Accelerating Dynamic Data Race Detection Using Static Thread Interference Analysis

Peng Di
School of Computer Science
and Engineering
University of New South Wales, Australia
pengd@cse.unsw.edu.au

Yulei Sui
School of Computer Science
and Engineering
University of New South Wales, Australia
ysui@cse.unsw.edu.au

ABSTRACT
Precise dynamic race detectors report an error if and only if more than one thread concurrently exhibits conflict on a memory access. They insert instrumentations at compile-time to perform runtime checks on all memory accesses to ensure that all races are captured and no spurious warnings are generated. However, a dynamic race check for a particular memory access statement is guaranteed to be redundant if the statement can be statically identified as thread interference-free. Despite significant recent advances in dynamic detection techniques, the redundant check remains a critical factor that leads to prohibitive overhead of dynamic race detection for multithreaded programs.

In this paper, we present a new framework that eliminates redundant race check and boosts the dynamic race detection by performing static optimizations on top of a series of thread interference analysis phases. Our framework is implemented on top of LLVM 3.5.0 and evaluated with an industry dynamic race detector TSAN which is available as a part of LLVM tool chain. 11 benchmarks from SPLASH2 are used to evaluate the effectiveness of our approach in accelerating TSAN by eliminating redundant interference-free checks. The experimental result demonstrates our new approach achieves from 1.4x to 4.0x (2.4x on average) speedup over original TSAN under 4 threads setting, and achieves from 1.3x to 4.6x (2.6x on average) speedup under 16 threads setting.

CCS Concepts
•Software and its engineering → Compilers; Software reliability; Software defect analysis; •Theory of computation → Concurrency;

Keywords
data race, concurrent program, static analysis, multithreaded

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PMAM ’16, March 12–16, 2016, Barcelona, Spain
© 2016 ACM. ISBN 978-1-4503-4196-7/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2883404.2883405

Figure 1: TSAN’s performance slowdown over native code for SPLASH2 benchmarks (under compiler option -O0).

1.1 Motivation
Languages like C and its variants are the de facto standard for implementing a wide variety of system software such as language runtime, device drivers, network servers and performance-critical software. A substantial number of these applications are written in multithreading paradigm (e.g., using pthread) to better utilize resources for parallel computing. However, multithreaded programs are notoriously prone to data race, which presents a big challenge for software testing and analysis.

A data race occurs when two threads concurrently perform conflicting memory accesses that read or write the same location, where at least one access is a write. The order in which the conflicting accesses are performed may affect the program’s subsequent state and behavior, likely with unintended or erroneous consequences. Such problems may arise only on rare interleavings, making them difficult to detect, reproduce.

The problems caused by data races have motivated much work on detecting races via static [1, 5, 33, 49, 26, 14, 16] or dynamic analysis [13, 28, 48, 52]. Among them, the dynamic approach is to have the compiler instrument the compiled code with runtime checks that ensure the safety of memory related accesses. For instance, ThreadSanitizer [39] (TSAN) that is an industry-strength precise dynamic race detector is available as part of LLVM tool chain. It consists of a com-
The authors present a new static analysis framework for accelerating TSAN, the state-of-the-art race detector for C/C++ using source-code level instrumentation. The approach gains from 1.4x to 4.0x (2.4x on average) speedup under 4 threads setting, and from 1.3x to 4.6x (2.6x on average) speedup under 16 threads setting.

2. DESIGN

We first describe a static thread model used for handling fork and join operations (Section 2.1). Then an overview of our framework in Figure 2 is given (Section 2.2) to show how to perform the subsequent static thread interference analysis phases for multithreaded programs (Section 2.3) and finally we discuss how to generate instrumentation code to perform runtime race check (Section 2.4).

2.1 Static Thread Model

2.1.1 Abstract thread

An abstract thread refers to a call of pthread_create() at a context-sensitive fork site during the analysis, so that a thread always refers to a context-sensitive fork site, i.e., a unique runtime thread unless it is multi-forked, in which case, it may represent more than one runtime thread.

Definition 1 (Multi-Forked Threads). A thread \( t \in M \) is a multi-forked thread if its fork site \( f_k \), resides in a loop, recursion or its spawner thread \( t' \in M \).

Figure 3 shows examples of context-sensitive abstract thread and multi-forked thread. A vertical line in the right subfigure is the execution order of one abstract thread. A horizontal line stands for a fork or join site and the spawning relation between two threads. It can be seen that the fork and join site of thread \( t_1[i] \) are nested by two “symmetric” loops separately, so that it is a multi-forked thread. We deal with an abstract multi-forked thread by LLVM’s SCEV alias analysis, discussed in Section 3.2. In addition, since main function calls bar twice (cs1 and cs2), the statements in bar, i.e., \( f_{k_2}, s_6 \) and \( j_{n_2} \), are executed twice by \( t_0 \) with different contexts. Hence, \( f_{k_2} \) spawns two different abstract threads \( t_2 \) and \( t_3 \).

2.1.2 Intra-thread Control Flow Graph

For an abstract \( t \), its intra-thread control flow structure of a thread is represented in a direct graph (aka. ICFG [23]), where a node \( s \) represents a program statement and a edge between statements corresponds to the control-flow.

We distinguish three kinds of control flow edges on ICFG: (1) an intra-procedure control flow \( s \xrightarrow{\text{intra}} s' \) from statement \( s \) to its successor \( s' \); (2) an interprocedural call edge \( s \xrightarrow{\text{call}[i]} s' \) from a call statement \( s \) to the entry statement \( s' \) of a callee via callsite \( i \); (3) an interprocedural return edge \( s \xrightarrow{\text{return}[i]} s' \) from a return statement \( s \) of a callee function to the statement \( s' \) immediate after the callsite \( i \) at a caller.
2.1.3 Modeling Thread Forks and Joins

In unstructured multithreaded programs, threads’ relations become more complicated. Imprecise modelling of synchronisations (e.g., fork/join, lock/unlock) may lead to spurious thread interleavings with overwhelming unrealizable relations that affects the precision of race detection.

![Figure 3: Examples of context-sensitive abstract thread and multi-forked thread.](image)

To model inter-thread relations between two threads, we use \( t \xrightarrow{(c, fj_k)} t' \) to represent the spawning relation that a spawned thread \( t' \) is directly transitive from \( t \) via a context-sensitive fork site \((c, fj_k)\), where \( c \) refers to the context stack, a sequence of classifies, \( [k_1 ... k_m] \), from program entry to the fork site \( fj_k \) with classifies. SCC are treated insensitively as intra-procedural control-flow.

For context stack \( c \)'s operations, \( c \oplus k \) denotes an operation for pushing a classify \( k \) into \( c \). \( c \ominus k \) denotes an operation for popping \( k \) from \( c \) if \( c \) contains \( k \) as its top value or is empty since a realizable path start and end in different functions.

For a thread \( t \) forked at \((c, fj_k)\), we write \( sr_t \) to stand for the start routine procedure of \( t \), where the execution of \( t \) begins. \( Entry(sr_t) = (c', s) \) maps \( sr_t \) to its the entry context-sensitive statement \((c', s)\), where \( c' = c \oplus i \) where thread \( t \) is forked at \((c, fj_k)\).

![Figure 4: Static modeling of fork and join operations.](image)

The spawning relation \( t \xrightarrow{(c, fj_k)} t' \) is transitive representing the fact that thread \( t \) can create \( t' \) directly or indirectly via fork site \( fj_k \) \((T\text{-FORK})\).

In pthread programs, a thread can be joined fully along all program paths or partially along some but not all paths. Two joining relation \( t \xrightarrow{(c,jn_i)} t' \) and \( t' \xrightarrow{(c,jn_j)} t'' \) is also transitive \((T\text{-JOIN})\) if and only if thread \( t' \) fully joins \( t'' \).

As our pre-analysis is flow- and context-insensitive, we achieve soundness by requiring \( t' \) joined at a join site \( pthread\_join() \) in the program to be excluded from \( M \), so that \( t' \) represents a unique runtime thread (under all contexts). Note that the joining relation is not transitive in the same sense as the spawning relation. In Pthreads programs, a thread can be joined fully along all program paths or partially along some but not all paths. Given \( t \xrightarrow{(c,jn_i)} t' \) and \( t' \xrightarrow{(c,jn_j)} t'' \), \( t \xrightarrow{(c,jn_i)} t'' \) holds when \( t' \xrightarrow{(c,jn_j)} t'' \) is a full join, denoted \( t' \xrightarrow{(c,jn_j)} t'' \).

If neither \( t \xrightarrow{(c,jn_i)} t' \) nor \( t' \xrightarrow{(c,jn_j)} t'' \) holds, then \( t \) and \( t' \) are siblings, denoted \( t \xrightarrow{\text{full}} t' \) \((T\text{-SIBLING})\). In this case, \( t \) and \( t' \) share a common ancestor thread \( t'' \), where \( t \neq t' \) and \( t \neq t'' \). Furthermore, \( t \) and \( t' \) do not happen-in-parallel if one happens before the other (as defined below).

**Definition 2** (Happens-Before (HB) Relation for Sibling Threads). *Given two sibling threads \( t \) and \( t' \), \( t \) happens before \( t' \), denoted \( t \xrightarrow{\text{HB}} t' \), if the fork site of \( t' \) is backward reachable to a join site of \( t \) along every program path.***

Algorithm 1 presents how to build the Thread Create Tree (TCT) \( \{t\} \) which represents a set of spawning relations \((T\text{-FORK})\) of the whole program. The algorithm performs iterative data flow analysis on call graph to process fork edges and call edges until a fixed point is reached. As is standard, recursion cycles discovered in the call graph of the program are collapsed into SCCs. The context-sensitivity is ignored inside a SCC. \( W \) represents the worklist containing context- and thread-sensitive procedure tuples in the form of \((c, t, p)\) for iterative resolution. \( T(t) \) maps a thread to its context-sensitive forksite. We assume a program starts with a default main thread at the entry of the main function with empty context and a dummy forksite.

2.2 Framework

We present a new race detection framework for handling multithreaded programs in Figure 2.

As shown in Figure 2, our framework, which is implemented in LLVM, comprises seven phases (described below) with five thread interference analyses. The arrows in the figure denote the order in which the phases and analyses are performed. In “Memory Pairs Collection”, every write-write or write-read pair in a program is first collected. For example, in Figure 5, statements \( s_1 \) and \( s_2 \) operate on variable \( p \), and they are treated as a memory access pair \((s_1, s_2)\). Five optimizations are then performed to refine the over-approximated sets of memory accesses pairs which may be involved in a race. They are “Reachability Optimization”, “Interleaving Optimization”, “Alias Optimization”, “Thread-Local Optimization” and “Lockset Optimization”. These optimizations rely on five static thread interference analyses: “Call Graph Construction”, “Interleaving Analysis”, “Pointer Analysis”, “Thread-Local Analysis” and “Lockset Analysis”. The optimization (on the right of the red dashed line) and...
2.3 Static Analyses

As shown in Figure 2, our framework consists of five analyses passes to statically infer non-interference pairs to accelerate dynamic data race detection. This section focuses on how to perform these five static analyses for refining the original pairs that are collected during “Memory Pairs Collection” phase.

2.3.1 Reachability Analysis

The reachability analysis prunes pairs using the fact that a pair of accesses may be involved in a race only if each access is reachable from main function of the program.

The set of reachable pairs, a subset of original pairs, is computed by traversal of call graph to eliminate accesses in dead functions. For instance, in Figure 5, function bar is not called through main direct or indirectly, so that the statement s7 cannot be executed by any active threads. There-

fore, the pairs made up by s7 is pruned from potential race pairs.

2.3.2 Interleaving Analysis

As shown in Figure 2, our static framework invokes interleaving analysis to compute pairs potentially involved in a race. The objective here is to reason about fork and join operations to identify all may-happen-in-parallel statements in the program.

Our interleaving analysis operates flow- and context-sensitively on the ICFGs of all the threads (but uses points-to information from the pointer analysis). For a statement s in thread t’s ICFG, our analysis approximates which threads may run in parallel with t when s is executed, denoted as I(t, c, s), where c is a calling context to capture one instance of s when its enclosing method is invoked under c. For example, if I(t1, c, s) = {t2, t3}, then threads t2 and t3 may be alive when s1 is executed under context c in t1.

Statement s1 in thread t1 may happen in parallel with statement s2 in thread t2, denoted as (t1, c1, s1) || (t2, c2, s2), if the following holds (with M from Definition 1):

\[
\begin{cases}
    t_2 \in I(t_1, c_1, s_1) \land t_1 \in I(t_2, c_2, s_2) & \text{if } t_1 \neq t_2 \\
    t_1 \in M & \text{otherwise}
\end{cases}
\]

(1)

Given (c, fk) (spawning relation), (c, jn) (joining relation), ≈ (thread sibling) and ∪ (HB from Definition 2), that are collected by performing Algorithm 1, our interleaving algorithm is formulated as solving a forward data-flow problem with semilattice (V, ∪, F). Here, V represents the set of all thread interleaving facts, ∪ is the meet operator (set union ∪). F : V → V represents the set of transfer functions associated with each node on ICFGs.

Our interleaving analysis is presented in Algorithm 2. The algorithm uses a standard data-flow analysis by propagating interleaving information iteratively along inter-procedural program control-flow. For a context-sensitive fork site (c, fk)
The code snippet is a description of an algorithm for interleaving analysis in a context-sensitive thread. The algorithm involves the following steps:

1. Define the context-sensitive interleaving information $I$.
2. Let $Entry(sr_t)$ be the entry of start routine function $t$.
3. Let $s_{tf}$ be the statement immediately after $t$’s fork site.
4. For each context-sensitive thread $t$ in TCT, perform the following:
   - Let $(c, fk)$ be the context-sensitive fork site which creates $t$.
   - $W := W \cup \{(t, c, Entry(sr_t))\}$.
   - Update ascendant threads $*$.
5. For each $t_{asc} \in Asc(t)$, perform the following:
   - Let $(c', fk')$ be the context-sensitive fork site which creates $t_{asc}$.
   - $I(t_{asc}, c', sa_{t_{asc}}) := I(t_{asc}, c', sa_{t_{asc}}) + < t >$.
   - If $I(t_{asc}, c', sa_{t_{asc}})$ changes then:
     - $W := W \cup \{(t_{asc}, c', sa_{t_{asc}})\}$.
   - Update sibling threads $*$.
6. For each $t_{sib} \in Sib(t_{asc})$, perform the following:
   - If $t_{sib} \neq t_{asc}$ and $t_{sib} \neq t_{asc}$ then:
     - $(c'', fk'') := TCT(t_{sib})$.
     - $I(t_{sib}, c'', sr_{t_{sib}}) := I(t_{sib}, c'', sr_{t_{sib}}) + < t >$.
     - If $I(t_{sib}, c'', sr_{t_{sib}})$ changes then:
       - $W := W \cup \{(t_{sib}, c'', sr_{t_{sib}})\}$.
7. While $W \neq \emptyset$ do the following:
   - $(t, c, s) := \text{SELECT}(W)$.
   - If $s$ is fork site $t$ then:
     - $I(t', c \oplus i, s_t) := I(t', c \oplus i, s_t) \cap I(t, c, s) + < t >$.
     - If $I(t', c \oplus i, s_t)$ changes then:
       - $W := W \cup \{(t', c \oplus i, s_t)\}$.
   - If $s$ is join site of $t$ then:
     - $I(t, c, s) := I(t, c, s) / < t >$.
   - For each outgoing control flow edge $ss \leftarrow s$ from $s$ do:
     - If $ss_{\text{intra}}$ then:
       - $c' := c$.
     - Else if $ss_{\text{call}[i]}$ then:
       - $c' := c \oplus i$.
     - Else if $ss_{\text{ret}[i]}$ then:
       - $c' := c \oplus i$.
     - $I(t, c', ss) := I(t, c', ss) \cap I(t, c, s)$.
     - If $I(t, c', ss)$ changes then:
       - $W := W \cup \{(t, c', ss)\}$.

Algorithm 3: Get Ascendant Threads Asc(t)

* Return the thread set Asc that is the ancestor of $t$ and $t$ itself $* /$
1. Asc := $\{t\}$;
2. foreach $t'$ in TCT do
   3. if $t' \Rightarrow t$ then
      4. Asc := Asc $\cup \{t'\}$
3. return Asc;

Algorithm 4: Get Sibling Threads Sib(t)

* Return the thread set Sib that is the sibling of $t$ $* /$
1. Sib := $\emptyset$;
2. foreach $t'$ in TCT do
   3. if $t \Rightarrow t'$ then
      4. Sib = Sib $\cup \{t'\}$
3. return Sib;

Next, the entry statements of the start routines of $t_{asc}$’s sibling threads $t_{sib}$ (Algorithm 4) is updated with $t$ (line 10-16). The ascendant and sibling threads are computed in Algorithm 3 and 4.

Then, an iterative flow- and context-sensitive analysis is performed on the inter-procedural control-flow by matching calls and returns (line 33-34). If the interleaving information $I$ of a context- and thread-sensitive statement changes, it will be added into the work list for fixed point resolution (line 19-36). For a fork site during resolution, the spawner thread $t$ is added to the interleaving set of the entry statement in a spawner’s start routine procedure (line 19-20). For a join site, fully joined thread is removed from the current interleaving set (line 23-24).

In the example shown in Figure 5, the interleaving information of six statements ($s_7$ is pruned in the previous optimization) is as follows: $I(t_0, c, s_1) = \{\}, I(t_0, c, s_2) = \{t_1\}, I(t_0, c, s_3) = \{t_1\}, I(t_1, c, s_4) = \{t_0\}, I(t_1, c, s_5) = \{t_0\}, I(t_1, c, s_6) = \{t_0\}$. According to Equation 1, $s_1$ cannot happen in parallel with the other statements, so that the pairs of $s_1$ are pruned. In addition, Equation 1 guarantees that the pair executed by the same non-multi-forked thread cannot happen in parallel, therefore, pair $(s_5, s_6)$ is also pruned.

### 2.3.3 Pointer Analysis

We use Andersen’s inclusion-based pointer analysis to prune alias pairs. The implemented analysis is field-sensitive. Each field of a struct is treated as a separate object, but arrays are considered monolithic. Distinct allocation sites are modeled by distinct abstract objects. We use the wave propagation technique [31, 34] for constraint resolution. The positive weight cycles (PWCs) [30] are detected using Nuutila’s SCC detection algorithm [29]. A program’s call graph is built on the fly and points-to sets are represented using sparse bit vectors.

Alias optimization obtains alias information from pointer...
analysis and prunes the non-aliased pairs. For instance, the statements \( s_3 \) and \( s_5 \) in Figure 5 operate two pointers \( p \) and \( q \) that points to different objects \( a \) and \( b \). Therefore, \( p \) and \( q \) are not aliased here and the pair \((s_3, s_5)\) will be pruned.

### 2.3.4 Thread-Local Analysis

The fourth phase of our approach refines alias pairs using thread-local analysis that identifies whether objects escape from thread or not. An object is not thread-local if it escapes via some arguments at a fork site or it escapes via global pointers. Thread-local analysis depends on our pointer analysis results.

Algorithm 5 presents how our thread-local analysis identifies escaped objects. In first step, the algorithm collects the objects that are pointed to by arguments of fork site (line 1–3) and global pointers (line 4–6). Then the algorithm recursively collects all other objects (field-sensitively) that may be pointed to by the identified objects (line 7–13).

Our optimization keeps the pairs that access thread escaped objects, because only such a kind of pair may be involved in a data race. Take the statements \( s_2 \) and \( s_4 \) in Figure 5 as an example, \( x \) accessed by \( s_4 \) is a thread-local object so that \( s_4 \) does not occurs data race with other accesses. Therefore, the pair \((s_2, s_4)\) is pruned.

### 2.3.5 Lockset Analysis

Statements from different mutex regions are interference-free if these regions are protected by a common lock. By capturing lock correlations, many pairs locked by a common lock can be pruned, such as \((s_2, s_6)\) in Figure 5. The optimizer does this by performing a flow- and context-sensitive analysis for lock/unlock operations (based on the points-to information).

The implementation of lockset analysis is similar to interleaving analysis. Firstly, all the context-sensitive locks are collected. Then we iteratively resolves context-sensitive lock spans for context-sensitive statements on inter-procedural CFG until a fixed point is reached.

### 2.4 Guided Instrumentation

Original TSAN’s strategy is to check all variables and statements in a program, expect \texttt{const} variables. Actually, a statement \( s \) may need to be checked only if exists a pair with \( s \) in the static result pairs set that is refined by all optimizations. Instead of TSAN’s full instrumentation, our static approach refines and prunes the original memory pairs by performing a set of optimizations and only a subset of all pairs to be instrumented at run time.

The implementation contains two steps: annotation and instrumentation. Based on the refined paired generated after the five static optimization phases (Figure 2), an instruction is annotated for checking if there exists a refined pair containing this instruction. Then, we simply enable TSAN’s instrumentor to instrument TSAN’s library functions at annotated memory accesses for runtime check.

### 3. EVALUATION

The objective is to show that our static may-happen-in-parallel analysis enables to significantly accelerate TSAN to check multithreaded programs using Pthreads.

#### 3.1 Experiment Setup

We have selected a set of 11 multithreaded programs from SPLASH2 benchmark, as shown in Table 1. All our experiments were conducted on a platform consisting of a 3.00GHz Intel Xeon(R) Quad E5450 processor with 32 GB memory, running Ubuntu Linux (kernel version 3.11.0).

The source code of each program is compiled into bit code files using clang and then merged together using LLVM Gold Plugin at link time stage (LTO) to produce a whole-program bc file. In addition, the compiler option mem2reg is turned on to promote memory into registers.

#### 3.2 Implementation

We have implemented our approach in LLVM (version 3.5.0). Andersen’s analysis (using the constraint resolution techniques from [31]) is used to perform its pointer analysis indicated in Figure 2.

In order to distinguish the concrete runtime threads represented by an abstract multi-forked thread (Definition 1) inside a loop, we use LLVM’s SCEV alias analysis to correlate a fork-join pair. Figure 6 shows a code snippet from ocean_ncp, where a fixed number of threads are forked and joined in two “symmetric” loops. Our interleaving can recognize that any statement in a spawner thread (with its start routine slave) does not happen in parallel with the statements after its join executed in the main thread.

#### 3.3 Results and Analysis

Algorithm 5: Thread-Local Analysis

<table>
<thead>
<tr>
<th>Data: Points-to information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: ( \text{Set}_{nlti} ): the set of all escaped objects</td>
</tr>
<tr>
<td>1 foreach argument pointer ( ap ) passing into a spawned procedure of a fork site do</td>
</tr>
<tr>
<td>2 ( \text{Pts}(ap) ): represents the points-to targets of pointer argument ( ap )</td>
</tr>
<tr>
<td>3 ( W := W \cup \text{Pts}(ap) )</td>
</tr>
<tr>
<td>4 foreach global pointer ( gp ) do</td>
</tr>
<tr>
<td>5 ( \text{Pts}(gp) ): represents the points-to targets of global pointer ( gp )</td>
</tr>
<tr>
<td>6 ( W := W \cup \text{Pts}(gp) )</td>
</tr>
<tr>
<td>7 while ( W \neq \emptyset ) do</td>
</tr>
<tr>
<td>8 ( o := \text{SELECT}(W) )</td>
</tr>
<tr>
<td>9 ( \text{Set}<em>{nlti} := \text{Set}</em>{nlti} \cup {o} )</td>
</tr>
<tr>
<td>10 ( \text{Pts}(o) ): represents the points-to targets of ( o )</td>
</tr>
<tr>
<td>11 foreach ( o' \in \text{Pts}(o) ) do</td>
</tr>
<tr>
<td>12 ( W := W \cup {o'} )</td>
</tr>
</tbody>
</table>

Figure 6: A multi-forked thread example in \texttt{ocean_ncp} from the SPLASH2 benchmark suite.
3.3.1 Static Analysis

Table 1 gives the static statistic. For each benchmark, we list the the static number of invocation numbers of Pthread API, including Fork, Join, Lock, Trylock and Unlock. We do not handled Wait and Signal statements, as in prior work, resulting in sound, i.e., over-approximate results.

The “Instrumented by TSAN” shows the number of instrumented write and read statements for original TSAN. The implementation of TSAN adds a simple refinement to eliminates “const” variables from the need-check memory locations. “Ignored” shows the number of refined statements that access “const” variables.

The “Instrumented by Ours” shows the number of instrumented write and read statements obtained by our new framework. After pruning of static optimizations, our approach can identify an average of 46.7% fewer instrumented statements than TSAN (achieving 60.5% at lu_cb).

3.3.2 Dynamic Execution

At run time, the “native” data set is chosen as the input of SPLASH2 benchmarks. Since runtime thread scheduling might be different among each execution, the result of data race detection is affected by scheduling and thus difficult to reproduce. Therefore, every benchmark is tested ten times to record all appearing conflicts and indicate all memory operations with race warning. The experimental result shows that every single warned statement detected by original TSAN and optimised TSAN is identical, meanwhile our static analyser accelerates the dynamic execution from 1.4x to 4.0x (2.4x on average) under 4 threads setting, and from 1.3x to 4.6x (2.6x on average) under 16 threads setting, shown in Figure 7.

4. RELATED WORK

We discuss the related work on thread interference analysis and race detection for multithreaded programs.

4.1 Interleaving Analysis for Multithreaded Programs

Past interleaving analyses have utilized a variety of techniques for discovering whether two statements may execute in parallel. In early works, Bristow et al. builds an abstraction, called Interprocess Precedence Graph, to indicate the synchronization-imposed execution ordering among processes [6]. Taylor models a concurrency graph based on a reduced flow graph representation of every task [46]. Several projects tackled the concurrency-determination problem by deducing complementary knowledge, such as partial execution orders and the Cannot-Happen-Together (CHT) relation[7, 11, 25].

When considering the high level programming models, attentions are paid to structured programming languages with restricted structures, such as Clik, X10, etc. The interleaving analysis turns to be more effective due to the simplified problem based on async-finish parallelism model [2, 24].

In the case of unstructured languages, such as C, the interleaving analysis is confronted with substantial challenges. A number of studies have appeared, introducing a variety of advanced techniques for discovering interleaving information in a program [27, 26]. Joisha et al. [21] present a coarse-grained analysis based on Procedural Concurrency Graphs (PCGs) to detect interleaving information at the level of procedures. Chen et al. [8] introduced a graph-based interleaving algorithm with a context-insensitive thread model. Shin et al.[40] presents a power-gating analysis framework (MTPG) for multithreaded programs. MTPG analyzes MHP information among threads to report the component usages shared by multiple threads in hierarchical BSP models. Compared to the above interleaving analyses, our work enables fine-grained flow- and context-sensitive analysis that achieves improved precision for C.
4.2 Pointer Analysis for Multithreaded Programs

Compared to pointer analysis for sequential programs [17, 45, 43, 44, 51, 19], flow-sensitive analysis for multithreaded programs is relatively unexplored. Earlier, Rugina and Rinard [35] introduced a pointer analysis for Clik programs with structured parallelism. They solved a standard dataflow problem to propagate points-to information iteratively along the control flow and evaluated their analysis with benchmarks with up to 4500 lines of code.

However, unstructured multithreaded C or Java programs are more challenging to analyze due to the use of non-lexically-scoped synchronization statements (e.g., fork/join and lock/unlock). For Java programs, a compositional approach [36] analyzes pointer and escape information of variables in a method that may be escaped and accessed by other threads. The approach performs a flow-sensitive lock-free analysis to analyze each method modularly but iteratively without considering strong updates. Flow-sensitivity is important to achieve precision required for C programs. The prior analyses on handling thread synchronizations are conservative, by ignoring locks [36] or joins [21] or dealing with only partial and/or nested joins [4]. In contrast, our work models such synchronization operations more accurately to produce precise results that can successfully guide instrumentations that reduce runtime checking overhead.

4.3 Data Race Detection for Multithreaded Programs

In recent years, some static race detecting tools have been presented [22, 33, 3] to analyze Posix C programs. Locksmith uses existential types to correlate locks and data in dynamic data structures [33]. Goblint relies on a sound thread-modular constant propagation and points-to analysis, with considering conditional locking schemes [47]. Relay presents an approach that enables to scale to millions of lines of code. But it is typically sound except for a few exceptions, and limits false positives as much as possible by performing several filters [49].

Most typical dynamic race detectors are based on two techniques: Lockset computation [37], Happens-Before (HB) ordering [9, 15], and a hybrid of these two [52, 50, 13, 32]. Lockset-based algorithms attempt to detect inconsistent use of locks by different threads [37]. These approaches are able to detect those potential races that are not observed in the execution being monitored with low overhead, but they are imprecise (by reporting false positives) because ignoring the ordering of events in program executions. The HB-based approaches are precise theoretically, but they are often very limited in detecting races because of conservative HB edges, and have a large of runtime overhead as they need to track all memory accesses. The state-of-practice HB-based detector TSAN adopts improved FastTrack [15] to achieve the better performance. Hybrid techniques focus on combining Lockset and HB to extend the detecting coverage of HB-based approach, but possibly report false positives [50]. To guarantee a sound result, Causally-Precedes [41] requires manual post-processing to refine false positives. Huang et al. present a sound predictive race detection technique to achieve the maximal possible detection capability with respect to the same input trace under the sequential consistency memory model [18].

Several other approaches exist to deal with data races, such as model checking [28]. Race warnings are obtained by running the target program with many different thread schedules, either concretely or symbolically. RaceFuzzer [38] controls a randomized scheduler and creates an actual race condition to detect races. IFRit [12] performs efficient dynamic data-race detection by exploring interference-free region via analysis of lock acquire and release operations. IFRit is faster than FastTrack, but may miss some races. In contrast, our work identifies redundant memory checks and reports races as accurate as TSAN.

5. CONCLUSION AND FUTURE WORK

This paper presents a new static flow- and context-sensitive analysis framework to eliminate interference-free check of dynamic data detector by performing a series of thread interference analysis phases. Our framework is implemented on top of LLVM 3.5.0 and effectively accelerates precise dynamic race detector TSAN. 11 programs from SPLASH2 benchmarks is used to evaluate the effectiveness of our techniques. The experimental result demonstrates our framework is 1.4x to 4.0x (2.4x on average) faster than original TSAN under 4 threads setting, and 1.3x to 4.6x (2.6x on average) faster under 16 threads setting.

This paper is an extension of our previous work [42, 10]. In future work, we will extend our framework to support C++ programs. We also plan to combine this framework with POCL [20] library to develop race detection techniques for high-level programming languages in heterogeneous systems.

Acknowledgments

We thank all the reviewers for their constructive comments on an earlier version of this paper. This research is supported by ARC grants DP130101970 and DP150102109.

6. REFERENCES


