Dynamic Data-Race Detection Through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, IL, USA

Umang Mathur
University of Illinois at Urbana-Champaign, IL, 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 $\sigma$ with $N$ events, $T$ threads and $L$ locks. Our main results are as follows. First, we show that happens-before $HB$ races can be detected in $O(N \cdot \min(T, L))$ time, improving over the standard $O(N \cdot T)$ bound when $L = o(T)$. 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 3-OV and thus have a cubic lower bound, when $T = \Omega(N)$. This establishes a complexity separation from HB races which are known to be strictly less expressive. Third, we show that lock-cover races are hard for 2-OV, and thus have a quadratic lower-bound, even when $T = 2$ and $L = \omega(\log N)$. The similar notion of lock-set races is known to be detectable in $O(N \cdot L)$ time, and thus we achieve a complexity separation between the two. Moreover, we show that lock-set races become hitting-set (HS)-hard when $L = \Theta(N)$, 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.

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 accesses 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, 27], and they have been deemed pure evil [6] due to the problems they have caused in the past [44]. Moreover, many compiler optimizations are unsound in the presence of data races [37, 41], while data-race freeness is often a requirement for assigning well-defined semantics to programs [7].

© Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis; licensed under Creative Commons License CC-BY 4.0
32nd International Conference on Concurrency Theory (CONCUR 2021).
Editors: Serge Haddad and Daniele Varacca; Article No. 16; pp. 16:1–16:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Dynamic Data-Race Detection Through the Fine-Grained Lens

The importance of data races in concurrency has led to a multitude of techniques for detecting them efficiently [4, 40]. 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 [38, 21, 29]. As full dynamic data race prediction is NP-hard in general [25], 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 [23]. 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 [36]. As vector clock computation is known to require \( \Theta(N \cdot T) \) time on traces of \( N \) events and \( T \) threads [10], HB race detection is often assumed to suffer the same bound, and has thus been a subject of further practical optimizations [30, 16].

Synchronization preserving races. HB races were recently generalized to sync(hronization)-preserving races [26]. 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 [40]. 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 [34]. 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 Section 2.3 for relevant notions in fine-grained complexity and popular hypotheses. The input is always a concurrent trace $\sigma$ of length $N$, consisting of $T$ threads, $L$ locks, and $V$ variables.

**Happens-before race.** 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 $\sigma$ that participate in an HB race, which is known to take $O(N \cdot T)$ time [19]. Note that the bound is quadratic when $T = \Theta(N)$, 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 $O(N^{2-\epsilon})$, 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 [43]. It is also known that the strong exponential time hypothesis (SETH) implies the Orthogonal Vectors hypothesis [42]. 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 $N^{3/2}$.

- **Theorem 2.** For any $\epsilon > 0$, there is no $(2^N, N^{3/2+\epsilon})$-fine-grained reduction from SAT to the problem of detecting any HB race, 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 $\forall \exists$ on structures of size $n$ with $m$ relational tuples (denoted $FO(\forall \exists)$).

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

  It is known that $FO(\forall \exists)$ 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 $FO(\forall \exists)$, then detecting any HB race cannot take $O(N^{1+\epsilon})$ time for any $\epsilon < 1/2$. Although improvements for $FO(\forall \exists)$ over the current $O(m^{3/2})$ bound might be possible, we find that a truly linear bound $O(m)$ would require major breakthroughs.

  Under this hypothesis, Theorem 3 implies a super-linear bound for HB races.

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

- **Theorem 4.** Deciding whether $\sigma$ has an HB race can be done in time $O(N \cdot \min(T, L))$.

  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 $\sigma$ as racy).

---

1 Even the well-studied problem of testing triangle freeness, which is a special case of the similarly flavored $FO(\exists \exists)$, has the super-linear bound $O(n^{\omega})$, where $\omega$ is the matrix multiplication exponent.
Synchronization-preserving races. Next, we turn our attention to the recently introduced sync-preserving races [25]. It is known that detecting sync-preserving races takes $O(N \cdot V \cdot T^3)$ 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 $O(N^{3-\epsilon})$, 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 $O(N^2 \cdot L)$ time for lock-cover races and in $O(N \cdot L)$ 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 $T \geq 2$ and any $L = \omega(\log N)$, there is no algorithm that detects even a single lock-cover race in time $O(N^{2-\epsilon})$, unless the OV hypothesis fails.

Observe that the $O(N \cdot L)$ bound for lock-set races also becomes quadratic, when the number of locks is unbounded (i.e., $L = \Theta(N)$). 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 $(2^N, N^{1+\epsilon})$-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 $T = \omega(\log n)$, there is no algorithm that detects even a single lock-set race in time $O(N^{2-\epsilon})$, 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 $\sigma$ has a lock-set race on a given variable $x$ can be performed in $O(N)$ time. Thus, deciding whether $\sigma$ has a lock-set race can be performed in $O(N \cdot \min(L, V))$ time.

Hence, Theorem 9 strengthens the $O(N \cdot L)$ upper bound for lock-set races when $V = o(L)$. 
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) [23] and the lock-set principle in Eraser [34] 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 [30] and FastTrack [16], altogether different algorithms (e.g., the GoldiLocks algorithm [15]), and hybrid techniques [28]. 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 [35]. While earlier work on data race prediction focused on explicit [35] or symbolic [32, 33] enumeration, recent efforts have focused on scalability [38, 24, 21, 29, 31, 39]. The more recent notion [26] of sync-preserving races generalizes the notion of HB. As the complexity of race prediction is prohibitive (NP-hard in general [25]), 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 [43] 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 [25] 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 \( \ellab(e) = (t, op) \), 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 write \( e = (t, op) \) instead of \( \ellab(e) = (t, op) \). For the purpose of this presentation, an operation can be one of

(a) read (\( r(x) \)) from, or write (\( w(x) \)) to, a shared memory variable \( x \), or
(b) acquire (\( \text{acq}(\ell) \)) or release (\( \text{rel}(\ell) \)) of a lock \( \ell \).

For an event \( e = (t, op) \), we use \( \ellid(e) \) and \( \ellop(e) \) to denote respectively the thread identifier \( t \) and the operation \( op \). For a trace \( \sigma \), we use \( \ellevents_\sigma \) to denote the set of events that appear in \( \sigma \). Similarly, we will use \( \ellthreads_\sigma \), \( \elllocks_\sigma \) and \( \ellvars_\sigma \) to denote respectively the set of threads, locks and shared variables that appear in trace \( \sigma \). We denote by \( N = |\ellevents_\sigma| \), \( T = |\ellthreads_\sigma| \), \( L = |\elllocks_\sigma| \), and \( V = |\ellvars_\sigma| \). The set of read events and write events on variable \( x \in \ellvars_\sigma \) will be denoted by \( \ellreads_\sigma(x) \) and \( \ellwrites_\sigma(x) \), and further we let \( \ellaccesses_\sigma(x) = \ellreads_\sigma(x) \cup \ellwrites_\sigma(x) \). Similarly, we let \( \ellacquires_\sigma(\ell) \) and \( \ellreleases_\sigma(\ell) \) denote the set of lock-acquire and lock-release events, respectively, of \( \sigma \) on lock \( \ell \). The trace order of \( \sigma \), denoted \( \leq_{\sigma} \), is the total order on \( \ellevents_\sigma \) induced by the sequence \( \sigma \). Finally, the thread-order of \( \sigma \), denoted \( \leq_{TO} \), is the smallest partial order on \( \ellevents_\sigma \) such that for any two events \( e_1, e_2 \in \ellevents_\sigma \), if \( e_1 \leq_{\sigma} e_2 \) and \( \ellid(e_1) = \ellid(e_2) \), then \( e_1 \leq_{TO} e_2 \).
Traces are assumed to be well-formed in that critical sections on the same lock do not overlap. For a lock \( \ell \in \text{Locks}_\sigma \), let \( \sigma_\ell \) be the projection of the trace \( \sigma \) on the set of events \( \text{Acquires}_\sigma(\ell) \cup \text{Releases}_\sigma(\ell) \). Also, let \( t_1, \ldots, t_k \) be the thread identifiers in \( \text{Threads}_\sigma \). Well-formedness then entails that for each lock \( \ell \), the projection \( \sigma_\ell \) is a prefix of some string in the language of the grammar with production rules \( S \rightarrow \varepsilon | S \cdot S_1 | S \cdot S_2 | \cdots | S \cdot S_m \) and \( S_1 \rightarrow \langle t_1, \text{acq}(\ell) \rangle \cdot \langle t_1, \text{re1}(\ell) \rangle \) and start symbol \( S \). Thus, every release event \( e \) has a unique matching acquire event, which we denote by \( \text{match}_\sigma(e) \). Likewise for an acquire event \( e \), \( \text{match}_\sigma(e) \) denotes the unique matching release event if one exists. For an acquire event \( e \), the critical section of \( e \) is the set of events \( \text{CS}_\sigma(e) = \{ f | e \leq^\sigma_{\text{TO}} f \leq^\sigma_{\text{TO}} \text{match}_\sigma(e) \} \) if \( \text{match}_\sigma(e) \) exists, and \( \text{CS}_\sigma(e) = \{ f | e \leq^\sigma_{\text{TO}} f \} \) otherwise.

Data Races. Two events \( e_1, e_2 \in \text{Events}_\sigma \) 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) \( \text{tid}(e_1) \neq \text{tid}(e_2) \), (ii) \( e_1, e_2 \in \text{Accesses}_\sigma(x) \) for some \( x \in \text{Vars}_\sigma \), and (iii) \( \{ e_1, e_2 \} \cap \text{Writes}_\sigma(x) \neq \emptyset \). An event \( e \in \text{Events}_\sigma \) is said to be enabled in a prefix \( \rho \) of \( \sigma \), if for every event \( e' \neq e \) with \( e' \leq^\sigma_{\text{TO}} e \), we have \( e' \in \text{Events}_\rho \). A data race in \( \sigma \) is a pair of conflicting events \( (e_1, e_2) \) such that there is a prefix \( \rho \) in which both \( e_1 \) and \( e_2 \) are simultaneously enabled.

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. Here we describe in detail some popular approaches to dynamic race detection that are subject of this work.

Happens-Before Races. Given a trace \( \sigma \), the happens before order \( \leq_{\text{HB}}^\sigma \) is the smallest partial order on \( \text{Events}_\sigma \) such that
(a) \( \leq^\sigma_{\text{TO}} \leq_{\text{HB}}^\sigma \), and
(b) for any lock \( \ell \in \text{Locks}_\sigma \) and for events \( e \in \text{Releases}_\sigma(\ell) \) and \( f \in \text{Acquires}_\sigma(\ell) \), if \( e \leq_{\text{HB}}^\sigma f \), then \( e \leq_{\text{HB}}^\sigma f \).

A pair of conflicting events \( (e_1, e_2) \) is an HB-race in \( \sigma \) if they are unordered by HB, i.e., \( e_1 \not<_{\text{HB}}^\sigma e_2 \) and \( e_2 \not<_{\text{HB}}^\sigma e_1 \). The associated decision question is, given a trace \( \sigma \), determine whether \( \sigma \) 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 \([36, 2, 1]\). That is, they solve the following function problem: given a trace \( \sigma \), determine all events \( e_2 \in \text{Events}_\sigma \) for which there exists an event \( e_1 \in \text{Events}_\sigma \) such that \( e_1 \not<_{\text{HB}}^\sigma e_2 \), and \( (e_1, e_2) \) is an HB race of \( \sigma \). The standard algorithm for solving both versions of the problem is a vector-clock algorithm that runs in \( O(N \cdot T) \) time \([19]\).

Synchronization Preserving Races. Next, we present the notion of synchronization-preserving races \([25]\). For a trace \( \sigma \) and a read event \( e \), we use \( w_\sigma(e) \) to denote the write event observed by \( e \). That is, \( e' = w_\sigma(e) \) is the last (according to the trace order \( \leq_{\text{HB}}^\sigma \)) write event \( e' \) of \( \sigma \) such that \( e \) and \( e' \) access the same variable and \( e' \leq_{\text{HB}}^\sigma e \); if no such \( e' \) exists, then we write \( w_\sigma(e) = \perp \). A trace \( \rho \) is said to be a correct reordering of trace \( \sigma \), if
(a) \( \text{Events}_\rho \subseteq \text{Events}_\sigma \)
(b) \( \text{Events}_\rho \) is downward closed with respect to \( \leq^\rho_{\text{TO}} \leq_{\text{HB}}^\sigma \), and further \( \leq^\rho_{\text{TO}} \leq_{\text{HB}}^\sigma \leq^\rho_{\text{TO}} \), and
(c) for every read event \( e \in \text{Events}_\rho \), \( w_\rho(e) = w_\sigma(e) \).
Further, $\rho$ is sync-preserving with respect to $\sigma$ if for every lock $\ell$ and for any two acquire events $e_1, e_2 \in \text{Acquires}_\rho(\ell)$, we have $e_1 \leq^\rho e_2$ iff $e_1 \leq^\sigma e_2$. Thus, the order of critical sections on the same lock is the same in $\sigma$ and $\rho$.

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

**Lock-Cover and Lock-Set Races.** Lock-cover and lock-set races indicate violations of the locking discipline. For an event $e$ in a trace $\sigma$, let $\text{locksHeld}_\sigma(e) = \{\ell | \exists f \in \text{Acquires}_\sigma(\ell), \text{such that } e \in \text{CS}_\sigma(f)\}$, i.e., $\text{locksHeld}_\sigma(e)$ is the set of locks held by thread $\text{tid}(e)$ when $e$ is executed. A pair $(e_1, e_2)$ of conflicting events might indicate a data race if $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) = \emptyset$. Although this condition doesn’t 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 $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) = \emptyset$. The decision question is, given a trace $\sigma$, determine if $\sigma$ has a lock-cover race. The problem is solvable in $O(N^2 \cdot L)$ 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 [34]). A trace $\sigma$ has a lock-set race on variable $x \in \text{Vars}_\sigma$ if (a) there exists a pair of conflicting events $(e_1, e_2) \in \text{Writes}_\sigma(x) \times \text{Accesses}_\sigma(x)$, and (b) $\bigcap_{e \in \text{Accesses}_\sigma(x)} \text{locksHeld}_\sigma(e) = \emptyset$.

The associated decision question is, given a trace $\sigma$, determine if $\sigma$ has a lock-set race. Note that a lock-cover race implies a lock-set race, but not vice versa. On the other hand, determining whether $\sigma$ has a lock-set race is easily performed in $O(N \cdot L)$ 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 $\sigma_6$ in Figure 1a. The events $e_2$ and $e_4$ are conflicting and unordered by $\leq_{\text{HB}}$, thus $(e_2, e_4)$ is an HB-race. Second, in trace $\sigma_5$,
of Figure 1b, the pair \((e_1, e_6)\) is not an HB-race as \(e_1 \leq_{HB} e_6\). 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 \(\sigma_e\) 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 \(\text{locksHeld}_{\sigma_e}(e_2) \cap \text{locksHeld}_{\sigma_e}(e_7) = \emptyset\). Finally, the trace \(\sigma_d\) in Figure 1d has no HB, sync-preserving or lock-cover race, as all \(w(x)\) are protected by a common lock. But there is a lock-set race on \(x\) as there is no single lock that protects all \(w(x)\).

2.3 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 [43] for a detailed exposition on the topic. This theory relates the computational complexity of problems under the popular notion of fine-grained reductions (See Appendix A for a formal definition).

Such a reduction \(A \preceq_{\text{fgr}} B\) 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] (See Appendix A for a formal definition) for the classic CNF-SAT problem. 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 [42]. Thus, a conditional lower bound under OVH implies one under SETH as well, leading to numerous conditional lower bound results under OVH [See [43] for a detailed literature review]. We next formally define \(k\)-OV and OVH.

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

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

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

- **Hypothesis 1** (Orthogonal Vectors Hypothesis (OVH)). No randomized algorithm can solve \(k\)-OV for an instance \(\text{OV}(n,d,k)\) in time \(O(n^{(k-\epsilon)} \cdot \text{poly}(d))\) for any constant \(\epsilon > 0\).

The following impossibility result from [9] proves that a reduction under SETH, and hence under OVH, is not possible unless the \(\text{NSETH}\) conjecture (definition included in Appendix A) is false.

- **Theorem 10**. If \(\text{NSETH}\) holds and a problem \(C \in \text{NTIME}[T_C] \cap \text{coNTIME}[T_C]\), 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 \(B \preceq_{\text{fgr}} C\) where \(c = T_C^{(1+\gamma)}\).

We show some of our problems satisfy the conditions of Theorem 10, and hence show lower bounds for these conditioned on two other hypotheses 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 \(\text{HS}(n,d)\).

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

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

Finally we consider the subclass of first order formulae over structures of size \(n\) and with \(m\) relational tuples [17].
Problem 3 (FO(∀∃∃)). Decide if a given a first-order formula quantified by ∀∃∃ has a model on a structure of size n with m relational tuples.

It is known that FO(∀∃∃) can be solved in $O(m^{3/2})$ time using ideas from triangle detection algorithms [17]. For dense graph 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.

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

We first outline our $O(N \cdot L)$-time algorithm for checking if a trace $\sigma$ 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 $\sigma$ in the rest of the discussion.

Lockstamps. A lockstamp is a mapping from locks to natural numbers (including infinity) $L : \text{Locks}_\sigma \rightarrow \mathbb{N} \cup \{\infty\}$. Given lockstamps $L, L_1, L_2$ and lock $\ell$, we use the notation

(i) $L[\ell \mapsto c]$ to denote the the lockstamp $\lambda_m$ if $m = \ell$ then $c \ell$ 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 $\ell$,

(iii) $L_1 \sqcap L_2$ to denote the pointwise minimum, and

(iv) $L_1 \sqsubseteq L_2$ to denote the predicate $\forall \ell. L_1(\ell) \leq L_2(\ell)$.

Our algorithm computes acquire and release lockstamps $\text{AcqLS}_e^\sigma$ and $\text{RelLS}_e^\sigma$ for every event $e \in \text{Events}_\sigma$, defined next. For a lock $\ell$ and acquire event $f \in \text{Acquires}_\sigma(\ell)$ (resp. release event $g \in \text{Releases}_\sigma(\ell)$), let $\text{pos}_\sigma(f) = \{|f' \in \text{Acquires}_\sigma(\ell) | f' \leq f\}$ (resp. $\text{pos}_\sigma(g) = \{|g' \in \text{Releases}_\sigma(\ell) | g' \leq g\}$) denote the relative position of $f$ (resp. $g$) among all acquire events (resp. release events) of $\ell$. Then, for an event $e \in \text{Events}_\sigma$ the lockstamps $\text{AcqLS}_e^\sigma$ and $\text{RelLS}_e^\sigma$ are defined as follows (we assume that $\exists \sigma = 0$ and $\min \sigma = \infty$.)

\[
\begin{align*}
\text{AcqLS}_e^\sigma(\ell) & = \lambda \ell \cdot \max\{|\text{pos}_\sigma(f) | f \in \text{Acquires}_\sigma(\ell), f \leq \sigma e\} \\
\text{RelLS}_e^\sigma(\ell) & = \lambda \ell \cdot \min\{|\text{pos}_\sigma(g) | g \in \text{Releases}_\sigma(\ell), e \leq \sigma g\} 
\end{align*}
\]  

Our $O(N \cdot L)$ algorithm now relies on the following observations. First, the HB partial order can be inferred by comparing lockstamps of events (Lemma 11). Second, there is an $O(N \cdot L)$ 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 $O(N)$ pairs of conflicting events (using their lockstamps), instead of all possible $O(N^2)$ pairs (Lemma 12). Finally, we can also examine all the $O(N)$ pairs in time $O(N \cdot L)$ (using $O(N)$ 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 11. Let $e_1 \leq^\sigma e_2$ be events in $\sigma$ such that $\text{tid}(e_1) \neq \text{tid}(e_2)$. We have, $e_1 \leq^\text{HB} e_2 \iff \neg(\text{AcqLS}_{e_2}^\sigma \subseteq \text{RelLS}_{e_1}^\sigma)$

The proof of Lemma 11 is presented in Appendix B.1.
Computing Lockstamps. We now illustrate how to compute the acquire lockstamps for all events, by processing the trace $\sigma$ in a forward pass. For each thread $t$ and lock $\ell$, we maintain lockstamp variables $C_t$ and $L_\ell$. We also maintain an integer variable $p_\ell$ for each lock $\ell$ that stores the index of the latest acquire($\ell$) event in $\sigma$. Initially, we set $C_t$ and $L_\ell$ to the bottom map $\ell \cdot 0$, and $p_\ell$ to 0, for each thread $t$ and lock $m$. We traverse $\sigma$ 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 $e = (t, op)$. At the end of each handler, we assign the lockstamp $\text{AcqLS}_e^x$ 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 $O(L)$ time giving us a total running time of $O(N \cdot L)$ to assign lockstamps.

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acquire($t$, $\ell$):</td>
</tr>
<tr>
<td>2</td>
<td>$p_\ell \leftarrow p_\ell + 1$</td>
</tr>
<tr>
<td>3</td>
<td>$C_t \leftarrow C_t[\ell \mapsto p_\ell] \cup L_\ell$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{AcqLS}_e^x \leftarrow C_t$</td>
</tr>
<tr>
<td>5</td>
<td>release($t$, $\ell$):</td>
</tr>
<tr>
<td>6</td>
<td>$L_\ell \leftarrow C_t$</td>
</tr>
<tr>
<td>7</td>
<td>$\text{AcqLS}_e^x \leftarrow C_t$</td>
</tr>
<tr>
<td>8</td>
<td>read($t$, $x$):</td>
</tr>
<tr>
<td>9</td>
<td>$\text{AcqLS}_e^x \leftarrow C_t$</td>
</tr>
<tr>
<td>10</td>
<td>write($t$, $x$):</td>
</tr>
<tr>
<td>11</td>
<td>$\text{AcqLS}_e^x \leftarrow C_t$</td>
</tr>
</tbody>
</table>

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

**Lemma 12.** A trace $\sigma$ has an HB-race iff there is pair of consecutive conflicting events in $\sigma$ that is an HB-race. Moreover, $\sigma$ has at most $O(N)$ many consecutive conflicting pairs of events.

Checking for an HB race. We now describe the algorithm for checking for an HB race in $\sigma$. We perform a forward pass on $\sigma$ 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 event by comparing the acquire lockstamp of $e$ with a previously stored release lockstamp. More precisely, we maintain a variable $W_x$ 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, $t_w^x = \text{NIL}$, $W_x = \lambda \ell \cdot \infty$, and $S_x = \emptyset$. The updates performed at each read or write event $e$ are presented in the corresponding handler in Algorithm 2; no updates need to be performed at acquire or release events in this case.

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read($t$, $x$):</td>
</tr>
<tr>
<td>2</td>
<td>if $t_w^x \notin {\text{NIL}, t} \wedge \text{AcqLS}_e^x \subseteq W_x$ then</td>
</tr>
<tr>
<td>3</td>
<td>declare “race” and exit</td>
</tr>
<tr>
<td>4</td>
<td>$S_x \leftarrow S_x \cup {(t, \text{RelLS}_e^x)}$</td>
</tr>
<tr>
<td>5</td>
<td>write($t$, $x$):</td>
</tr>
<tr>
<td>6</td>
<td>if $t_w^x \notin {\text{NIL}, t} \wedge \text{AcqLS}_e^x \subseteq W_x$ then</td>
</tr>
<tr>
<td>7</td>
<td>declare “race” and exit</td>
</tr>
<tr>
<td>8</td>
<td>if $\exists (u, L) \in S_x, t \neq u \wedge \text{AcqLS}_e^x \subseteq L$ then</td>
</tr>
<tr>
<td>9</td>
<td>declare “race” and exit</td>
</tr>
<tr>
<td>10</td>
<td>$t_w^x = t; S_x \leftarrow \emptyset; W_x \leftarrow \text{RelLS}_e^x$</td>
</tr>
</tbody>
</table>

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 $G^n_{\text{HB}}$, which can be thought of as a form of transitive reduction of the HB relation. For an integer $n \geq 1$, we use $[n]$ to denote $\{1, \ldots, n\}$.

The graph $G^n_{\text{HB}}$. Given a trace $\sigma$, the graph $G^n_{\text{HB}}$ is a directed graph with node set $\text{Events}_\sigma$, and we have an edge $(e_1, e_2)$ in $G^n_{\text{HB}}$ if

(i) $e_2$ is the immediate successor of $e_1$ in thread order $\leq^\sigma_{\text{TO}}$, or
(ii) $e_1 \in \text{Acquires}_\sigma(\ell)$, $e_2 \in \text{Releases}_\sigma(\ell)$, $e_1 \leq^\sigma_\sigma e_2$, and there is no intermediate event in $\sigma$ that accesses the common lock $\ell$.

It follows easily that for any two distinct events $e_1, e_2$, we have $e_1 \leq^\sigma_{\text{HB}} e_2$ iff $e_2$ is reachable from $e_1$ in $G^n_{\text{HB}}$. Moreover, every node has out-degree $\leq 2$ and thus $G^n_{\text{HB}}$ is sparse, while it can be easily constructed in $O(N)$ time.

OV hardness of write-read HB races

Given a OV instance $\text{OV}(n, d)$ on two vector sets $A_1, A_2$, we create a trace $\sigma$ as follows. For the part $A_1$ of OV, we introduce $n(d+1)$ threads $\{t_{x,i}\}_{x \in A_1, i \in [d]}$, and $d$ locks $\{l_i\}_{i \in [d]}$. For the second part $A_2$ we introduce $nd$ locks denoted by $\{l_{y,i}\}_{y \in A_2, i \in [d]}$, and $n$ threads $\{t_y\}_{y \in A_2}$. Finally, we have a single variable $z$.

We first describe the threads $t(x, i)$. For each vector $x$, for each $i \in [d]$ with $x[i] = 1$, 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 $w(z)$, 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 $w(z)$ event is ordered by HB before all other events of $t(x, i)$.

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 $r(z)$.

Finally, we construct $\sigma$ 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 2 for an illustration.

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

Conditional impossibility for SETH-based hardness

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Reducing OV to finding HB races. For simplicity, we show the graph $G^n_{\text{HB}}$ instead of the trace $\sigma$. The HB race is marked in red, corresponding to the orthogonal pair $(x_2, y_2)$.}
\end{figure}
Problem 4 (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 12, detecting whether there is an HB race in \( \sigma \) reduces to testing MCONN between all \( O(N^2) \) pairs of consecutive conflicting events in \( \sigma \).

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 \( \text{NTIME}[N^{3/2}] \cap \text{coNTIME}[N^{3/2}] \). At a first glance, the bound \( \text{NTIME}[N^{3/2}] \) may seem too optimistic, as there are \( \Theta(N) \) paths \( P_i: s_i \leadsto t_i \), and each of them can have size \( \Theta(N) \). 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 \( \sigma \) has an HB-race. Phrased as a MCONN problem on \( G^\sigma_{HB} \), 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 \( s_i \leadsto t_i \) path. Since \( G^\sigma_{HB} \) is sparse, this can be easily verified in \( O(N^2) \) time.

Now consider the case when there is no HB-race. Phrased as a MCONN problem on \( G^\sigma_{HB} \), 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 \( A = \{(s_i, t_i)\}_{i \in \mathbb{N}} \).

1. In the first phase, the algorithm repeatedly guesses a node \( u \) that lies on at least \( N^{1/2} \) paths \( s_i \leadsto t_i \), 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 \( P_i: s_i \leadsto t_i \), and verifies that \( P_i \) is a valid path.

Phase 1 can be execute at most \( N^{1/2} \) iterations, while each iteration takes \( O(N) \) time since \( G^\sigma_{HB} \) is sparse. Hence the total time for phase 1 is \( O(N^{3/2}) \). Phase 2 takes \( O(N^{3/2}) \) time, as every node of \( G^\sigma_{HB} \) appears in at most \( N^{1/2} \) 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 \( \text{FO}(\forall \exists \exists) \) takes as input a first-order formula \( \phi \) with quantifier structure \( \forall \exists \exists \) and whose atoms are tuples, and the task is to verify whether \( \phi \) 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 \( \phi \) a formula that characterizes the presence/absence of edges (e.g., \( \phi = \forall x \exists y \exists z \ e(x, y) \land \neg e(y, z) \)).

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

Lemma 13. \( \text{FO}(\forall \exists \exists) \) reduces to MCONN on a graph \( G \) with \( O(n^2) \) nodes in \( O(n^2) \) time.

Finally, we arrive at Theorem 3 by constructing in \( O(n^2) \) time a trace \( \sigma \) with \( N = \Theta(n^2) \) such that \( G^\sigma_{HB} \) is similar in structure to the graph \( G \) of Lemma 13. In the end, detecting an HB race in \( \sigma \) in \( O(N^{1+\epsilon}) \) time yields an algorithm for \( \text{FO}(\forall \exists \exists) \) in \( \Theta(n^{2+\epsilon}) \) 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 \textit{sync}(\ell) event represents the sequence \textit{acq}(\ell), r(x_\ell), w(x_\ell), \textit{rel}(\ell). The key ideas behind the sync events are as follows. First, if \( X_\ell \) appears only in \textit{sync}(\ell) events, then there can be no race involving these. Second, assume that in a trace \( \sigma \) we have two \textit{sync}(\ell) events \( e_1 \) and \( e_2 \) with \( e_1 <^\sigma e_2 \). Then any correct reordering \( \rho \) of \( \sigma \) with \( e_2 \in \text{Events}_\rho \) satisfies the following.
For every (b) We have \( v_1 \in \text{Events}_\sigma \). This is because, for any two consecutive \( \text{sync}(e) \) events \( e \leq_T \ e', \)
\[ w_\sigma(e') = e_\sigma, \] where \( e_\sigma \) is the \( r(x_\ell) \) event in the sync sequence \( e \), and \( e_\sigma \) is the \( w(x_\ell) \) event in the sync sequence \( e \).

(b) For every \( e'_1, e'_2 \in \text{Events}_\sigma \) such that \( e'_1 \leq_T \ e_1 \) and \( e_2 \leq_T e'_2 \), we have \( e'_1 <_T e'_2 \).

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

**Informal Description**

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 \( \text{acq}(X) \), \( w(z) \), \( \text{rel}(X) \). Similarly, every vector \( y \in A_2 \) is represented by a thread \( t_y \), ending with the critical section \( \text{acq}(Y) \), \( r(z) \), \( \text{rel}(Y) \). There are no further access events, hence we can only have a race between the write event of a thread \( t_x \) and the read event of a thread \( t_y \). To encode the vectors in \( A_3 \), we use \( k \) threads \( t_k \), for \( k \in [d] \). Each thread \( t_k \) has \( n \) segments such that the \( i \)-th segment of \( t_k \) encodes \( z_i[k] \). Finally, there is a single thread \( t \) that will have the property (enforced using sync events) that it must be included in any reordering if and only if all the threads encoding \( A_3 \) are included entirely. The thread \( t \) also has locks that, if present in a reordering, prevent the access events of \( z \) from being in race with each other.

If there is a triplet of vectors \( x \in A_1 \), \( y \in A_2 \) and \( z \in A_3 \) that is orthogonal, then a valid reordering of the trace \( \sigma \) need only contain the threads corresponding to \( A_3 \) up to the events of \( z \); the thread \( t \) is not required to be a part of this reordering, causing the events of \( x \) and \( y \) to be in race. If no such triple exists, then the notion of sync-preservation ensures that all events of the threads representing \( A_3 \) must be present in any valid reordering of \( \sigma \), thus enforcing \( t \) also to be a part of such a reordering. Thus, the access events belonging to some threads \( t_x \) and \( t_y \) will be in race if and only if there is a vector \( z \in A_3 \) that makes the triplet \( x, y, z \) orthogonal.

**Reduction**

Given a 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 \( \sigma \) as follows (see Figure 3). We have \( T = 2 \cdot n + d + 1 \) threads, while all access events (not counting the sync events) are of the form \( w(z)/r(z) \) in a single variable \( z \). We first describe the threads, and then how they interleave in \( \sigma \).
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 $\text{acq}(l_k), \text{rel}(l_k)$ in $t_x^1$. We create $t_x^2$ as the sequence $\text{acq}(X), w(z), \text{rel}(X)$, 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 $\text{acq}(l_k), \text{rel}(l_k)$ in $t_y^1$. In contrast to the $t_x^1$, every $t_y^1$ also has an event $\text{sync}(y_y)$ at the very beginning. We create $t_y^2$ as the sequence $\text{acq}(Y), r(z), \text{rel}(Y)$, 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 $\text{sync}(l_k), \text{sync}(l_k)$ to $t_k$. On the other hand, if $z_i[k] = 1$, we interleave these sync events with two critical sections, by appending the sequence $\text{acq}(l_k), \text{sync}(l_k), \text{acq}(l_k), \text{rel}(l_k), \text{sync}(l_k), \text{rel}(l_k)$.

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

\[
\begin{align*}
t^1 &= \text{sync}(l_1), \ldots, \text{sync}(l_k), \text{sync}(s_{y_1}), \ldots, \text{sync}(s_{y_n}) \\
t^2 &= (\text{acq}(Y), \text{sync}(l_1), \ldots, \text{sync}(l_k), \text{sync}(l_1), \ldots, \text{sync}(l_k), \text{rel}(Y))^{n-1} \\
t^3 &= \text{acq}(Y), \text{sync}(l_1), \ldots, \text{sync}(l_k), \text{acq}(X), \text{rel}(X), \text{rel}(Y)
\end{align*}
\]

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

1. We execute the auxiliary thread $t$ and all threads $t_k$, 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 $\text{sync}(l_k)$ events
   (a) starts with the $\text{sync}(l_k)$ event of $t_k$ and proceeds with the $\text{sync}(l_k)$ event of $t$,
   (b) strictly alternates in every two $\text{sync}(l_k)$ events between $t$ and $t_k$, and
   (c) ends with the last $\text{sync}(l_k)$ event of $t_k$.

2. We execute all $t_x^1$ and $t_y^1$ (i.e., the first parts of all threads that correspond to the vectors in $A_1$ and $A_2$) arbitrarily, but after all threads $t_k$, for $k \in [d]$.

3. We execute all $t_x^2$ (i.e., the second parts of all threads that correspond to the vectors in $A_1$) arbitrarily, but before the segment $\text{acq}(X), \text{rel}(X), \text{rel}(Y)$ of $t$.

4. We execute all $t_y^2$ (i.e., the second parts of all threads that correspond to the vectors in $A_2$) arbitrarily, but after the segment $\text{acq}(X), \text{rel}(X), \text{rel}(Y)$ of $t$.

We refer to the full paper [22] 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 $\text{OV}(n,d)$ on two vector sets $A_1, A_2$, we create a trace $\sigma$ 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 $e = (t_i, w(x))$. 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 $\sigma$ 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 [22] 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 $\sigma$ 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 $\sigma$. The more involved part is in computing the lock-set of $x$, i.e., the set $\bigcap_{e \in \text{Accesses}_e(x)} \text{locksHeld}_e(e)$, in linear time. Indeed, each intersection alone requires $\Theta(L)$ time, resulting to $\Theta(N \cdot L)$ time overall.

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

$$A = \text{locksHeld}_e(e) \quad B = \text{Locks}_e \cap \bigcap_{e' \in \text{Accesses}_e(x), e' \leq e} \text{locksHeld}_e(e') \quad C = A \cap B$$

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

### Algorithm 3 Computing the lock-set of variable $x$.

1. acquire$(t, \ell)$: 5. release$(t, \ell)$: 9. read$(t, y)$: 13. write$(t, y)$:
2. $A \leftarrow A \cup \{\ell\}$ 6. $A \leftarrow A \setminus \{\ell\}$ 10. if $x = y$ then
3. if $\ell \in B$ then 7. if $\ell \in B$ then 11. $B \leftarrow B \setminus C$
4. $C \leftarrow C \setminus \{\ell\}$ 8. $C \leftarrow C \cup \{\ell\}$ 12. $C \leftarrow \emptyset$
5. if $x = y$ then

The correctness of Algorithm 3 follows by proving the invariant in Equation (2). We refer to [22] 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 $\text{NTIME}[\mathcal{N}] \cap \text{coNTIME}[\mathcal{N}]$, and we can thus use [9, Corollary 2] to prove Theorem 7.

Proof of Theorem 7. First we argue that the problem is in $\text{NTIME}[\mathcal{N}]$. 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 $O(N)$ time.

Now we argue that the problem is in $\text{coNTIME}[\mathcal{N}]$, by giving a certificate to verify in linear time that $\sigma$ does not have a race of the required form. The certificate has size $O(|\text{Vars}_\sigma|)$, 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 $\sigma$.

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 $\text{HS}(n,d)$ on two vector sets
Dynamic Data-Race Detection Through the Fine-Grained Lens

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 HS 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


16:18 Dynamic Data-Race Detection Through the Fine-Grained Lens


A Fine-grained Complexity

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 (a)\(\leq_{\text{B}}\) 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 \(n_{1}, \ldots, n_{q}\), such that, \(\sum_{i=1}^{q}(b(n_{i}))^{1-\epsilon} \leq (a(n))^{1-\epsilon}\).

\[\begin{align*}
\text{Hypothesis 3} \ (\text{Strong Exponential Time Hypothesis (SETH)).} \quad & \forall \epsilon > 0 \ \exists k \geq 3 \ \text{such that CNF-SAT on formulas with clause size at most k and n variables cannot be solved in O}(2^{((1-\epsilon)n)}) \ \text{time even by a randomized algorithm.} \\
\text{Hypothesis 4} \ (\text{Non-deterministic SETH (NSETH)).} \quad & \forall \epsilon > 0 \ \exists k \text{ such that k-TAUT is not in NTIME}[2^{n(1-\epsilon)}], \ \text{where k-TAUT is the language of all k-DNF formulas which are tautologies.}
\end{align*}\]

B Proofs of Section 3

B.1 Proofs from Section 3.1

\[\begin{align*}
\text{Lemma 11.} \quad & \text{Let } e_{1} \leq_{\text{HB}} e_{2} \text{ be events in } \sigma \text{ such that } \text{tid}(e_{1}) \neq \text{tid}(e_{2}). \ \text{We have, } e_{1} \leq_{\text{HB}} e_{2} \iff \neg(\text{AcqLS}_{e_{2}}^{\sigma} \subseteq \text{RelLS}_{e_{1}}^{\sigma})
\end{align*}\]

Proof. \((\Rightarrow)\) Let \(e_{1} \leq_{\text{HB}} e_{2}\). Using the definition of \(\leq_{\text{HB}}\), there must be a sequence of events \(f_{1}, f_{2}, \ldots, f_{k}\) with \(k > 1\), \(f_{1} = e_{1}\), \(f_{k} = e_{2}\), and for every \(1 \leq i < k\), \(f_{i} \leq_{\text{HB}} f_{i+1}\) and either \(f_{i} \leq_{\text{HB}} f_{i+1}\) or there is a lock \(\ell\), such that \(f_{i} \in \text{Releases}_{\sigma}(\ell)\) and \(f_{i+1} \in \text{Acquires}_{\sigma}(\ell)\). Let \(j\) be the smallest index \(i\) such that \(\text{tid}(f_{i}) \neq \text{tid}(f_{i+1})\); such an index exists as \(\text{tid}(e_{1}) \neq \text{tid}(e_{2})\). Observe that there must be a lock \(\ell\) for which \(\text{op}(f_{j}) = \text{rel}(\ell)\) and \(\text{op}(f_{j+1}) = \text{acq}(\ell)\). Observe that \(\text{pos}_{\sigma}(f_{j}) < \text{pos}_{\sigma}(f_{j+1})\), \(\text{RelLS}_{e_{1}}^{\sigma}(\ell) \leq \text{pos}_{\sigma}(f_{j})\) and \(\text{pos}_{\sigma}(f_{j+1}) \leq \text{AcqLS}_{e_{2}}^{\sigma}(\ell)\), giving us \(\text{RelLS}_{e_{1}}^{\sigma}(\ell) < \text{AcqLS}_{e_{2}}^{\sigma}(\ell)\).

\((\Leftarrow)\) Let \(\ell\) be a lock such that \(\text{RelLS}_{e_{1}}^{\sigma}(\ell) < \text{AcqLS}_{e_{2}}^{\sigma}(\ell)\). Then, there is a release event \(f\) and an acquire event \(g\) on lock \(\ell\) such that \(\text{pos}_{\sigma}(f) < \text{pos}_{\sigma}(g), e_{1} \leq_{\text{HB}} f\) and \(g \leq_{\text{HB}} e_{2}\). This means \(f \leq_{\text{HB}} g\) and thus \(e_{1} \leq_{\text{HB}} e_{2}\). \(\blacksquare\)
For the sake of completeness, we present the computation of release lockstamps. The computation of release lockstamps takes place in the reverse order of $\leq \sigma^t$ (i.e., from right to left), unlike the case of acquire lockstamps. As with Algorithm 1, we maintain the following variables. For each thread $t$ and lock $\ell$, we will maintain variables $C_t$ and $L_\ell$ that take values from the space of all lockstamps. We also additionally maintain an integer variable $p_\ell$ for each lock $\ell$ that stores the index (or relative position) of the earliest (according to the trace order $\leq \sigma^t$) release event of lock $\ell$ in the trace suffix seen so far. Initially, we set $C_t$ and $L_m$ 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 in reverse, 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 $RelLS^t_\ell$ to the event $e$.

Algorithm 4 Assigning release lockstamps to events in the trace. Events are processed in reverse order.

\begin{align*}
&1 \text{ acquire}(t, \ell): & 4 \text{ release}(t, \ell): & 8 \text{ read}(t, x): \\
&2 \quad L_\ell \gets C_t & 5 \quad p_\ell \gets p_\ell - 1 & 9 \quad RelLS^t_\ell \gets C_t \\
&3 \quad RelLS^t_\ell \gets C_t & 6 \quad C_t \gets C_t[\ell \mapsto p_\ell] \cap L_\ell & 10 \quad write(t, x): \\
& & 7 \quad RelLS^t_\ell \gets C_t & 11 \quad RelLS^t_\ell \gets C_t
\end{align*}

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

Lemma 14. On input trace $\sigma$, Algorithm 1 and Algorithm 4 correctly compute the lockstamps $AcqLS^t_\ell$ and $RelLS^t_\ell$ respectively for each event $e \in \text{Events}_\sigma$.

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 $C_t$, $L_\ell$ and $p_\ell$ for each thread $t$ and lock $\ell$, which we state next. Let $\pi$ be the prefix of the trace processed at any point in the algorithm. Let $C^\pi_t$, $L^\pi_\ell$ and $p^\pi_\ell$ be the values of the variables $C_t$, $L_\ell$ and $p_\ell$ after processing the prefix $\pi$. Then, the following invariants are true:

- $C^\pi_t = AcqLS^\pi_t = AcqLS^\pi_\ell$, where $e^\pi_\ell$ is the last event in $\pi$ performed by thread $t$.
- $L^\pi_\ell = AcqLS^\pi_\ell = AcqLS^\pi_\ell$, where $e^\pi_\ell$ is the last acquire event on lock $\ell$ in $\pi$.
- $p^\pi_\ell = pos^\pi(e^\pi_\ell)$, where $e^\pi_\ell$ is the last acquire event on lock $\ell$ in $\pi$.

These invariants can be proved using a straightforward induction on the length of the trace, each time noting the definition of $\leq^\sigma HB$.

Lemma 15. For a trace with $N$ events and $L$ locks, Algorithm 1 and Algorithm 4 both take $O(T \cdot L)$ 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 $p_\ell$, $O(L)$ time for doing the $\cup$ operation, and $O(L)$ time for the copy operation ($AcqLS^t_\ell \gets C_t$). For a release event, we spend $O(L)$ for the two copy operations. At read and write events, we spend $O(L)$ for copy operations. This gives a total time of $O(N \cdot L)$.

Lemma 12. A trace $\sigma$ has an $HB$-race iff there is pair of consecutive conflicting events in $\sigma$ that is an $HB$-race. Moreover, $\sigma$ has at most $O(\hat{N})$ many consecutive conflicting pairs of events.
Proof. We first prove that if there is a an HB-race in $\sigma$, 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 $e_2 \leq_{\text{HB}} e'_2$ or $e_2 = e'_2$ and $e'_1 \leq_{\text{HB}} e_1$. We remark that such a race $(e_1, e_2)$ exists if $\sigma$ 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 $f \in \text{Writes}_\sigma(x)$ such that $e_1 <_{\text{HB}} f <_{\text{HB}} e_2$. 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 $\sigma$. Thus, $e_1 \leq_{\text{HB}} f$ and $f \leq_{\text{HB}} e_2$, which gives $e_1 \leq_{\text{HB}} e_2$, another contradiction.

We now turn our attention to the number of consecutive conflicting events in $\sigma$. 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 $2N$ consecutive conflicting pairs of events.

Let us now state the correctness of Algorithm 2.

Lemma 16. For a trace $\sigma$, Algorithm 2 reports a race iff $\sigma$ 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, $e_2^\sigma$ is the last event with $\text{op}(e_2^\sigma) = w(x)$ in a trace $\pi$.

- After processing the prefix $\pi$ of $\sigma$, $t_y^\pi = \text{tid}(e_2^\sigma)$ and $W_y = \text{RelLS}_{e_2^\sigma}^\sigma$.
- After processing the prefix $\pi$ of $\sigma$, the set $S_x$ is $\{(\text{tid}(e), \text{RelLS}_{e}^\sigma) | e \in \text{Reads}_\pi(x), e_2^\sigma \leq_{\text{HB}} e\}$. The rest of the proof follows from Lemma 11 and Lemma 12.

Let us now characterize the time complexity of Algorithm 2.

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

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

Theorem 4. Deciding whether $\sigma$ has an HB race can be done in time $O(N \cdot \min(T, L))$.

Proof. We focus on proving that there is an $O(N \cdot L)$ time algorithm, as the standard vector-clock algorithm [19] for checking for an HB-race runs in $O(N \cdot T)$ time. Our algorithm’s correctness is stated in Lemma 14 and Lemma 16 and its total running time is $O(N \cdot L)$ (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 $O(N^{2-\epsilon})$, unless the OV hypothesis fails.

Proof. Consider a pair of events $w(z)$ from the $d$ threads $t(x, i), i \in [d]$, and $r(z) \in t_y$ for some $x, i, y$. We have $w(z) \leq_{\text{HB}} r(z)$ iff there is some path from $w(z)$ to $r(z)$ in $G_{\text{HB}}$. As $w(z)$ and $r(z)$ 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 \( \mathbf{w}(z) \) is in \( t(x', i) \), we are done. Otherwise the only common lock between these threads \( t(x', i) \) and those of \( \mathbf{w}(z) \) can be one of the \( l_i \). The threads of \( \mathbf{w}(z) \) 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 \( \mathbf{w}(z) \) to \( \mathbf{r}(z) \) 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 \( \mathbf{OV} \) thus corresponds to a write-read \( \text{HB} \)-race in the reduced trace.

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

\begin{lemma}
\( \text{FO} (\forall \exists \exists) \) reduces to \( \text{MCONN} \) on a graph \( G \) with \( O(n) \) nodes in \( O(n^2) \) time.
\end{lemma}

\begin{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., \( \phi = \forall x \exists y \exists z (\psi_1 \lor \psi_2 \lor \ldots \psi_k) \), where \( x, y, z \) represent nodes of the graph, and each \( \psi_i \) is a conjunction of 3 variables representing edges of the graph, for example \( e(x, y) \land \neg e(y, z) \land e(x, z) \). 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 \( \phi \) is defined by \( H(I, J) \), where \( I \) and \( J \) are respectively the sets of nodes and edges of \( H \).

The instance of \( \text{MCONN} \) is constructed given \( H \) and \( \phi \) as follows. Construct a \((2k + 2)\)-partite graph \( G(V, E) \) by first creating \( 2k + 2 \) copies of \( I \). Denote these copies by \( S, Y_1, Z_1, T, t \), \( i \in [k] \), and the copy of each node \( x \in I \) in any part, say \( S \), by \( x(S) \). \( \psi_i = (e_1 \land e_2 \land e_3) \) 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 \( x, y \), \((x, y) \) \( \in J \) \( \iff \) \((x(S), y(Y_i)) \) \( \in E \). 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., \((x, y) \not\in J \) \( \iff \) \((x(S), y(Y_i)) \not\in E \).

Finally define \( |I| \) pairs \((x(S), x(T))\) as the \((s, t)\) pairs for \( \text{MCONN} \).

We now prove this reduction is correct. First, assume \( \phi \) is true. Then for every node \( x \), there exist nodes \( y, z \) such that some predicate is true. If \( \psi_i \) 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 \((x, y) \not\in J \) 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 \( \text{MCONN} \) 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 \( \psi_i \) 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 \( \phi \) 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 \( |G| = O(m+n+(n^2-m)+n) = O(n^2) \).
Theorem 3. For any $\epsilon > 0$, if there is an algorithm for detecting any HB race in time $O(N^{1+\epsilon})$, then there is an algorithm for $\text{FO}(\forall\exists\exists)$ formulas in time $O(m^{1+\epsilon})$.

Proof. We first reduce the instance of $\text{FO}(\forall\exists\exists)$ to $\text{MCONN}$ as in the proof of Lemma 13. Let $G(V, E)$ be the multi-partite graph for $\text{MCONN}$ 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 $\sigma$ 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 $\sigma$, and the only HB-race events can be the pairs of write events corresponding to the pairs of nodes given as input to $\text{MCONN}$. 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 $|\sigma|$ 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 $\sigma$. If deciding if the given trace has an HB-race has an $O((n^2)^{1+\epsilon})$ time algorithm, then $\text{FO}(\forall\exists\exists)$ can be solved in $O(n^{2+\epsilon})$ time, which is $O(n^{1+\epsilon'})$ time for properties on dense structures.

Due to space constraints, we include the remaining proofs of the Theorems from Sections 4 and 5 in the full paper [22].