in [d]$. Each thread t_x consists of two segments t_x^1 and t_x^2. We create t_x^1 as follows. For every $k \in [d]$ where $x[k] = 1$, we add an empty critical section $\mathtt{acq}(l_k), \mathtt{rel}(l_k)$ in t_x^1. We create t_x^2 as the sequence $\mathtt{acq}(X), \mathtt{w}(z), \mathtt{rel}(X)$, where X is a new lock, common for all t_x^2.

For the vectors in A_2, we introduce threads similar to those of part A_1, as follows. We have a thread t_y for every vector $y \in A_2$ and a lock l'_k for every $k \in [d]$. Each thread t_y consists of two segments t_y^1 and t_y^2. For every $k \in [d]$ where $y[k] = 1$, we add an empty critical section $\mathtt{acq}(l'_k), \mathtt{rel}(l'_k)$ in t_y^1. In contrast to the t_x^1, every t_y^1 also has an event $\mathtt{sync}(s_y)$ at the very beginning. We create t_y^2 as the sequence $\mathtt{acq}(Y), \mathtt{r}(z), \mathtt{rel}(Y)$, where Y is a new lock, common for all t_y^2.

The construction of the threads corresponding to the vectors in A_3 is more involved. We have one thread t_k for every $k \in [d]$. Each thread has some fixed `sync` events, as well as critical sections corresponding to one coordinate of all n vectors in A_3. In particular, we construct each t_k as follows. We iterate over all z_i, and if $z_i[k] = 0$, we simply append two events $\mathtt{sync}(\ell_k), \mathtt{sync}(\ell_k)$ to t_k. On the other hand, if $z_i[k]$, we interleave these sync events with two critical sections, by appending the sequence $\mathtt{acq}(l_k), \mathtt{sync}(\ell_k), \mathtt{acq}(l'_k), \mathtt{rel}(l_k), \mathtt{sync}(\ell_k), \mathtt{rel}(l'_k)$.

Lastly, we have a single auxiliary trace t that consists of three parts t^1, t^2 and t^3, where

$$t^1 = \texttt{sync}(\ell_1), \dots, \texttt{sync}(\ell_k), \texttt{sync}(s_{y_1}), \dots \texttt{sync}(s_{y_n})$$
$$t^2 = (\texttt{acq}(Y), \texttt{sync}(\ell_1), \dots, \texttt{sync}(\ell_k), \texttt{sync}(\ell_1), \dots, \texttt{sync}(\ell_k), \texttt{rel}(Y))^{n-1}$$
$$t^3 = \texttt{acq}(Y), \texttt{sync}(\ell_1), \dots, \texttt{sync}(\ell_k), \texttt{acq}(X), \texttt{rel}(X), \texttt{rel}(Y)$$

Concurrent trace. We are now ready to describe the interleaving of the above threads in order to obtain the concurrent trace σ.

1. We execute the auxiliary trace t and all traces t_k, for $k \in [d]$ (i.e., the threads corresponding to the vectors of A_3) arbitrarily, as long as for every $k \in [d]$, every sequence of $\texttt{sync}(\ell_k)$ events (a) starts with the $\texttt{sync}(\ell_k)$ event of t_k and proceeds with the $\texttt{sync}(\ell_k)$ event of t, (b) strictly alternates in every two $\texttt{sync}(\ell_k)$ events between t and t_k, and (c) ends with the last $\texttt{sync}(\ell_k)$ event of t_k.
2. We execute all t^1_x and t^1_y (i.e., the first parts of all threads that correspond to the vectors in A_1 and A_2) arbitrarily, but after all traces t_k, for $k \in [d]$.
3. We execute all t^2_x (i.e., the second parts of all traces that correspond to the vectors in A_1) arbitrarily, but before the segment $\texttt{acq}(X), \texttt{rel}(X), \texttt{rel}(Y)$ of t.
4. We execute all t^2_y (i.e., the second parts of all traces that correspond to the vectors in A_2) arbitrarily, but after the segment $\texttt{acq}(X), \texttt{rel}(X), \texttt{rel}(Y)$ of t.

We refer to Appendix C for the correctness of the reduction and thus the proof of Theorem 5.

5 Violations of the Locking Discipline

5.1 Lock-Cover Races

We start with a simple reduction from OV to detecting lock-cover races. Given a OV instance OV(n,d) on two vector sets A_1, A_2, we create a trace σ as follows. We have a single variable x and two threads t_1, t_2. We associate with each vector of the set A_i a write access event $e = \langle t_i, \texttt{w}(x) \rangle$. Moreover, each such event holds up to d locks, so that e holds the k^{th} lock iff k^{th} coordinate of the vector corresponding to the event is 1. The trace σ is formed by ordering the sequence of events corresponding to vectors of A_1 of OV first, in a fixed arbitrary order, followed by the sequence of events corresponding to A_2, again in arbitrary order. We refer to Appendix D for the correctness, which concludes the proof of Theorem 6.

5.2 Lock-Set Races

We now turn our attention to lock-set races. We first prove Theorem 9, i.e., that determining whether a trace σ has a lock-set race on a specific variable x can be performed in linear time.

A linear-time algorithm per variable. Verifying that there are two conflicting events on x is straightforward by a single pass of σ. The more involved part is in computing the lock-set of x, i.e., the set $\bigcap_{e \in \mathsf{Accesses}_\sigma(x)} \mathsf{locksHeld}_\sigma(e)$, in linear time. Indeed, each intersection alone requires $\Theta(\mathcal{L})$ time, resulting to $\Theta(\mathcal{N} \cdot \mathcal{L})$ time overall.

Here we show that a somewhat more involved algorithm achieves the task. The algorithm performs a single pass of σ, while maintaining three simple sets A, B, and C. While processing an event e, the sets are updated to maintain the invariant

$$A = \mathsf{locksHeld}_\sigma(e) \quad B = \mathsf{Locks}_\sigma \cap \bigcap_{e' \in \mathsf{Accesses}_\sigma(x), e' \leq^\sigma_{\mathsf{tr}} e} \mathsf{locksHeld}_\sigma(e') \quad C = \overline{A} \cap B \qquad (2)$$

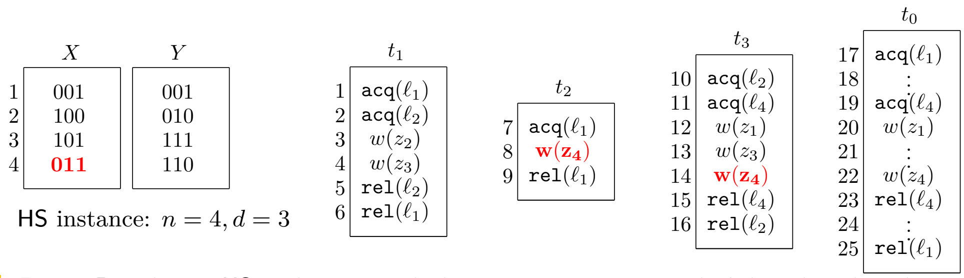

Figure 5 Reducing HS to detecting a lock-set race on trace σ with d threads. Thread t_k uses lock l_i if $y_i[k] = 0$, and $w(z_j)$ if $x_j[k] = 1$. Vector x_4 hits all vectors in Y, implying a lock-set race on z_4.

The sets are initialized as $A = \varnothing$, $B = C = \mathsf{Locks}_\sigma$. Then the algorithm performs a pass over σ and processes each event e according to the description of Algorithm 3.

Algorithm 3 *Computing lock-set of variable x*

```
acquire(t, ℓ):
    A ← A ∪ {ℓ}
    if ℓ ∈ B then
        C ← C \ {ℓ}
release(t, ℓ):
    A ← A \ {ℓ}
    if ℓ ∈ B then
        C ← C ∪ {ℓ}
read(t, y):
    if x = y then
        B ← B \ C
        C ← ∅
write(t, y):
    if x = y then
        B ← B \ C
        C ← ∅
```

The correctness of Algorithm 3 follows by proving the invariant in Equation (2). We refer to Appendix D for the details, which concludes the proof of Theorem 9.

Short witnesses for lock-set races. Besides the advantage of a faster algorithm, Theorem 9 implies that lock-set races have short witnesses that can be verified in linear time. This allows us to prove that detecting a lock-set race is in $\mathsf{NTIME}[\mathcal{N}] \cap \mathsf{coNTIME}[\mathcal{N}]$, and we can thus use [9, Corollary 2] to prove Theorem 7.

Proof of Theorem 7. First we argue that the problem is in $\mathsf{NTIME}[\mathsf{n}]$. Indeed, the certificate for the existence of a lock-set race is simply the variable x on which there is a lock-set race. By Theorem 9, verifying that we indeed have a lock-set race on x takes $O(\mathcal{N})$ time.

Now we argue that the problem is in $\mathsf{coNTIME}[\mathsf{n}]$, by giving a certificate to verify in linear time that σ does not have a race of the required form. The certificate has size $O(|\mathsf{Vars}_\sigma|)$, and specifies for every variable, either the lock that is held by all access events of the variable, or a claim that there exist no two conflicting events on that variable. The certificate can be easily verified by one pass over σ. ◀

Lock-set races are Hitting-Set hard. Finally we prove Theorem 8, i.e., that determining a single lock-set race is HS-hard, and thus also carries a conditional quadratic lower bound. We establish a fine-grained reduction from HS. Given a HS instance HS(n,d) on two vector sets X, Y, we create a trace σ using $d+1$ threads $\{t_j\}_{j \in \{0\} \cup [d]}$, n locks $\{\ell_i\}_{i \in [n]}$, and n variables $\{z_i\}_{k \in [n]}$. Thread t_0 that executes $\mathtt{acq}(\ell_1), \dots, \mathtt{acq}(\ell_n), \mathtt{w}(z_1), \dots \mathtt{w}(z_n), \mathtt{rel}(\ell_n), \dots \mathtt{rel}(\ell_1)$. Each of the threads t_j, for $j \in [d]$, has a single nested critical section consisting of the locks $\ell_i \in [n]$ such that the i^{th} vector of Y has its j^{th} coordinate 0, i.e, $y_i[j] = 0$. The events in the critical section are all write events of all variables $z_k \in [n]$ with $x_k[j] = 1$. The trace orders all events of each thread t_d consecutively, and all the events overall in increasing order of d. See Figure 5 for an illustration. We refer to Appendix D for the correctness, which concludes the proof of Theorem 8.

6 Conclusion

In this work we have taken a fine-grained view of the complexity of popular notions of dynamic data races. We have established a range of lower bounds on the complexity of detecting HB races, sync-preserving races, as well as races based on the locking discipline (lock-cover/lock-set races). Moreover, we have characterized cases where lower bounds based on SETH are not possible under NSETH. Finally, we have proven new upper bounds for detecting HB and lock-set races. To our knowledge, this is the first work that characterizes the complexity of well-established dynamic race-detection techniques, allowing for a rigorous characterization of their trade-offs between expressiveness and running time.

References

1. Helgrind: a thread error detector. `https://valgrind.org/docs/manual/hg-manual.html`. Accessed: 2021-04-30.
2. Intel Inspector. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html`. Accessed: 2021-04-30.
3. Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, page 377–391, USA, 2016. Society for Industrial and Applied Mathematics.
4. Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 69–78, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1147403.1147416`, `doi:10.1145/1147403.1147416`.
5. Hans-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, page 3, USA, 2011. USENIX Association.
6. Hans-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, page 9–14, New York, NY, USA, 2012. Association for Computing Machinery. URL: `https://doi.org/10.1145/2414729.2414732`, `doi:10.1145/2414729.2414732`.
7. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 68–78, New York, NY, USA, 2008. Association for Computing Machinery. URL: `https://doi.org/10.1145/1375581.1375591`, `doi:10.1145/1375581.1375591`.
8. Karl Bringmann. Fine-Grained Complexity Theory (Tutorial). In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/10243`, `doi:10.4230/LIPIcs.STACS.2019.4`.
9. Marco L Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 261–270, 2016.
10. Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11 – 16, 1991. URL: `http://www.sciencedirect.com/science/article/pii/002001909190055M`, `doi:https://doi.org/10.1016/0020-0190(91)90055-M`.
11. Peter Chini, Jonathan Kolberg, Andreas Krebs, Roland Meyer, and Prakash Saivasan. On the Complexity of Bounded Context Switching. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7873`, `doi:10.4230/LIPIcs.ESA.2017.27`.
12. Peter Chini, Roland Meyer, and Prakash Saivasan. Fine-grained complexity of safety verification. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 20–37, Cham, 2018. Springer International Publishing.

13 Peter Chini and Prakash Saivasan. A Framework for Consistency Algorithms. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020)*, volume 182 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/13283`, `doi:10.4230/LIPIcs.FSTTCS.2020.42`.

14 Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '91, pages 85–96, New York, NY, USA, 1991. ACM. URL: `http://doi.acm.org/10.1145/122759.122767`, `doi:10.1145/122759.122767`.

15 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM. URL: `http://doi.acm.org/10.1145/1250734.1250762`, `doi:10.1145/1250734.1250762`.

16 Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1542476.1542490`, `doi:10.1145/1542476.1542490`.

17 Jiawei Gao, Russell Impagliazzo, Antonina Kolokolova, and Ryan Williams. Completeness for first-order properties on sparse structures with algorithmic applications. *ACM Trans. Algorithms*, 15(2), December 2018. URL: `https://doi.org/10.1145/3196275`, `doi:10.1145/3196275`.

18 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

19 Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Toward integration of data race detection in dsm systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, November 1999. URL: `http://dx.doi.org/10.1006/jpdc.1999.1574`, `doi:10.1006/jpdc.1999.1574`.

20 Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 406–422, New York, NY, USA, 2013. Association for Computing Machinery. URL: `https://doi.org/10.1145/2517349.2522736`, `doi:10.1145/2517349.2522736`.

21 Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 157–170, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3062341.3062374`, `doi:10.1145/3062341.3062374`.

22 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. URL: `http://doi.acm.org/10.1145/359545.359563`, `doi:10.1145/359545.359563`.

23 Umang Mathur, Dileep Kini, and Mahesh Viswanathan. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):145:1–145:29, October 2018. URL: `http://doi.acm.org/10.1145/3276515`, `doi:10.1145/3276515`.

24 Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. The complexity of dynamic data race prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 713–727, New York, NY, USA, 2020. Association for Computing Machinery. URL: `https://doi.org/10.1145/3373718.3394783`, `doi:10.1145/3373718.3394783`.

25 Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. Optimal prediction of synchronization-preserving races. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. URL:

https://doi.org/10.1145/3434317, doi:10.1145/3434317.

26 Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 22–31, New York, NY, USA, 2007. Association for Computing Machinery. URL: https://doi.org/10.1145/1250734.1250738, doi:10.1145/1250734.1250738.

27 Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, June 2003. URL: http://doi.acm.org/10.1145/966049.781528, doi:10.1145/966049.781528.

28 Andreas Pavlogiannis. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. URL: https://doi.org/10.1145/3371085, doi:10.1145/3371085.

29 Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. *SIGPLAN Not.*, 38(10):179–190, June 2003. URL: http://doi.acm.org/10.1145/966049.781529, doi:10.1145/966049.781529.

30 Jake Roemer, Kaan Genç, and Michael D. Bond. High-coverage, unbounded sound predictive race detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 374–389, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3192366.3192385, doi:10.1145/3192366.3192385.

31 Grigore Rosu. RV-Predict, Runtime Verification. https://runtimeverification.com/predict/, 2018. Accessed: 2018-04-01.

32 Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1986308.1986334.

33 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997. URL: http://doi.acm.org/10.1145/265924.265927, doi:10.1145/265924.265927.

34 Koushik Sen, Grigore Roşu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

35 Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. WBIA '09, 2009.

36 Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

37 Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2103656.2103702, doi:10.1145/2103656.2103702.

38 Martin Sulzmann and Kai Stadtmüller. Efficient, near complete, and often sound hybrid dynamic data race prediction. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR 2020, page 30–51, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3426182.3426185, doi:10.1145/3426182.3426185.

39 Christoph von Praun. *Race Detection Techniques*, pages 1697–1706. Springer US, Boston, MA, 2011. URL: https://doi.org/10.1007/978-0-387-09766-4_38, doi:10.1007/978-0-387-09766-4_38.

40 Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, June 2011. URL: https://doi.org/10.1145/1993316.1993534, doi:10.1145/1993316.1993534.

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

42 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the ICM*, volume 3, pages 3431–3472. World Scientific, 2018.

43 M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, March 2009. URL: https://doi.org/10.1109/MSP.2009.56, doi:10.1109/MSP.2009.56.

A Fine-Grained Complexity and Popular Hypotheses

In this section we present notions of fine-grained complexity theory that are relevant to our work. We refer to the survey [42] for a detailed exposition on the topic.

This theory relates the computational complexity of problems under the following, more refined, notion of reduction than the standard ones used in traditional complexity theory. Informally, the definition says that if there is an algorithm for some problem B faster than its assumed lower bound, then such a reduction from some problem A to B gives an algorithm for A thatis faster than its conjectured lower bound.

Fine-grained Reductions. Assume that A and B are computational problems and $a(n)$ and $b(n)$ are their conjectured running time lower bounds, respectively. Then we say A (a, b)-reduces to B, denoted by A ${}_{(\mathsf{a})}\preceq_{(\mathsf{b})}$ B, if for every $\epsilon > 0$, there exists $\delta > 0$, and an algorithm R for A that runs in time $a(n)^{(1-\delta)}$ on inputs of length n, making q calls to an oracle for B with query lengths $n_1, \ldots, n_q$, where,

$$\sum_{1}^{q}(b(n))^{(1-\epsilon)} \leq (a(n))^{(1-\delta)}.$$

Problems that can be reduced to each other such that the lower bounds for each problem are the same in both reductions, i.e., $\mathrm{A}_{(\mathsf{a})}\preceq_{(\mathsf{b})}\mathrm{B}$ and $\mathrm{B}_{(\mathsf{b})}\preceq_{(\mathsf{a})}\mathrm{A}$, are intuitively thought to have the same underlying 'reason' for hardness, and are said to be fine-grained equivalent.

A reduction $\mathrm{A}_{(\mathsf{a})}\preceq_{(\mathsf{b})}\mathrm{B}$ would be interesting for B if $a(n)$ was a proven or well-believed conjectured lower bound on A, thus implying a believable lower bound on B. One such well-believed conjecture in complexity theory is SETH [18] for the classic CNF-SAT problem, originally defined for deterministic algorithms, but now widely believed for randomized algorithms as well.

▶ **Hypothesis 1** (Strong Exponential Time Hypothesis (SETH)). *For every $\epsilon > 0$ there exists an integer $k \geq 3$ such that CNF-SAT on formulas with clause size at most k and n variables cannot be solved in $O(2^{(1-\epsilon)n})$ time even by a randomized algorithm.*

SETH implies a lower bound conjecture, denoted by OVH, on the Orthogonal Vectors problem OV, as shown by a reduction from CNF-SAT to k-OV [41]. Thus, a conditional lower bound under OVH implies one under SETH as well, leading to numerous conditional lower bound results under OVH [See [42] for a detailed literature review]. This paper will also prove such results on several data race detection problems, hence we now state k-OV and OVH formally.

An instance of k-OV is an integer $d = \omega(\log n)$ and k sets $A_i \subseteq \{0, 1\}^d$, $i \in [n]$ such that $|A_i| = n$, and denoted by $\mathsf{OV}(n, d)$.

▶ **Problem 2** (Orthogonal Vectors (k-OV)). *Given an instance $\mathsf{OV}(n, d, k)$, the k-OV problem is to decide if there are k vectors $a_i \in A_i$ for all $i \in [n]$ such that the sum of their point wise product is zero, i.e., $\sum_{j=1}^{d} \prod_{i=1}^{k} a_i[j] = 0$.*

For ease of exposition, we denote $\mathsf{OV}(n, d, 2)$ and 2-OV by OV(n,d) and OV respectively.

▶ **Hypothesis 2** (Orthogonal Vectors Hypothesis (OVH)). *No randomized algorithm can solve k-OV for an instance $\mathsf{OV}(n, d, k)$ in time $O(n^{(k-\epsilon)} \cdot \mathrm{poly}(d))$ for any constant $\epsilon > 0$.*

There is an impossibility result from [9] that proves that a reduction under SETH, and hence under OVH, is not possible unless the following NSETH conjecture is false.

▶ **Hypothesis 3** (Non-deterministic SETH (NSETH)). *For every $\epsilon > 0$, there exists a k so that k-TAUT is not in $\mathsf{NTIME}[2^{\mathsf{n}(1-\epsilon)}]$, where k-TAUT is the language of all k-DNF formulas which are tautologies.*

The impossibility result [9, Corollary 2] is as follows.

▶ **Theorem 13.** *If NSETH holds and a problem $C \in \mathsf{NTIME}[\mathsf{T_C}] \cap \mathsf{coNTIME}[\mathsf{T_C}]$, then for any problem B that is SETH-hard under deterministic reductions with time T_B, and $\gamma > 0$, we cannot have a fine-grained reduction $B \ {}_{(\mathsf{T_B})}\!\preceq_{(\mathsf{c})}\ C$ where $c = T_C^{(1+\gamma)}$.*

We show some of our problems satisfy the conditions of Theorem 13, and hence show lower bounds for these conditioned on one of two other hypotheses called HSH and FOPH($\forall\exists\exists$), described below.

An instance of the hitting set problem, denoted by HS, is an integer $d = \omega(\log n)$ and sets $X, Y \subseteq \{0,1\}^d$, $i \in [n]$ such that $|X| = |Y| = n$, and denoted by HS(n,d).

▶ **Problem 3** (Hitting Sets (HS))**.** *Given an instance HS(n,d), the HS problem is to decide if there is a vector $x \in X$ such that for all $y \in Y$ we have $x \cdot y \neq 0$, or informally, some vector in X hits all vectors in Y.*

▶ **Hypothesis 4** (Hitting Sets Hypothesis (HSH))**.** *No randomized algorithm can solve HS for an instance HS(n,d) in time $O(n^{(2-\epsilon)} \cdot \mathrm{poly}(d))$ for any constant $\epsilon > 0$.*

HSH implies OVH, but the reverse direction is not known.

Finally we consider a subclass of first order formula over structures of size n and with m relational tuples [17].

▶ **Problem 4** (FO($\forall\exists\exists$))**.** *Decide if a given a first-order formula quantified by $\forall\exists\exists$ is has a model on a structure of size n with m relational tuples.*

It is known that FO($\forall\exists\exists$) can be solved in $O(m^{3/2})$ time using ideas from triangle detection algorithms [17]. For dense structures ($m = \Theta(n^2)$), this yields the bound $O(n^3)$. Although sub-cubic algorithms might be possible, achieving a truly quadratic bound seems unlikely or at least highly non-trivial.

B Proofs of Section 3

B.1 Proofs from Section 3.1

▶ **Lemma 10.** *Let $e_1 \leq^{\sigma}_{tr} e_2$ be events in σ such that $\mathsf{tid}(e_1) \neq \mathsf{tid}(e_2)$. We have, $e_1 \leq^{\sigma}_{HB} e_2 \iff \neg(\mathsf{AcqLS}^{\sigma}_{e_2} \sqsubseteq \mathsf{RelLS}^{\sigma}_{e_1})$*

Proof. ($\Rightarrow$) Let $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$. Using the definition of $\leq^{\sigma}_{\mathsf{HB}}$, there must be a sequence of events $f_1, f_2 \dots f_k$ with $k > 1$, $f_1 = e_1$, $f_k = e_2$, and for every $1 \leq i < k$, $f_i \leq^{\sigma}_{\mathsf{tr}} f_{i+1}$ and either $f_i \leq^{\sigma}_{\mathsf{TO}} f_{i+1}$ or there is a lock ℓ, such that $f_i \in \mathsf{Releases}_{\sigma}(\ell)$ and $f_{i+1} \in \mathsf{Acquires}_{\sigma}(\ell)$. Let j be the smallest index i such that $\mathsf{tid}(f_i) \neq \mathsf{tid}(f_{i+1})$; such an index exists as $\mathsf{tid}(e_1) \neq \mathsf{tid}(e_2)$. Observe that there must be a lock ℓ for which $\mathrm{op}(f_j) = \mathtt{rel}(\ell)$ and $\mathrm{op}(f_{j+1}) = \mathtt{acq}(\ell)$. Observe that $pos_{\sigma}(f_j) < pos_{\sigma}(f_{j+1})$, $\mathsf{RelLS}^{\sigma}_{e_1}(\ell) \leq pos_{\sigma}(f_j)$ and $pos_{\sigma}(f_{j+1}) \leq \mathsf{AcqLS}^{\sigma}_{e_2}$, giving us $\mathsf{RelLS}^{\sigma}_{e_1}(\ell) < \mathsf{AcqLS}^{\sigma}_{e_2}(\ell)$.

($\Leftarrow$) Let ℓ be a lock such that $\mathsf{RelLS}^{\sigma}_{e_1}(\ell) < \mathsf{AcqLS}^{\sigma}_{e_2}(\ell)$. Then, there is a release event f and an acquire event g on lock ℓ such that $pos_{\sigma}(f) < pos_{\sigma}(g)$,$e_1 \leq^{\sigma}_{\mathsf{HB}} f$ and $g \leq^{\sigma}_{\mathsf{HB}} e_2$. This means $f \leq^{\sigma}_{\mathsf{HB}} g$ and thus $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$. ◀

For the sake of completeness, we present the computation of release lockstamps. As with Algorithm 1, we maintain the following variables. For each thread t and lock ℓ, we will maintain variables $\mathbb{C}_t$ and $\mathbb{L}_\ell$ that take values from the space of all lockstamps. We also additionally maintain an integer variable p_ℓ for each lock ℓ that stores the index (or relative position) of the earliest (according to the trace order $\leq^{\sigma}_{\mathsf{tr}}$) release event of lock ℓ in the trace.

Initially, we set each $\mathbb{C}_t$ and $\mathbb{L}_m$ to $\lambda\ell \cdot \infty$, for each thread t and lock m. Further, for each lock m, we set p_m to $n_m + 1$, where n_m is the number of release events of m in the trace; this can be obtained in a linear scan (or by reading the value of p_m at the end of a run of Algorithm 1). We traverse the events according to the total trace order and perform updates to the data structures as described in Algorithm 4, by invoking the appropriate *handler* based on the thread and operation of the event $e = \langle t, op\rangle$ being visited. At the end of each handler, we assign the lockstamp RelLS^σ_e to the event e.

Algorithm 4 *Assigning release lockstamps to events in the trace*

`acquire`$(t,\ \ell)$:
- $\mathbb{L}_\ell \leftarrow \mathbb{C}_t$
- $\mathsf{RelLS}^\sigma_e \leftarrow \mathbb{C}_t$

`release`$(t,\ \ell)$:
- $\mathsf{p}_\ell \leftarrow \mathsf{p}_\ell - 1$
- $\mathbb{C}_t \leftarrow \mathbb{C}_t[\ell \mapsto \mathsf{p}_\ell] \sqcap \mathbb{L}_\ell$
- $\mathsf{RelLS}^\sigma_e \leftarrow \mathbb{C}_t$

`read`$(t,\ x)$:
- $\mathsf{RelLS}^\sigma_e \leftarrow \mathbb{C}_t$

`write`$(t,\ x)$:
- $\mathsf{RelLS}^\sigma_e \leftarrow \mathbb{C}_t$

Let us now state the correctness of Algorithm 1 and Algorithm 4.

▶ **Lemma 14.** *On input trace σ, Algorithm 1 and Algorithm 4 correctly compute the lockstamps AcqLS^σ_e and RelLS^σ_e respectively for each event $e \in \mathsf{Events}_\sigma$.*

Proof Sketch. We focus on the correctness proof of Algorithm 1; the proof for Algorithm 4 is similar. The proof relies on the invariant maintained by Algorithm 1 the variables $\mathbb{C}_t$, $\mathbb{L}_\ell$ and p_ℓ for each thread t and lock ℓ, which we state next. Let π be the prefix of the trace processed at any point in the algorithm. Let C^π_t, L^π_ℓ and p^π_ℓ be the values of the variables $\mathbb{C}_t$, $\mathbb{L}_\ell$ and p_ℓ after processing the prefix π. Then, the following invariants are true:

- $C^\pi_t = \mathsf{AcqLS}^\pi_{e^\pi_t} = \mathsf{AcqLS}^\sigma_{e^\pi_t}$, where e^π_t is the last event in π performed by thread t
- $L^\pi_\ell = \mathsf{AcqLS}^\pi_{e^\pi_\ell} = \mathsf{AcqLS}^\sigma_{e^\pi_\ell}$, where e^π_ℓ is the last acquire event on lock ℓ in π.
- $p^\pi_\ell = pos^\pi_\ell(e^\pi_\ell)$, where e^π_ℓ is the last acquire event on lock ℓ in π.

These invariants can be proved using a straightforward induction, each time noting the definition of $\leq^\sigma_{\mathsf{HB}}$. ◀

▶ **Lemma 15.** *For a trace with $\mathcal{N}$ events and $\mathcal{L}$ locks, Algorithm 1 and Algorithm 4 both take $O(\mathcal{T} \cdot \mathcal{L})$ time.*

Proof. We focus on Algorithm 1; the analysis for Algorithm 4 is similar. At each acquire event, the algorithm spends $O(1)$ time for updating p_ℓ, $O(\mathcal{L})$ time for doing the $\sqcup$ operation, and $O(\mathcal{L})$ time for the copy operation ('$\mathsf{AcqLS}^\sigma_e \leftarrow \mathbb{C}_t$'). For a release event, we spend $O(\mathcal{L})$ for the two copy operations. At read and write events, we spend $O(\mathcal{L})$ for copy operations. This gives a total time of $O(\mathcal{N} \cdot \mathcal{L})$. ◀

▶ **Lemma 11.** *A trace σ has an HB-race iff there is pair of consecutive conflicting events in σ that is an HB-race. Moreover, σ has $O(\mathcal{N})$ many consecutive conflicting pairs of events.*

Proof. We first prove that if there is a an HB-race in σ, then there is a pair of consecutive conflicting events that is in HB-race. Consider the first HB-race, i.e., an HB-race (e_1, e_2) such that for every other HB-race (e'_1, e'_2), either $e_2 \leq^\sigma_{\mathsf{tr}} e'_2$ or $e_2 = e'_2$ and $e'_1 \leq^\sigma_{\mathsf{tr}} e_1$. We remark that such a race (e_1, e_2) exists if σ has any HB-race. We now show that (e_1, e_2) are a consecutive conflicting pair (on variable x). Assume on the contrary that there is an event $f \in \mathsf{Writes}_\sigma(x)$ such that $e_1 <^\sigma_{\mathsf{tr}} f <^\sigma_{\mathsf{tr}} e_2$. If either (e_1, f) or (f, e_2) is an HB-race, then this contradicts our assumption that (e_1, e_2) is the first HB-race in σ. Thus, $e_1 \leq^\sigma_{\mathsf{HB}} f$ and $f \leq^\sigma_{\mathsf{HB}} e_2$, which gives $e_1 \leq^\sigma_{\mathsf{HB}} e_2$, another contradiction.

We now turn our attention to the number of consecutive conflicting events in σ. For every read or write event e_2, there is at most one write event e_1 such that (e_1, e_2) is a consecutive conflicting pair (namely the latest conflicting write event before e_2) Further, for every read event e_1, there is at most one write event e_2 such that (e_1, e_2) is a consecutive conflicting pair (namely the earliest conflicting write event after e_1). This gives at most $2\mathcal{N}$ consecutive conflicting pairs of events. ◀

Let us now state the correctness of Algorithm 2.

▶ **Lemma 16.** *For a trace σ, Algorithm 2 reports a race iff σ has an HB-race.*

Proof Sketch. The proof relies on the following straightforward invariants; we skip their proofs as they are straightforward. In the following, e^π_x is the last event with $\mathsf{op}(e^\pi_x) = \mathtt{w}(x)$ in a trace π.

- After processing the prefix π of σ, $t^w_x = \mathsf{tid}(e^\pi_x)$ and $\mathbb{W}_x = e^\pi_x$.
- After processing the prefix π of σ, the set S_x is $\{(t, L) \,|\, \exists e \in \mathsf{Reads}_\pi(x), e^\pi_x \leq^\pi_{\mathsf{tr}} e, \mathsf{tid}(e) = t, \mathsf{RelLS}^\sigma_e = L\}$.

The rest of the proof follows from Lemma 15 and Lemma 14. ◀

Let us now characterize the time complexity of Algorithm 2.

▶ **Lemma 17.** *On an input trace with $\mathcal{N}$ events and $\mathcal{L}$ locks, Algorithm 2 runs in time $O(\mathcal{N} \cdot \mathcal{L})$.*

Proof Sketch. Each pair (t, L) of thread identifier and lockstamp is added atmost once in some set S_x (for some x). Also, each such pair is also compared against another timestamp atmost once. Each comparison of timestamps take $O(\mathcal{L})$ time. This gives a total time of $O(\mathcal{N} \cdot \mathcal{L})$. ◀

▶ **Theorem 4.** *Deciding whether σ has an HB race can be done in time $O(\mathcal{N} \cdot \min(\mathcal{T}, \mathcal{L}))$.*

Proof. We focus on proving that there is an $O(\mathcal{N} \cdot \mathcal{L})$ time algorithm, as the standard vector-clock algorithm [19] for checking for an HB-race runs in $O(\mathcal{N} \cdot \mathcal{T})$ time. Our algorithm's correctness is stated in Lemma 16 and its total running time is $O(\mathcal{N} \cdot \mathcal{L})$ (Lemma 17 and Lemma 15). ◀

B.2 Proofs from Section 3.2

▶ **Theorem 1.** *For any $\epsilon > 0$, there is no algorithm that detects even a single HB race that involves a read in time $O(\mathcal{N}^{2-\epsilon})$, unless the OV hypothesis fails.*

Proof. Consider a pair of events $\mathtt{w}(z)$ from the d threads $t(x, i), i \in [d]$, and $\mathtt{r}(z) \in t_y$ for some x, i, y. We have $\mathtt{w}(z) \leq^\sigma_{\mathsf{HB}} \mathtt{r}(z)$ iff there is some path from $\mathtt{w}(z)$ to $\mathtt{r}(z)$ in $\mathsf{G}(\leq^\sigma_{\mathsf{HB}})$. As $\mathtt{w}(z)$ and $\mathtt{r}(z)$ are in different threads, such a path can only be through lock events in a sequence of threads such that the first and last threads are $t(x, i)$ for some $i \in [d]$ and t_y, and every consecutive pair of threads in the sequence holds a common lock. Now all the locks in t_y are $l(y, i)$ for all i where $y[i] = 1$. Consider the lock corresponding to any $i \in [d]$. The only thread $t(x', i)$ that also holds this lock corresponds to the last x' such that $x'[i] = 1$. The only other lock held by $t(x', i)$ is l_i. If $\mathtt{w}(z)$ is in $t(x', i)$, we are done. Otherwise the only common lock between these threads $t(x', i)$ and those of $\mathtt{w}(z)$ can be one of the l_i. The threads of $\mathtt{w}(z)$ contain all l_i where $x[i] = 1$. Hence, for there to be a common lock between these threads, there must be at least one i such that $x'[i] = 1$ and $x[i] = 1$. As this thread also has the lock $l(y, i)$, $y[i]$ is also 1.

Thus, there is a path from $\mathtt{w}(z)$ to $\mathtt{r}(z)$ if and only if there is at least one $i \in [d]$ such that $x[i] = y[i] = 1$, hence x and y are not orthogonal. A pair of orthogonal vectors of OV thus corresponds to a write-read HB-race in the reduced trace.

Finally we turn our attention to the complexity. In time $O(n \cdot d)$, we have reduced an OV instance to determining whether there is a write-read HB race in a trace of $\mathcal{N} = O(nd)$ events. If there was a sub-quadratic i.e. $O((n \cdot d)^{(2-\epsilon)}) = n^{(2-\epsilon)} \cdot \text{poly}(d)$ algorithm for detecting a write-read HB race, then this would also solve OV in $n^{(2-\epsilon)} \cdot \text{poly}(d)$ time, refuting the OV hypothesis. ◀

▶ **Lemma 12.** *FO($\forall\exists\exists$) reduces to MCONN on a graph G with $O(n)$ nodes in $O(n^2)$ time.*

Proof. For intuition, assume the first order property is on an undirected graph with n variables and m edges. Let the property be specified in quantified 3-DNF form with a constant number of predicates, i.e., $\phi = \forall x \exists y \exists z \ (\psi_1 \vee \psi_2 \vee \dots \psi_k)$, where x, y, z represent nodes of the graph, and each ψ_i is a conjunction of 3 variables representing edges of the graph, for example $e(x, y) \wedge \neg e(y, z) \wedge e(x, z)$. The property is then true if and only if some predicate is satisfied, which is true if all of its variables are satisfied ($e(x, y)$ is satisfied when edge (x, y) is in the graph). Denote the graph on which ϕ is defined by $H(I, J)$, where I and J are respectively the sets of nodes and edges of H.

The instance of MCONN is constructed given H and ϕ as follows. Construct a $(2k+2)$-partite graph $G(V, E)$ by first creating $2k+2$ copies of I. Denote these copies by $S, Y_i, Z_i, T, \ i \in [k]$, and the copy of each node $x \in I$ in any part, say S, by $x(S)$. $\psi_i = (e_1 \wedge e_2 \wedge e_3)$ is encoded by connecting the sets (S, Y_i) to represent e_1, (Y_i, Z_i) for e_2 and (Z_i, T) for e_3 as follows. If e_i is of the form $e(x, y)$ (and not its negation), then draw a copy of H between its corresponding sets, say S and Y_i without loss of generality. That is, for every x, y, $(x, y) \in J \Leftrightarrow (x(S), y(Y_i)) \in E$. If on the other hand e_i is of the form $\neg e(x, y)$ then connect a copy of the complement of H, i.e., $(x, y) \notin J \Leftrightarrow (x(S), y(Y_i)) \in E$.

Finally define $|I|$ pairs $(x(S), x(T))$ as the (s, t) pairs for MCONN.

We now prove this reduction is correct. First, assume ϕ is true. Then for every node x, there exist nodes y, z such that some predicate is true. If ψ_i is the predicate that is satisfied for some node u, then there is a path between $u(S)$ and $u(T)$ through the parts S, Y_i, Z_i and T as follows. As the first variable is satisfied, then if it is $e(x, y)$, then $(x, y) \in J$, and $x(S)$ is connected to $y(Y_i)$, and if it is $\neg e(x, y)$, then $(x, y) \notin J$ and again $x(S)$ is connected to $y(Y_i)$. Similarly, $y(Y_i)$ is connected to $z(Z_i)$, and $z(Z_i)$ to $x(T)$. These edges form a 3 length path between $x(S)$ and $x(T)$.

Now consider the reverse case, and assume the MCONN problem is true, that is , there is a path between every $(x(S), x(T))$ pair. Note that the construction of edges in G is such that any path from $x(S)$ to $x(T)$ has to be a 3 length path, connecting the copy of x in S to its copy in some Y_i, from this Y_i to its corresponding Z_i, and from Z_i to T. Also, this path exists only if all variables of the corresponding ψ_i are true. Hence, as there is a path between every pair $(x(S), x(T))$, and one pair is defined for every variable x, some predicate is satisfied for every x. Thus ϕ is also true.

Finally, the time of the reduction is equal to the size of G. This is $2k+2 = O(1)$ graphs, each of which is either H or its complement. Hence $|G| = O(m + n + (n^2 - m) + n) = O(n^2)$. ◀

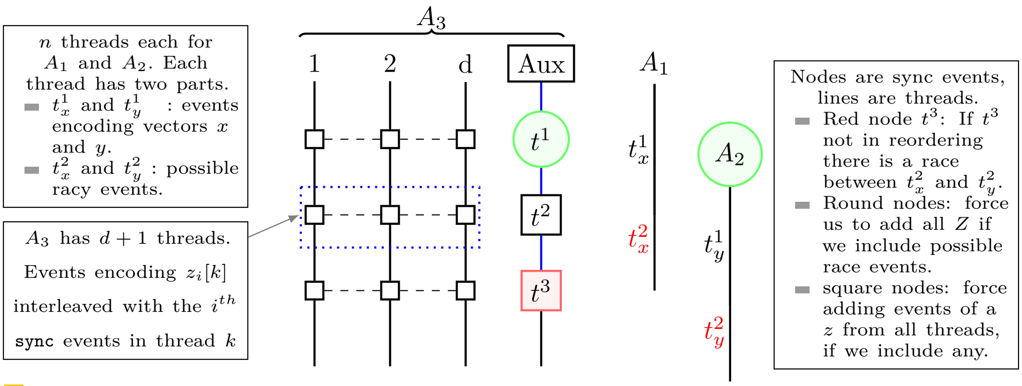

Figure 6 Intuition for the reduction from 3-OV to `sync`-preserving race detection. Thread *Aux* allows forming a trace such that 3-OV has a solution iff there is a `sync`-preserving race.

▶ **Theorem 3.** *For any $\epsilon > 0$, if there is an algorithm for detecting any HB race in time $O(\mathcal{N}^{1+\epsilon})$, then there is an algorithm for FO(∀∃∃) formulas in time $O(m^{1+\epsilon})$.*

Proof. We first reduce the instance of FO(∀∃∃) to MCONN as in the proof of Lemma 12. Let $G(V, E)$ be the multi-partite graph for MCONN and S, T the first and last parts of nodes of G. We add a sufficient number of nodes, referred as dummy nodes, to make G sparse. Let every node x of $V \setminus T$ correspond to a distinct thread t_x and form one write access event to a distinct variable v_x in the thread. Let each node t in T also correspond to a write access event of the variable corresponding to the copy of t in S, and be in a new thread. Define $|E|$ locks, and for every edge $(a, b) \in E$, let the events corresponding to v_a and v_b hold the lock $l_{(a,b)}$ corresponding to (a, b). The trace σ for first lists all threads corresponding to the dummy nodes in some fixed arbitrary order, then the threads corresponding to nodes in S, followed by those in each Y_i, followed by those in each Z_i, in a fixed arbitrary order, and finally those in T.

This reduction is seen to be correct by observing that G was modified to be the transitive reduction graph of σ, and the only HB-race events can be the pairs of write events corresponding to the pairs of nodes given as input to MCONN. Thus, each pair of events does not form an HB-race if and only if G has a path between its corresponding pair of nodes.

To analyze the time of the reduction, first we see that the size of σ is the size of G, with dummy nodes added to have $n = O(n^2)$, and hence $O(n^2)$. There are $O(n^2)$ variables, locks and threads in σ. If deciding if the given trace has an HB-race has an $O((n^2)^{1+\epsilon})$ time algorithm, then FO(∀∃∃) can be solved in $O(n^{2+\epsilon'})$ time, which is $O(m^{1+\epsilon'})$ time for properties on dense structures. ◀

C Proofs of Section 4

▶ **Theorem 5.** *For any $\epsilon > 0$, there is no algorithm that detects even a single sync-preserving race in time $O(\mathcal{N}^{3-\epsilon})$, unless the 3-OV hypothesis fails. Moreover, the statement holds even for traces over a single variable.*

Proof. Consider any sync-preserving correct reordering ρ of σ that exposes a data race $(\mathtt{w}(z), \mathtt{r}(z))$ on the local traces t_x and t_y. The following statements are straightforward to verify based on the definition of sync-preserving correct reorderings.

1. For every $k \in [d]$, the first $\mathtt{sync}(\ell_k)$ event the trace t_k is also in ρ.

2. The auxiliary trace t cannot have an open critical section in ρ. This implies that for every trace t_k with $k \in [d]$, the last event of t_k in ρ cannot be its i^{th} $\mathtt{sync}(\ell_k)$ event, where i is even. Moreover, the number of $\mathtt{sync}(\ell_k)$ events in ρ is the same for every trace t_k with $k \in [d]$.

First, consider that the 3-OV instance has a solution, i.e., there exist $x \in A_1$, $y \in A_2$ and $z \in A_3$ such that x, y, z are orthogonal, and we argue that σ has a data race that is also sync-preserving. We construct a sync-preserving correct reordering ρ of σ that exposes the data race. We only specify the local traces that exist in ρ, as their interleaving that constructs ρ will be identical to the one in σ (in other words, we only specify the prefix up to which every local trace of σ is executed in ρ). We execute the traces t_x and t_y all the way before the corresponding $\mathtt{w}(z)$ and $\mathtt{r}(z)$ events (hence we are exposing a race between these two events). For every $k \in [d]$ if $z[k] = 0$ or $x[k] = 0$, we execute t_k up to the $(2 \cdot i)^{th}$ $\mathtt{sync}(\ell_k)$ event, where i is such that z is the i^{th} vector of A_3. On the other hand, if $y[k] = 0$, we execute t_k up to the first $\mathtt{rel}(\ell_k)$ event that appears after the $(2 \cdot i)^{th}$ $\mathtt{sync}(\ell_k)$ event in t^k. Finally, we execute t^k until its $(i-1)^{th}$ $\mathtt{rel}(X)$ event.

It is easy to verify that ρ is a valid correct reordering. Indeed, we have two open critical sections in the threads t_x and t_y, on the locks X and Y respectively. Moreover, for every $k \in [d]$, we have the following.

1. If $z[k] = 0$, there are no other open critical sections.
2. If $z[k] = 1$ and $x[k] = 0$, there is one open critical section in the thread $z[k]$ on lock l_k.
3. If $z[k] = x[k] = 1$ and $y[k] = 0$, there is one open critical section in the thread $z[k]$ on lock l'_k.

We now consider the opposite direction, i.e., assume that there is a sync-preserving race in σ, and we argue that there exist $x \in A_1$, $y \in A_2$ and $z \in A_3$ such that x, y, z are orthogonal. Consider any sync-preserving correct reordering ρ that exposes a race on the access events of two local traces t_x and t_y. Because of Item 1 above, every trace t_k is at least partially present in ρ. Because of Item 1 above, every such trace executes the same number of $\mathtt{sync}(\ell_k)$ events in ρ, and this number is odd. We argue that the triplet x, y, z is orthogonal, where z is the i^{th} vector of A_3 such that each t_k executes $2 \cdot (i-1) + 1$ $\mathtt{sync}(\ell_k)$ events in ρ. Indeed, consider any $k \in [d]$ and assume that $x[k] = y[k] = 1$. If $z[k] = 1$, then we have a $\mathtt{acq}(\ell)$ event in t_k that immediately precedes its last $\mathtt{sync}(\ell_k)$ event. Since $x[k] = 1$, the trace t_x also has an $\mathtt{acq}(l_k)$ event. Since the $\mathtt{acq}(l_k)$ event of t_x is after the $\mathtt{acq}(l_k)$ event of t_k in σ, the matching $\mathtt{rel}(l_k)$ of t_k must also be in ρ. This implies that the $\mathtt{acq}(l'_k)$ event of t_k that immediately succeeds its last $\mathtt{rel}(l_k)$ event is also in ρ. Since $y[k] = 1$, the trace t_y also has an $\mathtt{acq}(l'_k)$ event. Since the $\mathtt{acq}(l'_k)$ event of t_y is after the $\mathtt{acq}(l'_k)$ event of t_k in σ, the matching $\mathtt{rel}(l'_k)$ of t_k must also be in ρ. However, we now have another $\mathtt{sync}(\ell_k)$ event of t_k in ρ, in particular, the $\mathtt{sync}(\ell_k)$ event that immediately precedes its last $\mathtt{rel}(l'_k)$ event. But this results in an even number of $\mathtt{sync}(\ell_k)$ events of t_x being present in ρ, which contradicts our observation in Item 2. Thus, if $x[k] = y[k] = 1$, we necessarily have that $z[k] = 0$, and the triplet x, y, z is orthogonal.

The desired result follows. ◀

D Proofs of Section 5

▶ **Theorem 6.** *For any $\epsilon > 0$, any $\mathcal{T} \geq 2$ and any $\mathcal{L} = \omega(\log \mathcal{N})$, there is no algorithm that detects even a single lock-cover race in time $O(\mathcal{N}^{2-\epsilon})$, unless the OV hypothesis fails.*

Proof of Theorem 6. To see why the reduction is correct, observe that if there a solution to OV, that is, a pair of vectors x, y such that for all $k \in [d]$, $x[k] = 0$ or $y[k] = 0$, implies that the corresponding events in σ, say e_x and e_y, have for each lock $k \in [d]$ either e_x does not hold the lock or e_y does not. As they have distinct thread ids too, e_x and e_y form two conflicting events with $\mathsf{locksHeld}_\sigma(e_1) \cap \mathsf{locksHeld}_\sigma(e_2) = \varnothing$. Similarly, a lock-cover race implies that for every lock, one of the events in race do not hold the lock, hence have their corresponding coordinate in OV 0. The events are thus orthogonal to each other.

Regarding the complexity, we have used $O(n \cdot d)$ time to construct a trace σ with $\mathcal{N} = O(n \cdot d)$ events. If we can detect a lock-cover race in σ in $O(\mathcal{N}^{(2-\epsilon)})$ time, then OV can be solved in $O(n^{(2-\epsilon)} \cdot \mathrm{poly}(d))$ time, contradicting the OV hypothesis. ◀

▶ **Theorem 9.** *Deciding whether a trace σ has a lock-set race on a variable x can be performed in $O(\mathcal{N})$ time. Thus, deciding whether σ has a lock-set race can be performed in $O(\mathcal{N} \cdot \min(\mathcal{L}, \mathcal{V}))$ time.*

Proof. We first argue that the algorithm maintains the invariant stated in Equation (2). The invariant for A is trivial to verify. Moreover, it is easy to see that, assuming that the invariant holds before processing an $\mathtt{acq}(\ell)$ or $\mathtt{rel}(\ell)$ event, it also holds after processing that event. Indeed, for an event $\mathtt{acq}(\ell)$, we have $\ell \in A$, and to maintain $C = \overline{A} \cap B$, we remove ℓ from C if $\ell \in B$. Similarly for an event $\mathtt{rel}(\ell)$. To see that the invariant is maintained after processing an access event $\mathtt{w}(x)/\mathtt{r}(x)$, note that we have

$$B \setminus C = B \cap \overline{C} = B \cap \left(\overline{B \cap \overline{A}}\right) = B \cap \left(\overline{B} \cup A\right) = B \cap A$$

and thus updating $B \leftarrow B \setminus C$ yields

$$\mathsf{Locks}_\sigma \cap \bigcap_{\substack{e' \in \mathsf{Accesses}_\sigma(x) \\ e' <^\sigma_{\mathsf{tr}} e}} \mathsf{locksHeld}_\sigma(e') \cap \mathsf{locksHeld}_{tr}(e) = \mathsf{Locks}_\sigma \cap \bigcap_{\substack{e' \in \mathsf{Accesses}_\sigma(x) \\ e' \leq^\sigma_{\mathsf{tr}} e}} \mathsf{locksHeld}_\sigma(e')$$

Finally, at this point we have $\overline{A} \cap B = \overline{A} \cap B \cap A = \varnothing$, thus the invariant also holds for C.

We now turn our attention to complexity. Using a bit-set representation of the sets A, B and C, it is clear that each of the operations except $\mathtt{w}(x)/\mathtt{r}(x)$ take constant time per event. Each $\mathtt{w}(x)/\mathtt{r}(x)$ operation takes $O(|C|)$ time. Note, however, that because of the previous invariant, every lock is removed from B at most once, hence the total time for performing all set differences $B \leftarrow B \setminus C$ is $O(\mathcal{N} + \mathcal{L}) = O(\mathcal{N})$. Thus the total time is $O(\mathcal{N})$. The desired result follows. ◀

▶ **Theorem 8.** *For any $\epsilon > 0$ and any $\mathcal{T} = \omega(\log n)$, there is no algorithm that detects even a single lock-cover race in time $O(\mathcal{N}^{2-\epsilon})$, unless the HS hypothesis fails.*

Proof. First, assume there is a solution to HS, i.e., $\exists x_k \in X\ \forall y_i \in Y\ \exists j \in [d]\ x_k[j] = y_i[j] = 1$. Then for the variable z_k, for every lock ℓ_i, there is a thread t_j that contains $\mathtt{w}(z_k)$ (as $x_k[j] = 1$) but does not contain lock ℓ_i (as $y_i[j] = 1$). Thus $\bigcap_{e \in \mathsf{Accesses}_\sigma(z_k)} \mathsf{locksHeld}_\sigma(e) = \varnothing$, and we have a lock-set race on variable z_k as there are at least two $\mathtt{w}(z_k)$ conflicting events, one in the thread t_j and the other in thread t_0. For the opposite direction, assume that HS does not have a solution, i.e., $\forall x_k \in X\ \exists y_i \in Y\ \forall j \in [d]\ (x_k[j] = 0 \text{ or } y_i[j] = 0)$. Then for each variable z_k, there is some lock ℓ_i such that every thread that contains a write event $\mathtt{w}(z_k)$ (thus $x_k[j] = 1$) also contains the lock ℓ_i (as necessarily $y_i[j] = 0$). Hence, for every variable z_k, some lock ℓ_i is held by all its access events. Thus σ does not have a lock-set race.

Regarding the complexity, we have created a trace σ with $\mathcal{N} = O(n \cdot d)$ events in $O(n \cdot d)$ time. Thus, any $O(\mathcal{N}^{(2-\epsilon)})$ time algorithm for HS implies an $O(n^{(2-\epsilon)} \cdot \mathrm{poly}(d))$ time algorithm for HS, contradicting the HS hypothesis. ◀