What's the Optimal Performance of Precise Dynamic Race Detection? – A Redundancy Perspective*

Jeff Huang[1] and Arun K. Rajagopalan[2]

1 **Texas A&M University, US**
`jeff@cse.tamu.edu`
2 **Texas A&M University, US**
`arunxls@tamu.edu`

Abstract

In a precise data race detector, a race is detected only if the execution exhibits a real race. In such tools, every memory access from each thread is typically checked by a happens-before algorithm. What's the optimal runtime performance of such tools? In this paper, we identify that a significant percentage of memory access checks in real-world program executions are often redundant: removing these checks affects neither the precision nor the capability of race detection. We show that if all such redundant checks were eliminated with no cost, the optimal performance of a state-of-the-art dynamic race detector, FastTrack, could be improved by 90%, reducing its runtime overhead from 68X to 7X on a collection of CPU intensive benchmarks. We further develop a purely dynamic technique, ReX, that efficiently filters out redundant checks and apply it to FastTrack. With ReX, the runtime performance of FastTrack is improved by 31% on average.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Data Race Detection, Dynamic Analysis, Concurrency, Redundancy

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.15

1 Introduction

In recent years, the performance of precise Happens-Before (HB) based dynamic race detectors has greatly improved thanks to techniques such as FastTrack [12]. For many small-scale programs their performance is now close to that of the imprecise LockSet-based tools [27]. This is primarily due to the recent epoch-based advancement [12] that greatly cuts down the size of the vector clocks from $O(N_{threads})$ to almost $O(1)$, where $N_{threads}$ is the number of threads. However, it remains difficult to scale these tools to large software applications with a large number of threads. The reason is that these tools must still check races and maintain states for all memory accesses, the complexity of which is in $O(N_{events})$, *i.e.*, the number of memory accesses. Because N_{events} can be as large as the dynamic instruction count, it essentially dominates the race detection scalability. In our experiments, for instance, FastTrack incurs 80X runtime slowdown on the Java Grande benchmark suite [1].

What is the optimal performance of precise dynamic race detection? Is the FastTrack algorithm the best we can do? Due to the significance of race detection, this question is highly important to answer and in fact has attracted many researchers [6, 20, 34, 10, 30, 14, 17]. One promising way to gain further performance is to reduce the size of N_{events}. Along this

* This work was supported by a Google Faculty Research Award and NSF award CCF-1552935.

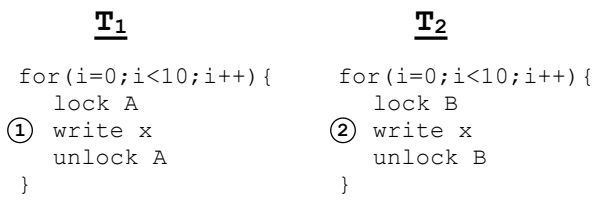

Figure 1 An example exhibiting redundancy.

direction, previous research has explored two ideas: dynamic sampling [6, 20, 34] and static analysis [26, 14, 10, 4]. However, none of them is sound or precise. Dynamic sampling techniques generally reduce the race detection capability because sampling may lead to missing races. While static analysis can be used to coalesce multiple checks [26, 10, 4] or prune memory checks that no race will be found [14], it is hard to obtain a sound static analysis in practice, especially for large complex applications with dynamic features, e.g., dynamic code generation and reflection; hence static analysis may lead to both missing races and imprecise results.

In this paper, we attempt to answer this question from another perspective: *redundancy* – which also aims to reduce N_{events}, but is *both sound and complete*. Our key observation stems from the fact that most dynamic memory accesses in real-world program executions are typically from the same program location, and they often lead to the same race because they are caused by repeated executions of the same racy instruction. Therefore, repeated memory accesses that do not reveal *any new* races can be skipped by the race detection tool, since removing them does not affect the capability nor precision of race detection. We term such memory accesses as *redundant events*.

Figure 1 shows a motivating example. The two threads T_1 and T_2 both write to a shared address x. T_1 acquires lock A before writing to x, while T_2 acquires lock B.

Because T_1 and T_2 do not share a common lock while writing to x, there exists a data race between ① and ②. Since the racy statements exist inside the loops that run 10 times, a race detector will check for races each time the event is generated. However, to detect this race it suffices to check only one event for each ① and ② and skip all the rest events. The rest events are redundant because they would not lead to any new *unique* race to be discovered. If we can remove these redundant events, the performance of race detection may be significantly improved, because for redundant events the race detection tools do not need to check races and to track their states. This optimization is tremendous in modern day multithreaded programs, as this type of redundancy is prevalent due to the single-process-multiple-data (SPMD) architectural design.

On the surface, this problem seems simple to solve by removing the events from the same program location. However, a treatment as such may remove important dependency information and produce incorrect results. For instance, consider another example in Figure 2. The example is slightly modified from that in Figure 1, but it contains additional wait and notify statements in the loops. There is still a race between ① and ② (starting from the second loop instance of ①). However, if we perform race checks for each lexical location just once (i.e., check only the first two writes at ① and ② and ignore the rest), the race will be missed, because the first write by T_1 happens before the first write by T_2 (incurred by the wait and notify statements).

To precisely and optimally capture such redundant events, we introduce *concurrential equivalence*, a new criterion that characterizes redundancy based on purely the dynamic

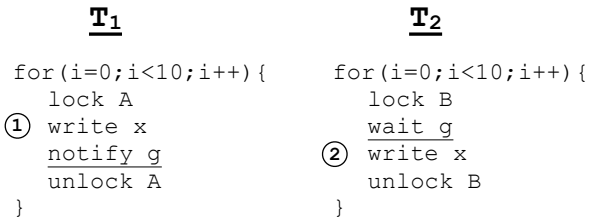

Figure 2 The race between ① and ② would be missed if we only check events from each lexical location once.

information associated with each event, without any static information of the program such as the data flow or control flow. Specifically, two events are concurrentially equivalent if they have the same *concurrency context* – a history of inter-thread happens-before context of the thread that performs the event. We prove that concurrential equivalence is sound and, for precise dynamic race detection, if there are two or more lexically-identical concurrentially equivalent events that access the same memory location, it is sufficient to keep any one copy of them for at most two threads and safely drop all the others.

Moreover, we show that such redundant events pervasively exist in both popular benchmarks and real-world programs, typically accounting for more than 99% of the events in the execution. If all these redundant events are removed, the (*hypothetical*) optimal performance of race detection can be improved by 90% for FastTrack, reducing its runtime overhead from 68X to 7X on a collection of CPU intensive benchmarks. For most of the Java Grande benchmarks, the FastTrack runtime overhead could be improved by as much as 95%, reducing the overhead from 80X to 3X only.

We further develop a dynamic technique, called ReX, to efficiently identify redundant events. The challenge is how to achieve efficiency while still maintaining both precision and soundness. We propose a Trie-based synchronization-free algorithm that on average identifies 97% redundant events. By running it as a filter before FastTrack, ReX improves the overall runtime performance by 31%. To further balance the runtime performance and effectiveness of redundancy identification, we have also explored a relaxation of ReX for the imprecise LockSet-based algorithms, which improves the performance of FastTrack by 32%.

In summary, this paper makes the following contributions:

- We present the first study of race detection performance from the perspective of redundancy and propose a new criterion that precisely and optimally characterizes redundant events for precise HB-based dynamic race detection.
- We show that redundancy pervasively exists in real-world program executions and eliminating redundancy has the potential to improve the performance of the state-of-the-art FastTrack race detector by 90%.
- We present a generalized algorithm, ReX, to remove redundant events dynamically without affecting the soundness or precision of the race detector, nor changing the race detection algorithm. We also present an optimization of ReX for the LockSet-based race detection algorithm.
- We integrate ReX with FastTrack, resulting in significant performance improvement on popular benchmarks.

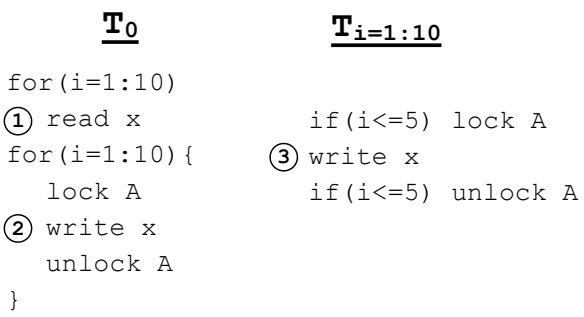

Figure 3 Intra- and inter-thread event redundancies.

2 Overview

In a precise dynamic race detector, a complete program execution trace, *i.e.*, a comprehensive sequence of critical events (memory accesses and thread synchronizations) is assumed to be observed. In general, no critical event should be missing. Otherwise, the detected races may be imprecise (i.e., false positives) or a real race may be missed. Nevertheless, a critical event may be redundant because it does not directly reveal any new races nor does it indirectly affect other new races to be detected. In particular, in a real-world program execution, we often observe multiple races between the same pair of lexical statements (i.e., same file, same class, same line and same column). However, just a single unique pair is enough to alert the user to a race between these two statements; the other warnings are superfluous and can be ignored. The race detector would need to filter out those superfluous races so that only unique race reports are sent to the user (to reduce the effort of the user). We note that, in practice, this functionality could be achieved for any race detector with an offline step that checks for equivalencies (like the ThreadSanitizer [2] dynamic detector does).

More pressingly, the additional computation required by detecting those superfluous races negatively impacts the runtime performance of race detection, since additional expensive operations must be performed, e.g., vector clock comparison and join, and memory operations for storing and loading the states associated with the memory accesses in a superfluous race.

We refer to an event in a trace as a *redundant event* if its exclusion from the trace does not lead to any missed races or false alarms by the same precise HB-based race detector. In other words, a redundant event does not reveal any additional useful information to the user of the race detector except negatively impacting the runtime performance. We next illustrate two kinds of redundant events (*intra-* and *inter*-thread redundancy) via an example in Figure 3. The main thread T_0 runs a loop inside which it reads and writes to a shared address x for 10 times. The shared lock A is used to protect the writes to x at ②, but not the reads to x at ①. Thread T_i ($i = 1, 2, \ldots, 10$) writes to this shared address protected conditionally through lock A for the first five threads. The remaining five threads write to x without previously acquiring lock A.

Consider the first iteration of the first loop. The read of x from ① by Thread T_0 races with the write from ③ (by any thread from T_1 to T_{10}). This pair of statements are the only race involving ① in the program. Subsequent iterations of ① all serve to expose the same lexical pair as a race and can be ignored (i.e., *intra-thread redundancy*). Consider the writes to x from ②. Among the threads T_i, although all of them are different in their execution, the traces by T_1-T_5 are identical, and the traces by T_6-T_{10} are identical. Because T_6-T_{10} do not acquire the lock before writing to x at ③, it results in two more races: a race with the write to x from ② by T_0, and a race between two instances of ③ by any two threads from

T_1-T_{10}. It is easy to see that four of the second five writes from ③ by T_6-T_{10} are redundant (i.e., *inter-thread redundancy*), because the existence of any one of the five is sufficient for precisely detecting all the three races: (①,③), (②,③) and (③,③).

How about the writes from ② and the first five writes from ③ by T_1-T_5? Is any of them redundant? For this example, it is tempting to conclude that most of them (except one write for each ② and ③) are redundant as well. However, they are not redundant from the perspective of a precise HB-based *dynamic* race detector, when the future execution is unknown. More specifically, these writes are protected by a lock; the lock operations introduce happens-before edges as required by the HB algorithm (albeit lock regions are commutative). These happens-before edges result in different happens-before relations for these writes, i.e., between these writes with possible future events, which may or may not produce new races depending on the specific happens-before relation. Therefore, because the future is unknown, we cannot remove any one of these writes. Interestingly, for an imprecise LockSet-based race detection algorithm [27], most of these writes are redundant, because their locksets are identical. We will elaborate this point more in Section 4.

From the above example, we can see that identical program location is only a necessary condition, but not the sufficient condition to determine if an event is redundant or not. A key contribution of this work is a criterion (called concurrential equivalence) that captures redundant events without any loss of race-detection ability or precision, for both intra-thread and inter-thread redundancies. Before introducing our criterion, we first need a precise definition of the HB-based race detection algorithm.

2.1 Happens-Before Race Detection

A happens-before algorithm [12], originated from Lamport's happens-before definition [18] for distributed systems, takes a dynamic trace (observed so far of a running program) and can precisely detect the *first* race. To detect the second or more races precisely, the algorithm needs a small extension that adds a happens-before edge between the two events in the first race. Our discussion in this paper concerns only the happens-before algorithm without extension.

To formally define the event redundancies, we need a model of a general program execution trace. Similar to other work [28, 17], we consider an event e in a program trace τ to be one of the following:

- **MEM**(t, a, m): A memory access event, where t refers to the thread performing the memory access, a can be one of **R***ead*/**W***rite* event and m the memory address being accessed.
- **ACQ**(t, l): A lock acquire event, where t denotes the thread acquiring the lock and l is the address of the acquired lock.
- **REL**(t, l): A lock release event, where t denotes the thread releasing the lock and l is the address of the released lock.
- **SND**(t, g): A message sending event, where t denotes the thread sending message with unique ID g.
- **RCV**(t, g): A message receiving event, where t denotes the thread receiving message with unique ID g.

For volatile accesses, they can be treated as **MEM** accesses enclosed by **ACQ** and **REL** events with unique lock addresses. For example, a write access to a volatile variable m corresponds to three consecutive events **ACQ**(t,l^*)-**MEM**(t,W,m)-**REL**(t,l^*), in which l^* is unique per dynamic memory location.

The *SND* and *RCV* events may be defined specifically to a language. For example, for Java, *SND*(*t*,*g*) and *RCV*(*t*,*g*) can be one of the following:

- If Thread T_1 starts T_2, it corresponds to a *SND*(T_1, g) and *RCV*(T_2, g).
- If Thread T_1 calls T_2.*join()*, *SND*(T_2, g) and *RCV*(T_1, g) are generated once T_2 terminates.
- If Thread T_1 calls *o.notify()* signaling a *o.wait()* on Thread T_2, this corresponds to a *SND*(T_1, g) and *RCV*(T_2, g).

For other complex synchronizations such as C11/C++11 atomic accesses, they can be treated conservatively as *REL*/*SND* operations each with a unique ID, such that they are always happens-before-ordered with the other events.

In addition, we associate each event with a static attribute *loc*, denoting the program location that generates the event.

Having defined a standard model of a program trace, we now formally define the happens-before (HB) relation.

Happens-Before Relation. The HB relation $\prec$ over events in a trace τ is the smallest relation such that:

- If a and b are events from the same thread and a occurs before b in the trace, then $a \prec b$.
- If a is a type of *SND* event and b is the corresponding *RCV* event, then $a \prec b$.
- If a is a type of *REL* event and b is the next *ACQ* event on the same lock, then $a \prec b$. This condition can be relaxed in a LockSet-based algorithm, which we will explain in Section 4.
- $\prec$ is transitively closed.

We note that the HB relation $\prec$ is a partial order over the trace. For any two events in the trace, e_i and e_j, either $e_i \prec e_j$ is true or $\neg(e_i \prec e_j)$ is true. Different from other algorithms [23, 15] that involve *may* happen-before, for an HB-based race detection algorithm, there is no may-happen-before relation.

To ease the presentation, we use $e_i||e_j$ to denote that e_i and e_j have no happens-before relation between each other: $\neg(e_i \prec e_j) \wedge \neg(e_i \prec e_j)$. In other words, $e_i||e_j$ means e_i and e_j can happen concurrently (if not in the observed execution, but can always happen in a certain execution of the same program).

Happens-Before Algorithm. The HB relation is usually checked by the use of vector clocks [21] or epoch-based clocks [12]. Two conflicting MEM accesses a and b (i.e., *Read/Write* events, at least one is a *Write*, accessing the same memory address), are determined to be in a race if they can happen concurrently: $a||b$.

2.2 Concurrential Redundancy

Having defined the happens-before algorithm, we are ready to define redundant events:

▶ **Definition 1 (Redundant Event).** Let HB-RaceDetect(τ) be the result of a precise dynamic HB-based algorithm running on τ, an input execution trace observed so far. HB-RaceDetect(τ) is either ϵ (if there is no race in τ) or (l_1,a_1,l_2,a_2), if there exist racing accesses $e_1||e_2$ in τ such that l_i=$loc(e_i)$ and $a_i = access(e_i)$ and $l_1 < l_2$ (the location ordering is for symmetry breaking). An event e is redundant iff HB-RaceDetect(τ) = HB-RaceDetect($\tau \backslash e$).

▶ **Definition 2 (Concurrential Equivalence).** The key observation behind concurrential equivalence is that, for two MEM events e_i and e_j, their *inter-thread happens-before* relation can determine their equivalence. Regardless of which thread(s) they are from, e_i and e_j are *concurrentially equivalent* if they satisfy the following conditions:

1. they share the same program lexical location (i.e., $loc(e_i)$=$loc(e_j)$) and have the same access type (i.e., both are reads, or both are writes);
2. they access the same dynamic memory location;
3. they have the same inter-thread HB relations with events from any other thread that is different from t_i *or* t_j. More formally, $\forall\ e_k$, $t_{e_k} \neq t_i \vee t_{e_k} \neq t_j$, such that $e_k \prec e_i \iff e_k \prec e_j$ and $e_i \prec e_k \iff e_j \prec e_k$.

For Condition 3, we note that there are two possible cases: 1) $t_i = t_j$ and 2) $t_i \neq t_j$. As long as t_{e_k} is different from any of them, the condition must be held. We also note that from the two $\iff$ conditions, we can derive $e_k||e_i \iff e_k||e_j$.

With concurrential equivalence, we can formally prove the following theorem:

▶ **Theorem 1 (Concurrential Redundancy).** *An event e is redundant if there already exists one concurrential equivalent event from the same thread, or two from different threads.*

Proof. The key insight for the proof is that a race involves only two events from two different threads. Let us assume two concurrentially equivalent events e_i and e_j, and consider an arbitrary event e_k. If e_i and e_j are from the same thread, and if e_k and e_i form a data race, then e_k and e_j must be a race too. The reason is that e_i and e_j have the same inter-thread HB relation, and e_k must be from a different thread. Hence, either e_i or e_j is redundant. On the other hand, if e_i and e_j are from different threads, and if e_k and e_i form a race, there are two possibilities. One is that e_k is from a third thread different from that of e_i and e_j. In that case, either e_i or e_j is redundant, because e_k would race with e_j too. The other case is that e_k is from the same thread as e_j. In that case, neither e_i nor e_j is redundant. However, for any other event e_w that is concurrentially equivalent to e_i and e_j, e_w must be redundant. The reason is that e_w would either form a race with e_k (if it is from a thread different from that of e_k), or is redundant with e_j (if it is from the same thread as e_k).

Meanwhile, we can prove that no new races would be reported if such an event e is removed from the trace. Let us assume a certain new race (e_i, e_j) is reported in $\tau\backslash e$ but not in τ. Then it must be the case that in $\tau\backslash e$, $e_i \prec e_j \vee e_j \prec e_i$, but in τ, $e_i||e_j$. The only possibility is that $e_i \prec e \prec e_j \vee e_j \prec e \prec e_i$. However, because in $\tau\backslash e$, there should exist an event e' that is concurrentially equivalent to e, then we should also have $e_i \prec e' \prec e_j \vee e_j \prec e' \prec e_i$. This contradicts to the assumption that (e_i, e_j) is a race. ◀

We can hence use Theorem 1 to identify redundant events. But is it optimal? Can we safely remove any more events from the trace without affecting the race detection results? Interestingly, Theorem 1 only defines optimal equivalence between events, but it is not optimal for defining redundancy. More specifically, we can improve Theorem 1 to capture more redundant events by relaxing Condition 3 with the "*HB-subsume*" relation.

▶ **Definition 3 (HB-subsume).** An event e_i *HB-subsumes* ($\preccurlyeq$) e_j if the HB relation of e_j is a subset of e_i for events from any other thread that is different from t_i *or* t_j. More formally, if $e_i \preccurlyeq e_j$ then $\forall\ e_k$, $t_{e_k} \neq t_i \vee t_{e_k} \neq t_j$, such that $e_k \prec e_i \implies e_k \prec e_j$ and $e_i \prec e_k \implies e_j \prec e_k$.

Comparing the conditions in HB-subsume with that in Condition 3, the difference is that $\iff$ is changed to $\implies$. That is, the inter-thread HB relations of e_i and e_j need not to be

equivalent, but is relaxed to be a subset relation. The key insight is that if $e_i \preccurlyeq e_j$, then e_j only represents a subset of the happens-before information represented by e_i; hence e_i can replace e_j for race detection. Similarly, we can define *concurrential-subsume equivalence* and prove Theorem 2:

▶ **Definition 4 (Concurrential-subsume Equivalence).** For two MEM events e_i and e_j, e_j is concurrentially-subsumed by e_i if:

1. they share the same program lexical location and have the same access type (i.e., both are reads, or both are writes);
2. they access the same dynamic memory location;
3. $e_i \preccurlyeq e_j$.

▶ **Theorem 2 (Optimal Concurrential Redundancy).** *An event e is redundant if there already exists one concurrential-subsuming equivalent event from the same thread, or two from different threads.*

Proof. The proof is similar to that of Theorem 1. The only difference is changing concurrential equivalence to concurrential-subsume equivalence. Meanwhile, since a race involves at least and at most two events, it is impossible to further remove any more such events, otherwise a certain race may be missed. Hence, we can also prove that Theorem 2 is optimal for characterizing concurrentially-redundant events. ◀

We can hence use Theorem 2 to precisely and optimally identify concurrentially-redundant events. To clarify, we note that Theorem 2 only considers those events that *may* be involved in data races but cannot introduce new races. We do not consider redundant events that can *never* participate in any data race, e.g., events that are always happens-before-ordered before some event for each thread. It is possible to further remove events beyond our definition of concurrential equivalency. Nevertheless, that would require checking the happens-before relation between events, which is as expensive as running a full HB algorithm.

3 The ReX Algorithm

For dynamically generated event streams from a running program, checking the first two conditions of concurrential-subsume equivalence is easy: lexical equivalence can simply check the originating program location of the event, access types can be recognized easily during instrumentation, and dynamic memory location is available at runtime. Checking the third condition (i.e., $\preccurlyeq$) however, if done naively, would prove prohibitively expensive, especially when the algorithm needs to be run online during program execution. To efficiently check the $\preccurlyeq$ condition, we introduce a new concept called *concurrency context*:

▶ **Definition 5 (Concurrency Context).** The concurrency context of a thread t, Γ_t, encodes the history of *SND* and *REL* events observed by t, with the thread attribute t ignored. The concurrency context of an event e generated by thread t is the value of Γ_t at the time e is observed.

It is easy to see that if two events e_i and e_j have the same concurrency context and e_i appears before e_j, then they must satisfy the $e_i \preccurlyeq e_j$ condition, because only *SND* and *REL* introduce outgoing inter-thread HB edges; for all the other event types (i.e., *RCV*, *ACQ* and*MEM*), they only introduce intra-thread or incoming HB edges.

Algorithm 1 ReX(e)

```
1: e ← input event
2: t = e.getThread
3: loc = e.getLocation
4: Γ_t // concurrency context of thread t
5: Θ_loc // concurrency history at location loc
6: switch e do
7:     case MEM:
8:         if CheckRedundancy(t, Θ_loc, Γ_t) then
9:             discard e
10:        else
11:            advance e
12:        end if
13:    case REL:
14:        Γ_t.add(e.l)//add the lock l
15:        advance e
16:    case SND:
17:        Γ_t.add(e.g)//add the message g
18:        advance e
19:    case Other:
20:        advance e
```

Finally, we introduce the concept of *concurrency history* for a particular lexical location:

▶ **Definition 6 (Concurrency History).** The concurrency history at a static program location loc, Θ_{loc}, stores the union of Γ_t of all threads t that have accessed this location.

The concurrency history Θ_{loc} can be used to filter out redundant events from location loc. Moreover, since the concurrency contexts of different events from the same location exhibit strong temporal locality due to stack based computational model of programs, a prefix sharing data-structure such as *trie* is ideal for storing Θ_{loc}. This results in compact storage and fast retrieval in our design of ReX.

We design ReX as a filter pass over the event stream generated by the program execution. It is generic by design and can be applied to any dynamic race detectors and it is sound for the precise HB-based race detection algorithms such as FastTrack. Algorithm 1 provides a high-level overview of how ReX applies the redundancy filters. It updates Γ_t as events stream by. The calls *discard* and *advance* indicate when ReX decides that the event is redundant and discard it or advance it to the race detector, respectively. The MEM events are handled separately from the other types of events:

1. **MEM**: Memory access events, both read and write are checked for redundancy (Algorithm 2). If this call returns true, the event is redundant and it is filtered.
2. **SND** and **REL**: These events always append to Γ_t their unique ID g or l.
3. **RCV** and **ACQ**: These events are not processed but just advanced to the race detector.

For each MEM event, the *CheckRedundancy* function determines its redundancy by checking the corresponding concurrency history Θ_{loc} and the current concurrency context Γ_t of the thread. Recall Theorem 2 that an event is redundant if there already exists one concurrential-subsuming equivalent event from the same thread, or two from different threads.

Algorithm 2 CheckRedundancy(t, Θ_{loc}, Γ_t)

```
1: stack ← getStack(Θ_loc,Γ_t)//get the stack associated with the concurrency context and location
2: if stack is empty then
3:     Θ_loc.add(Γ_t)
4:     return false
5: else if stack.contains(t) then
6:     return true
7: else if stack.size = 1 then
8:     stack.add(t)
9:     return false
10: else if stack is full then
11:    return true
12: end if
```

To check this condition, each node in Θ_{loc} contains a bounded stack of size **2** that is used to keep track of the number of concurrential equivalent events seen so far. If the stack is full, new events having the same Γ_t are filtered out since they are redundant. The elements of the stack denote the threads that have contributed to the particular concurrency context. The first step is to check the stack corresponding to the current thread's concurrency context. Based on the contents of this stack, there are four cases to consider:

1. **Stack is empty:** This implies that this particular concurrency context was not seen in any of the accesses so far, hence the event is not redundant. We proceed to add Γ_t into Θ_{loc} for future accesses, where t is the thread ID of the current event e and loc is the program location of e.
2. **Stack contains t:** This case falls in the category of intra-thread redundancy, so e and can be eliminated.
3. **Stack does not contain t and is of size 1:** Add t to the stack.
4. **Stack is full:** This case falls in the category of inter-thread redundancy, so e can be eliminated.

Synchronization-free Implementation. Algorithm 1 is simple and mostly straightforward to implement. The only problem is that the algorithm itself is multithreaded and the trie for storing each concurrency history Θ_{loc} is a shared data structure. Multiple threads may concurrently access Θ_{loc} with the same concurrency context Γ_t and attempt to store a new stack into the trie for the same entry Γ_t if a stack is not available for the entry (at line 2 in Algorithm 2). To correctly implement the algorithm, this operation must be synchronized. However, a synchronized implementation would slow down ReX significantly, especially for programs with a large number of threads running on multicore processors and for this scenario, the synchronization operation is performed for every check.

We develop a synchronization-free implementation that does not use any locks to protect the new stack store operation. Specifically, for each node in the trie, we maintain a hashmap from the current concurrency context ID (message g or lock l) to its children nodes. When a synchronization event with ID x is generated, the corresponding thread checks the hashmap to return a child node for x and creates a new node if not available. Multiple threads are allowed to check the hashmap and create new entries in it without synchronization. Because there is no synchronization, two threads may create two new nodes for the same concurrency context ID, and one of them would be overwritten by another.

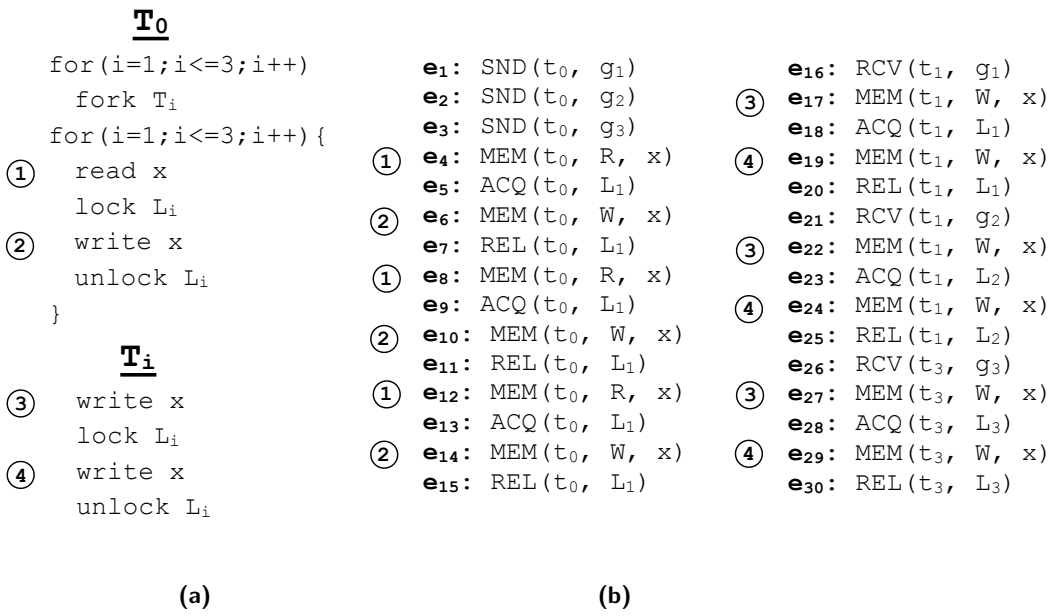

Figure 4 A program exhibiting event redundancies and a serialized execution trace.

The loss of one node causes the corresponding stack associated with the concurrency context ID x to miss one entry, which means that a redundant event may be missed. However, since the chance for two threads to check the hashmap with the same concurrency context ID at the same time is very small, this treatment rarely misses redundant events in practice (in our extensive experiments we only observed one or two such cases out of every one million events on average). Moreover, this treatment is sound that it does not miss any non-redundant events.

3.1 Example

We use the example in Figure 4 to illustrate ReX. This program in (a) contains two loops: the first spawns three threads, $T_{1,2,3}$, and the second performs a read at program location ①, followed a lock region on L_i protecting a write at ② in each loop for $i = 1, 2, 3$. Threads $T_{1,2,3}$ are all identical except in the lock addresses used to guard the write at ④. The write at ③ is unguarded. A trace corresponding to a serialized execution of the program that executes $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$ is shown in (b). If this trace is given to a precise HB-based race detector, a race between e_{14} and e_{17} will be detected as the first race: In fact, if all possible thread schedules are explored, a powerful race detector such as RVPredict [15] can detect 45 races in total in this program: ($e_{(4,8,12)}$, $e_{(17,19,22,24,27,29)}$), ($e_{(6,10,14)}$, $e_{(17,22,17)}$), (e_6, $e_{(24,29)}$), (e_{10}, $e_{(17,29)}$), (e_{14}, $e_{(17,24)}$), ($e_{17,19}$, $e_{(22,24,27,29)}$), and ($e_{22,24}$, $e_{(27,29)}$). However, only 7 of them have unique lexical locations: (①,③), (①, ④), (②, ③), (②, ④), (③, ③), (③, ④) and (④, ④). The rest 38 races are superfluous and should be removed from the race reports. We would like to use ReX to identify those redundant events that lead to these superfluous races.

Figure 5 illustrates how ReX works for this example. There are four program locations of interest, marked by ①-④.

Location ①: Following Algorithm 1, the three events $e_{1,2,3}$ first add their unique message ids into Γ_t. The read e_4 by T_0 is then added to the stack associated with the concurrency

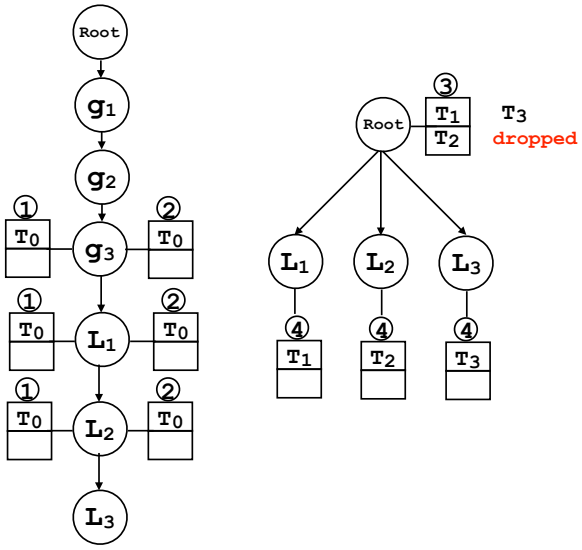

Figure 5 Trie states after applying ReX on the trace in Figure 4b.

context $g1 - g2 - g3$. Note that the lock acquire e_5 does not add anything to Γ_t, but the lock release e_7 appends $L1$ to it. Similarly, the read e_8 by T_0 is added to the stack associated with the concurrency context $g1 - g2 - g3 - L1$. At the end of three iterations, $e_{4,8,12}$ are added to the stacks associated with three different concurrency contexts. The stack at each of these locations contains the single thread T_0 and thus, none of the accesses is dropped.

Location ②: Similar to that of ①, $e_{6,10,14}$ are added to three different stacks, because the lock acquire events extending the concurrency context with $L1$, $L2$ and $L3$.

Location ③: The first two threads T_1 and T_2 access this location and get added into the stack. The third thread T_3 is however filtered since the stack is already full, exhibiting inter-thread redundancy.

Location ④: Similar to how each T_0 acquires a lock, the writes this location are each guarded by a different lock. Thus, the thread ID of each write is added to a different stack.

For the example above, the only redundant event dropped by ReX is the third event from location ③. Keen readers may wonder why we cannot drop some of the other events (e.g., the second and the third read events from location ①). The fundamental reason is that these events are not redundant for a precise HB-based race detection algorithm. For instance, it may appear that the second read event e_8 from ① is redundant to its first read event e_4; but according to the HB algorithm, the lock release event e_7 introduces an outgoing HB edge, resulting in a different inter-thread HB relation for e_8. Hence, a possible future event, say e_{100}, may race with e_8 but not e_4.

We next introduce a relaxation of ReX that is unsound for the precise HB-based algorithm, but is sound for the LockSet algorithm. It has the power to identify all those seemingly redundant events in this example. However, in principle, because the LockSet algorithm is unsound, this relaxation of ReX may result in missing real races. Nevertheless, in our experiments we rarely observe such cases.

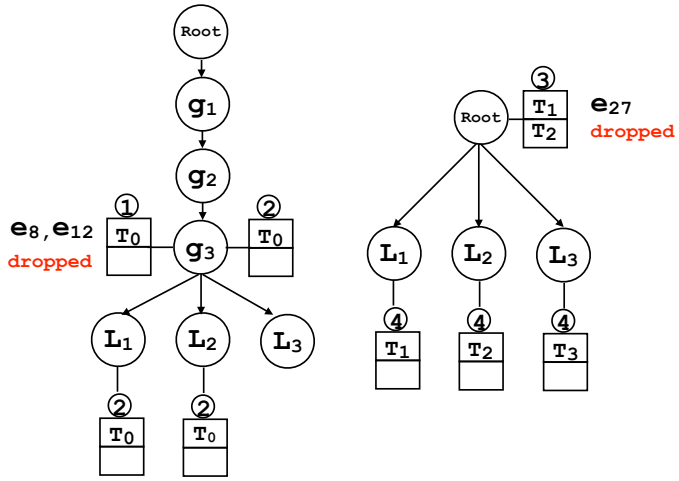

Figure 6 Trie states with the LockSet optimization for the trace in Figure 4b.

4 The LockSet Optimization

In a pure LockSet-based race detector [27] or hybrid race detectors [24, 16] which combine HB and LockSet, the contribution by locks is often ignored in the HB relation. Instead, ACQ and REL events are tracked separately using LockSet.

LockSet Condition: The set of locks currently held by a given thread is referred to as its LockSet. The LockSet condition states that two conflicting accesses are in a race if there is no HB relation between them, and the LockSets of the two threads do not overlap, *i.e.*, $L_i \cap L_j = \emptyset$, where L_i and L_j refer to the LockSet of T_i and T_j, respectively, at the time of event generation.

The LockSet condition allows us to optimize ReX by filtering redundant events across synchronization boundaries incurred by both ACQ and REL events, because events with the same LockSet may be redundant. For example, the second and the third read events from location ① in Figure 4 can now be filtered because according to the LockSet algorithm, these two events are redundant to the first read event from ①.

This optimization can be implemented by slightly modifying the ReX algorithm (Line 13 in Algorithm 1). Specifically, instead of appending the lock l to the thread concurrency context Γ_t for REL, upon an ACQ or REL event, we can perform the following:

- **ACQ**: add the lock address into Γ_t. If a lock previously acquired is acquired again, we ignore the event.
- **REL**: remove the lock address from Γ_t. In a well-formed trace, the corresponding lock acquire event of this address must have already been observed before this event is seen.

To support reentrant locks, we can further add a local counter to each lock address in the concurrency context Γ_t. The counter is zero initially, and is incremented (or decremented) by one upon each **ACQ** (or **REL**) of the corresponding lock address. The lock address is removed from Γ_t when the counter becomes zero upon a **REL**.

Figure 6 illustrates how ReX with this optimization works for the same example in Figure 4b. The main difference is **Location ①:** the three read events $e_{4,8,12}$ now have the same concurrent context $g1 - g2 - g3$ with an empty lockset. Therefore, e_8 and e_{12} can be filtered out.

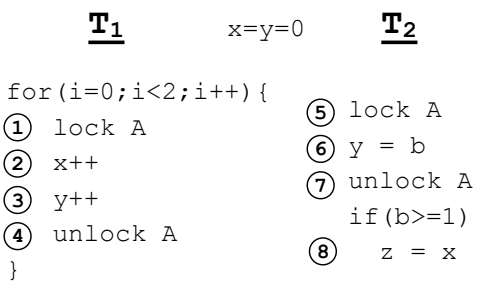

Figure 7 An example illustrating the unsoundness of the LockSet optimization.

We next use an example in Figure 7 to illustrate why this optimization is sound for the hybrid (or LockSet-based) algorithm only, but unsound for the HB-based algorithm such as FastTrack. For HB-based race detectors, the location ② and ⑧ are in a race when the schedule is ①-②-③-④-⑤-⑥-⑦-⑧-①-②-③-④. However, the LockSet optimization will determine that the second event from ② is redundant to the first event from ② and hence filter it out from the trace. Because the first event from ② happens before the event from ⑧ (introduced by the lock and unlock events from ④-⑤), the race will be missed.

5 Evaluation

Our evaluation focuses on answering the following four sets of research questions:

1. *Redundancy*: How much event redundancy is there in real-world execution traces?
2. *Optimal Performance*: Hypothetically, what is the optimal runtime performance of a precise HB-based dynamic race detector if all concurrentially-redundant events were removed with no cost?
3. *ReX Effectiveness and Efficiency*: How effective is ReX in removing redundant events? Can ReX improve runtime performance of dynamic race detectors? How much speedup or slowdown can ReX bring?
4. *ReX Precision and Soundness*: Does ReX affect the precision or soundness (*i.e.*, detection ability) of race detection in practice?

Evaluation Methodology. We use ReX as a preprocessing step in the RoadRunner tool chain [13], and compare the runtime performance and race detection results between FastTrack with and without ReX. FastTrack implements the fastest precise dynamic race detection algorithm, so we focus on integrating FastTrack with ReX. ReX intercepts the full event steam generated by RoadRunner (without any optimization), and passes an event to FastTrack when it determines that the specific event is not redundant.

We have evaluated ReX as well as the LockSet optimization on a collection of 13 commonly studied multithreaded benchmarks including all the eight benchmarks from the Java Grande suite, four from the DaCapo suite[1] [5] (which are all real-world applications), as well as the popular *Tsp* (traveling salesman problem) benchmark. Table 1 summarizes the benchmarks and their trace characteristics. The first eight benchmarks are from Java Grande and all of them were tested running on 20 threads. The next four benchmarks are from DaCapo

[1] DaCapo contains several other multithreaded applications, but we do not include them because the RoadRunner tool failed to instrument them.

Table 1 Benchmarks and the trace characteristics. For all benchmarks, 99+% of the events are memory reads or writes.

Benchmark	#Threads	#Events			
		MEM	Volatile	ACQ/REL	SND/RCV
LUFact	20	7.6G	23	12	38
Series	20	6M	0	12	38
Sor	20	2.65G	77.7M	12	38
Sparse	20	7.7G	0	12	38
Crypt	20	2.1G	0	12	76
MonteCarlo	20	492M	0	22	38
Moldyn	20	2.02G	23	22	38
RayTracer	20	3.55G	23	148	38
Avrora	7	1.4G	0	2.9M	580K
Xalan	9	1.1G	0	8.9M	1.7K
Sunflow	17	9.7G	0	1.8K	32
Lusearch	10	1.4G	1.2M	2.7M	136
Tsp	9	1.5G	0	62K	16

and were tested under the default configuration. Columns 3-6 report the number of different types of events in the execution, For all the benchmarks, the MEM events are the majority accounting for more than 99% of all the events. Note that volatile memory accesses are reported separately because they introduce happens-before according to the standard HB semantics [19], and they are not checked for redundancy since concurrent volatile accesses are not data races.

To evaluate the trace redundancy, we run ReX without optimization to obtain the number and the percentage of redundant events. To assess the optimal performance of dynamic race detection, we assume that all concurrentially-redundant events could be eliminated with no cost. The optimal overhead can hence be approximated by subtracting the running time of ReX+FastTrack with that of running ReX alone. An additional performance factor is the runtime cost of instrumentation, which is used to generate the event stream. In practice, because the instrumentation in RoadRunner is based on code rewriting, the cost can be high. Therefore, we also include the instrumentation cost and calculate the optimal performance as $X - Y + Z * (1 - redun\%)$, where X is the cost of ReX+FastTrack, Y the cost of ReX alone, Z the instrumentation cost of all events and $redun\%$ the percentage of redundant events. To measure Z, we run standalone RoadRunner on each benchmark with no race detection.

Hardware Configuration. The hardware used to run these experiments was an eight-core iMac machine with 4.0GHz Intel Core i7 processor, 32 GB DDR3 memory with Java JDK 1.8 installed.

Summary. The results are reported in Tables 2-5. All experimental data were averaged over three runs. Overall, ReX and its LockSet optimization identify 97% and 98.2% of the total events as redundant, respectively, improving the runtime performance of FastTrack by more than 30% while incurring 1.3X and 1.2X memory overhead, and producing the same unique data races as reported by FastTrack. If all redundant events were removed with no runtime cost, the optimal performance of FastTrack can be improved by 90% on average.

We next discuss the results with respect to these research questions.

Table 2 Results of event redundancy and race detection performance.

Benchmark	Native time	Instrument only	FastTrack time(o.h.)	ReX only	ReX+FT time(o.h.)	#redun(%)
LUFact	1.94s	36.2s(**18X**)	108s(**55X**)	51.9s(**26X**)	55.6s(**33X**)	7.59G(**99%**)
Series	78.9s	90.1s(**14%**)	86.6s(**10%**)	89s(**13%**)	86.1s(**9%**)	5.99M(**99%**)
Sor	2.82s	15s(**4X**)	35.3s(**12X**)	17.7s(**5X**)	18.5s(**6X**)	2.53G(**95%**)
Sparse	0.6s	39.8s(**65X**)	124s(**206X**)	60.7s(**100X**)	61.2s(**101X**)	7.69G(**99%**)
Crypt	0.35s	17s(**47X**)	55.8s(**158X**)	31.7s(**90X**)	50.4s(**143X**)	2.09G(**99%**)
MonteCarlo	0.59s	2.75s(**4X**)	9.9s(**16X**)	5.1s(**8X**)	7.4s(**11X**)	488M(**99%**)
Moldyn	0.42s	10.7s(**24X**)	30.1s(**71X**)	19.1s(**44X**)	18.8s(**44X**)	2.01G(**99%**)
RayTracer	0.36s	12.7s(**34X**)	43.1s(**119X**)	22.9s(**63X**)	24s(**66X**)	3.54G(**99%**)
Avrora	2.4s	19.4s(**7X**)	36s(**14X**)	32.1s(**12X**)	35s(**13X**)	1.21G(**87%**)
Xalan	1.7s	14.2s(**7X**)	26s(**14X**)	22.7s(**12X**)	26s(**14X**)	1G(**93%**)
Sunflow	1.8s	47.1s(**25X**)	157s(**86X**)	122.7s(**67X**)	132s(**72X**)	9.69G(**99%**)
Lusearch	1.2s	57.5s(**47X**)	68s(**56X**)	42.6s(**35X**)	58s(**47X**)	1.37G(**97%**)
Tsp	0.9s	31.9s(**34X**)	67s(**73X**)	55.2s(**60X**)	58s(**63X**)	1.49G(**99%**)
Average	-	**24X**	**68X**	**40X**	**47X (↓31%)**	**97%**

5.1 Redundancy and Optimal Performance

Table 2 Column 2 reports the native execution time of each benchmark, ranging from 0.35s for *Crypt* to 78.9s for *Series*. Column 3 reports the running time of the instrumented version and the instrumentation slowdown. The instrumentation incurs 24X slowdown on average, ranging between 14%-65X. Column 4-6 respectively report the time and overhead of FastTrack, ReX alone and ReX+FastTrack. Column 7 reports the number of the redundant events identified by ReX and their percentage over the total events.

Overall, redundant events are pervasive in these benchmarks. This is expected because repeated memory accesses from the same lexical locations via loops are typical in real-world programs. For most benchmarks (10 out of the 13 benchmarks), more than 99% of the events are redundant. On average, ReX identifies 97% of the total events as redundant. For the other three (*Sor, Avrora, Xalan*), ReX identifies 95%, 87%, and 93% redundant events, respectively.

Our result indicates that the performance of dynamic race detection has a large improvement space by removing the concurrentially-redundant events. If all concurrentially-redundant events were removed from the trace (e.g., by compiler analysis or a zero overhead runtime analysis to identify redundancy), the performance of FastTrack could be improved by 90% (following the formula $X - Y + Z * (1 - redun\%)$ described earlier), reducing the runtime overhead of FastTrack from 68X to 7X for all these benchmarks on average. For most of the Java Grande benchmarks, the FastTrack runtime overhead could be reduced by 95%, from 80X to 3X only. The only exception is *Crypt*, for which the overhead of FastTrack can be reduced by 66%, from 158X to 53X.

5.2 ReX Performance, Precision & Soundness

The number and percentage of redundant events identified by ReX without and with the LockSet optimization are reported in the last columns in Table 2 and Table 3, respectively.

Overall, ReX improves the runtime overhead of FastTrack by 31% (from 68X to 47X) on average. The LockSet optimization further improves the performance by 1% (to 46X). ReX identifies that on average 97% of the events in the trace are redundant. ReX with the LockSet optimization identifies 98.2% of the total events as redundant. For instance, for *Avrora*, while ReX only detects 87% redundant events, the LockSet optimization detects 95%.

Table 3 Runtime performance of ReX with the LockSet optimization.

Benchmark	ReX-**LockSet+FT**	
	#redun(%)	time(**o.h.**)
LUFact	52.1s(**26X**)	7.59G(**99%**)
Series	85s(**8%**)	5.99M(**99%**)
Sor	18s(**5X**)	2.6G(**98%**)
Sparse	59.7s(**99X**)	7.69G(**99%**)
Crypt	50.2s(**142X**)	2.09G(**99%**)
MonteCarlo	7s(**56X**)	488M(**99%**)
Moldyn	19.1s(**44X**)	2.01G(**99%**)
RayTracer	23.1s(**63X**)	3.54G(**99%**)
Avrora	26s(**10X**)	1.33G(**95%**)
Xalan	25s(**14X**)	1.08G(**98%**)
Sunflow	80s(**43X**)	9.69G(**99%**)
Lusearch	52s(**42X**)	1.39G(**99%**)
Tsp	54s(**59X**)	1.49G(**99%**)
Average	**46X**(↓32%)	**98.2%**

The reason is that the LockSet optimization can detect redundant events across the lock ACQ boundaries.

Table 4 reports the memory overhead of ReX. ReX incurs 1.3X memory overhead compared to FastTrack. The LockSet optimization further reduces the memory overhead to 1.2X, because more redundant events are filtered out.

Table 5 reports the total number of data races and the number of unique races among them detected by FastTrack, FastTrack with ReX, and FastTrack with ReX and the LockSet optimization. For all the benchmarks, ReX and the LockSet optimization both result in the same number of unique data races detected by FastTrack. Even though the LockSet optimization is unsound in theory for HB-based race detectors, it does not affect the race detection results in these benchmarks. We also empirically validated that the unique races detected by FastTrack match with the unique races detected upon using ReX and the optimization. This confirms that ReX is both theoretically sound and practically useful.

One additional benefit we observed, that was not originally planned, was that error output verbosity tended to be greatly reduced. Sometimes, we observed that FastTrack reports races on a particular race pair several hundreds of times, even though a single instance is sufficient to alert the programmer to the concurrency bug. ReX filters most of the redundant events before sending them to FastTrack, reducing the total number of the reported races and saving the user valuable time in parsing the tool output. For instance, FastTrack reports 644 total races in *Sor*, but only 10 of them are unique. With ReX-LockSet, this number is reduced to 38, containing the same number of unique races. This also proved very useful in our evaluation stage when we compared the race output with and without ReX. Of course, this benefit can also be achieved via an automatic offline analysis that filters out superfluous race reports.

6 Related Work

Data race detection has attracted a significant research attention in the past few years motivated by the multicore and manycore hardware architectures. Researchers have proposed

Table 4 Memory overhead (MB) of ReX.

Benchmark	**FastTrack**	ReX+**FT**	ReX-**LockSet+FT**
LUFact	2194	3137(**43%**)	3137(**43%**)
Series	339	290(-**14%**)	301(-**11%**)
Sor	1705	5326(**2.1X**)	2277(**34%**)
Sparse	902	885(-**2%**)	1984(**1.2X**)
Crypt	5905	11896(**1X**)	9994(**68%**)
MonteCarlo	1637	4494(**1.75X**)	4456(**1.72X**)
Moldyn	233	1267(**4.4X**)	1267(**4.4X**)
RayTracer	68	331(**3.9X**)	411(**5X**)
Avrora	347	1501(**3.3X**)	636(**83%**)
Xalan	3183	3212(**1%**)	2885(**-9%**)
Sunflow	1974	3254(**65%**)	2665(**35%**)
Lusearch	24985	22837(**-9%**)	26570(**6%**)
Tsp	848	879(**4%**)	2727(**2.2X**)
Average	-	**1.3X**	**1.2X**

a wide spectrum of race detection techniques, both static [23, 31] and dynamic [4, 10, 12]. targeting different application domains [3, 22] and different types of software [9, 11, 25].

To improve runtime performance, there are two areas where recent research has focused on: 1) improved underlying race-detection algorithms such as FastTrack [12], and 2) reduced static instrumentation or runtime checking of races. Reducing the number of instrumented or checked events by finding redundancies is orthogonal to the race-detection algorithm and works across different algorithms. There are three families of techniques that help in finding these redundancies, discussed below.

Static analysis based tools: Tools such as [8, 14, 34, 10, 30] target statically identifying redundant events that will never or less likely lead to races. They eliminate those accesses that are guaranteed to be race-free or would not result in generation of any new races. For example, IFRit [10] identifies interference free regions of the program and reduces instrumentation in them. RaceTrack [34] adds more instrumentation to those regions that are more susceptible of races and lesser instrumentation to regions that are not. However, precisely analyzing the source code and determining such regions is hard. These tools struggle to properly analyze external library features and program constructs such as reflections, which may result in loss of precision or soundness.

The static analysis closest to our work is RedCard [14], which proposes Span redundancy, a static release-free region from the same thread bounded by two outgoing HB edges. Span redundancy captures a subset of redundant events characterized by concurrential-subsume redundancy. For example, it does not capture redundant events across different threads. In addition, detecting redundant events at runtime has a number of benefits: 1) it greatly simplifies the algorithm design because the address and thread of memory accesses is available at runtime; 2) it handles dynamic program features automatically without expensive or undecidable static analysis; 3) it may detect more redundant events (those are input-specific).

BigFoot [26] is a recent technique that combines sophisticated static analysis with dynamic analysis to coalesce checks and compress metadata for checks. It significantly improves the performance of FastTrack by 60% because multiple accesses to an object or array may be converted to a single check that manipulates a single piece of compressed metadata, e.g., it may move a check out of a loop. Compared to BigFoot, ReX does not require static analysis.

Table 5 Number of detected total and unique races by different approaches.

Benchmark	FastTrack	ReX+FT	ReX-LockSet+FT
LUFact	0(**0**)	0(**0**)	0(**0**)
Series	0(**0**)	0(**0**)	0(**0**)
Sor	644(**10**)	44(**10**)	38(**10**)
Sparse	0(**0**)	0(**0**)	0(**0**)
Crypt	0(**0**)	0(**0**)	0(**0**)
MonteCarlo	100(**1**)	8(**1**)	38(**1**)
Moldyn	0(**0**)	0(**0**)	0(**0**)
RayTracer	100(**1**)	100(**1**)	100(**1**)
Avrora	200(**2**)	200(**2**)	200(**2**)
Xalan	168(**8**)	16(**8**)	16(**8**)
Sunflow	63(**8**)	13(**8**)	12(**8**)
Lusearch	205(**11**)	30(**11**)	28(**11**)
Tsp	100(**1**)	81(**1**)	63(**1**)

Online tools: To improve runtime performance, several online sampling techniques [6, 20, 34] have been proposed to scale dynamic race detection to long running programs. LiteRace [20], Pacer [6], and RaceTrack [34] all use sampling to reduce the tracing overhead and may achieve negligible runtime slowdown, at the cost of reduced race detection ratio. SlimState [33] introduces an online algorithm to optimize array shadow state representations. RoadRunner [13] has an inbuilt thread-local pass that is supposed to speed up dynamic analysis tools by filtering memory addresses that are solely accessed by a single thread. However, we found that the design of this filter is unsound and can result in missing races.

Post-processing on trace: Huang et al. [17] propose an offline trace analysis, TraceFilter, to remove redundant events in the context of predictive concurrency analysis for detecting concurrency analysis anomalies such as data races and atomicity violations. Our work is inspired by this analysis. However, TraceFilter only captures a subset of redundant events characterized by our concurrential redundancy criterion. Specifically, TraceFilter captures intra-thread redundant events with respect to the hybrid HB and LockSet algorithm, as well as entirely redundant threads. It does not capture inter-thread redundant events, and because of LockSet it is unsound for the precise HB algorithm.

Improved detection. Another area of much development is the design of tools that try to improve the race detection ability. Predictive trace analysis [15, 16, 7, 29, 32] records a single execution of the program and then generates other permutation of the trace events under different scheduling constraints allowing it to detect concurrency bugs not exposed in the original trace. We plan to investigate the applicability of ReX for this type of analyses dynamically in future work.

7 Conclusion

We have shown that for dynamic race detection there exists a significant percentage of redundant events that do not reveal any new races and we propose a criterion, concurrential redundancy, that precisely and optimally characterizes them. We have also shown that if such redundant events were all removed, the performance of the state-of-the-art precise dynamic

race detector FastTrack could be significantly improved by 90% on popular benchmarks and real-world programs. We have also presented a technique, ReX, that efficiently identifies redundant events and filters them out from the trace. The key enhancement over previous techniques is that ReX is sound, optimal, and purely dynamic. This gives us the ability to be completely unaware of complicated program semantics and perform filtering at runtime without changing the race detection algorithm, and without affecting the soundness and precision of the race detection result. Our evaluation results show that ReX improves the runtime performance of FastTrack by 31% on average.

Acknowledgements. We thank our shepherd, Sebastian Burckhardt, and the anonymous reviewers for their valuable feedback.

References

1 Java Grande benchmark suite. `https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/grande.html`.

2 ThreadSanitizer. `http://clang.llvm.org/docs/ThreadSanitizer.html`.

3 Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2015.

4 Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2015.

5 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2006.

6 Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: proportional detection of data races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

7 Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering*, pages 211–230, 2008.

8 Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

9 Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, 2014.

10 Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 467–484, 2012.

11 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

12 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

13 Cormac Flanagan and Stephen N Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2010.

14 Cormac Flanagan and Stephen N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming*, 2013.

15 Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

16 Jeff Huang and Charles Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ACM International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

17 Jeff Huang, Jinguo Zhou, and Charles Zhang. Scaling predictive analysis of concurrent programs by removing trace redundancy. *ACM Transactions on Software Engineering and Methodology*, 22(1):8:1–8:21, 2013.

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

19 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

20 Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

21 Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

22 Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. Sdnracer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 22:1–22:7, 2015.

23 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

24 Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

25 Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 151–166, 2013.

26 Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. BigFoot: Static check placement for dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

27 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

28 Koushik Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

29 Ohad Shacham, Mooly Sagiv, and Assaf Schuster. Scaling model checking of dataraces using dynamic information. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

30 Christoph von Praun and Thomas R. Gross. Object race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2001.

31 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. ESEC-Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering, 2007.

32 Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

33 James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–165, 2015.

34 Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles*, 2005.