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

PMAM'16, March 12-16, 2016, Barcelona, Spain

 ISBN 978-1-4503-4196-7/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2883404.2883405

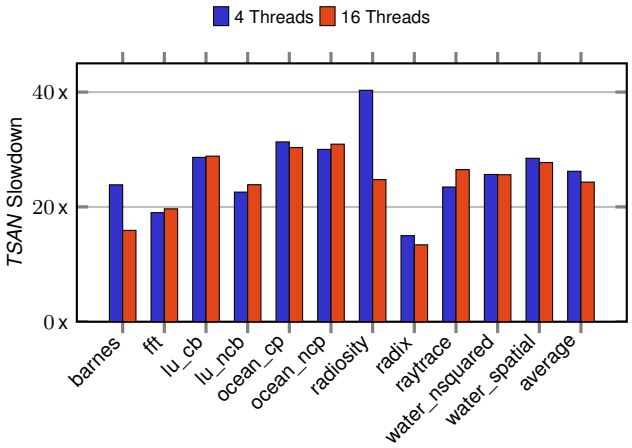


Figure 1: *TSAN*'s performance slowdown over native code for SPLASH2 benchmarks (under compiler option -O0).

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-

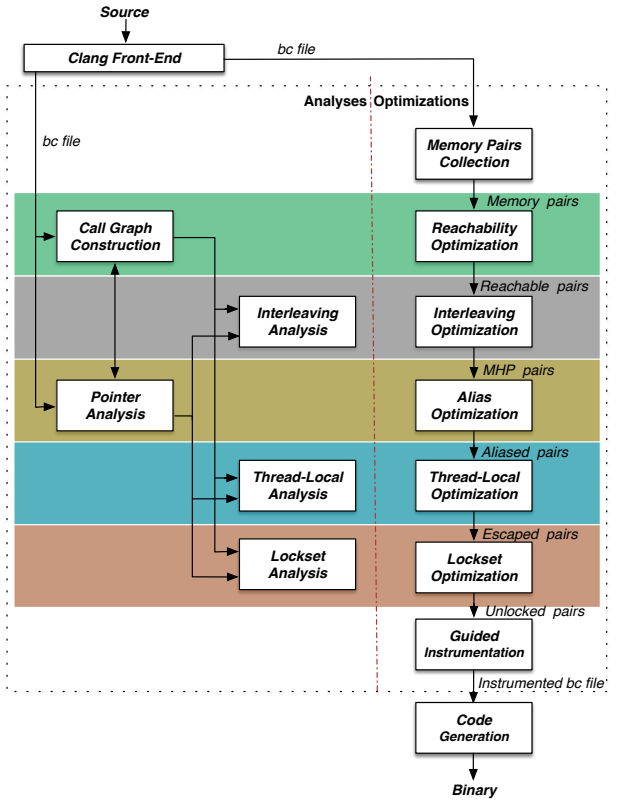


Figure 2: Static analyses and optimizations framework. Our main work is dotted framed.

piler instrumentation module and a run-time library to perform dynamic detection algorithm similar to FastTrack[15].

However, despite the development of a variety of implementation techniques (including vector clocks [32], epochs [52], accordion clocks [9], and others), the overhead of precise dynamic race detectors can still be prohibitive. Figure 1 shows the *TSAN*'s slowdown on SPLASH2 benchmarks with NATIVE inputs. The SPLASH2 slowdowns introduced by *TSAN* is from 15.0x to 40.3x, 26.2x on average, on 4 threads setting and from 13.4x to 30.9x, 24.3x on average, on 16 threads setting on a machine with Intel Xeon 4-core CPU E5450 3.0GHz and 32GB memory, running ubuntu linux.

1.2 Contributions

In this paper, we present a static flow- and context-sensitive thread interference analysis framework shown in Figure 2 for identifying and eliminating redundant race checks. The acceleration of dynamic data race detection is obtained by performing a series of static thread interleaving analysis phases and the source-level instrumentation for multithreaded C programs with Pthreads. We demonstrate its effectiveness against *TSAN* using all the 11 SPLASH2 programs. Specifically, this paper makes the following contributions:

- We present a new static analysis framework for accelerate dynamic race detection of multithreaded C programs with pthreads.
- We describe a series thread interference analyses for pruning memory access pairs so that it is sufficiently accurate in eliminating the redundant checks for multithreaded programs.
- We have fully implemented our approach on top of LLVM (version 3.5.0) and evaluated using 11 multithreaded programs from SPLASH2 benchmarks. The results shows its effectiveness in accelerating *TSAN*, the state-of-the-art race detector for C/C++ using source-code level instrumentation. Our 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* t refers to a call of pthread_create() at a context-sensitive fork site during the analysis[1], so that a thread t always refers to a context-sensitive fork site, i.e., a unique runtime thread unless t is multi-forked, in which case, t may represent more than one runtime thread.

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

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. fk_2, s_6 and jn_2, are executed twice by t_0 with difference contexts. Hence, fk_2 spawns two different abstract threads t_2 and t_2'.

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 $s \xrightarrow{intra} s'$ from statement s to its successor s'; (2) an interprocedural call edge $s \xrightarrow{call[i]} s'$ from a call statement s to the entry statement s' of a callee via callsite i; (3) an interprocedural return edge $s \xrightarrow{ret[i]} s'$ from a return statement s of a callee function to the statement s' immediate after the callsite i at a caller.

[1]We assume a program starts with a default root thread at the entry of the main function.

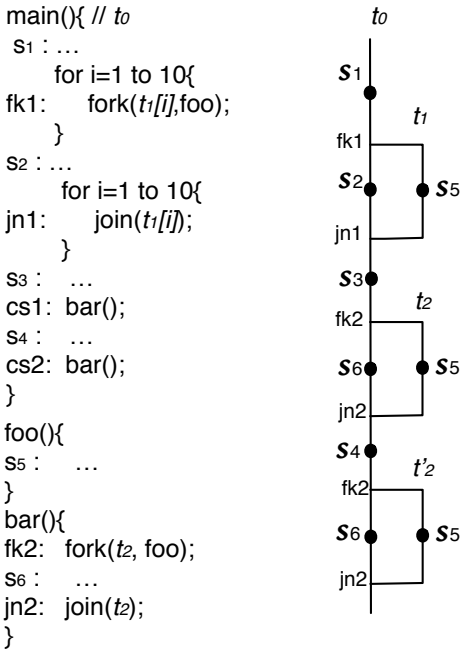


Figure 3: Examples of context-sensitive abstract thread and multi-forked thread.

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.

$$\texttt{[T-FORK]}\quad \frac{t \xrightarrow{(c,fk_i)} t' \quad t' \xrightarrow{(c',fk_{i'})} t''}{t \xrightarrow{(c,fk_i)} t''}$$

$$\texttt{[T-JOIN]}\quad \frac{t \xleftarrow{(c,jn_i)} t' \quad t' \xleftarrow[\text{full}]{(c',jn_{i'})} t''}{t \xleftarrow{(c,jn_i)} t''}$$

$$\texttt{[T-SIBLING]}\quad \frac{t'' \xrightarrow{(c,fk_i)} t \quad t'' \xrightarrow{(c',fk_{i'})} t' \quad (i \neq i' \vee c \neq c')}{t \bowtie t'}$$

Figure 4: Static modeling of fork and join operations.

To model inter-thread relations between two threads, we use $t \xrightarrow{(c,fk_i)} t'$ to represent the spawning relation that a *spawner thread* t creates a *spawnee thread* t' via a context-sensitive fork site (c, fk_i), where c refers to the context stack, a sequence of callsites, $[\kappa_1 \ldots \kappa_m]$, from program entry to the fork site fk_i with callsites. SCC are treated insensitively as intra-procedural control-flow.

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

For a thread t forked at (c, fk_i), 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, fk_i).

The spawning relation $t \xrightarrow{(c,fk_i)} t'$ is transitive representing the fact that thread t can create t' directly or indirectly via fork site fk_i ([T-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 $t \Leftarrow t'$ and $t' \Leftarrow t''$ is also transitive ([T-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 $\mathcal{M}$, 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 $t \xleftarrow{(c,jn_i)} t'$ and $t' \xleftarrow{(c',jn_{i'})} t''$, $t \xleftarrow{(c,jn_i)} t''$ holds when $t' \xleftarrow{(c',jn_{i'})} t''$ is a full join, denoted $t' \xleftarrow[\text{full}]{(c',jn_{i'})} t''$.

If neither $t \xrightarrow{(c,fk_i)} t'$ nor $t' \xrightarrow{(c',fk_{i'})} t$ holds, then t and t' are *siblings*, denoted $t \bowtie t'$ ([T-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 $t \succ t'$, 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) [4] which represents a set of spawning relations ([T-FORK]) of the whole program. The algorithm performance 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 two writes 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

Algorithm 1: TCT Construction

Data: Call graph
Result: Context-sensitive thread creation tree

Call graph SCC detection.
Let F be all fork sites of the program.
Let t_m be the main thread and p_m be main procedure of the program.
Let sr_t be the start routine procedure of thread t
$W := W \cup \{(\varnothing, t_m, p_m)\}$;
while $W \neq \varnothing$ **do**
  $(c, t, p) := SELECT(W)$;
  foreach *context-sensitive fork edge* $(c, t, p) \xrightarrow{fk_i} (c \oplus i, t', p')$ *from procedures* p **do**
    $TCT := TCT \cup \{t \xrightarrow{(c, fk_i)} t'\}$
    $T(t') := (c, fk_i)$
    $W := W \cup \{c \oplus i, t', sr_{t'}\}$
  foreach *context-sensitive call edge* $(c, t, p) \xrightarrow{i} (c \oplus i, t, p')$ *from procedures* p **do**
    $W := W \cup \{c \oplus i, t, p'\}$

its dependent analysis (on the left of the red dashed line) are framed into the same colourful block. The bi-directional arrow between "Call Graph Construction" and "Pointer Analysis" denotes that they are mutually dependent and are computed simultaneously. Also, the other three analyses depend upon both the call graph construction and pointer analysis.

Figure 5 presents how these optimizations refine the interference -free pairs phase by phase. In this example, spawner thread t_0 creates spawnee thread t_1, there are seven memory access (write or read) statements labelled from s_1 to s_7. All memory access pairs are shown in the following table, where the column and row of each entries stand for two statements of a pair. "R" means that this pair will be pruned by "Reachability Optimization". Similarly, "I", "A", "T" and "L" stand for pruned pairs by "Interleaving Optimization", "Alias Optimization", "Thread-Local Optimization" and "Lockset Optimization" respectively. "P" means that the two accesses may happen in parallel, which may lead to a race at runtime.

Note that a pair may satisfy pruning requirements of more than one optimization. Such a pair is labelled with multiple optimizations that any of them can prune this pair.

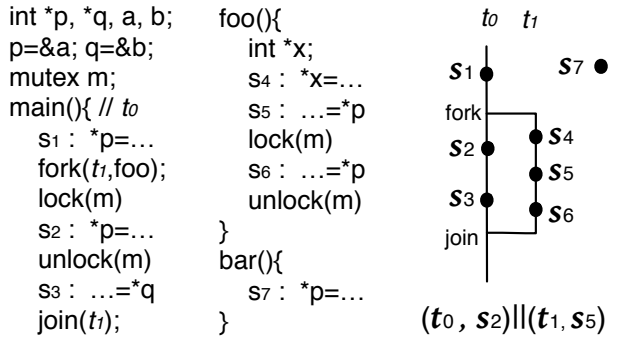
2.3 Static Analyses

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 s_7 cannot be executed by any active threads. Therefore, the pairs made up by s_7 is pruned from potential race pairs.



	s_1	s_2	s_3	s_4	s_5	s_6	s_7
s_1		I	I	IAT	I	I	R
s_2			IA	AT	P	L	R
s_3				AT	A	A	R
s_4					IT	IT	R
s_5						I	R
s_6							R
s_7							

Figure 5: An example of pruning thread interference-free memory access pairs. $\|$ denotes may-happen-in-parallel.

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_t, our analysis approximates which threads may run in parallel with t when s is executed, denoted as $\mathcal{I}(t, c, s)$, where c is a calling context to capture one instance of s when its enclosing method is invoked under c. For example, if $\mathcal{I}(t_1, c, s) = \{t_2, t_3\}$, then threads t_2 and t_3 may be alive when s_1 is executed under context c in t_1.

Statement s_1 in thread t_1 may happen in parallel with statement s_2 in thread t_2, denoted as $(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)$, if the following holds (with $\mathcal{M}$ from Definition 1):

$$\begin{cases} t_2 \in \mathcal{I}(t_1, c_1, s_1) \wedge t_1 \in \mathcal{I}(t_2, c_2, s_2) & \text{if } t_1 \neq t_2 \\ t_1 \in \mathcal{M} & \text{otherwise} \end{cases} \quad (1)$$

Given $\xrightarrow{(c, fk_i)}$ (spawning relation), $\xleftarrow{(c, jn_i)}$ (joining relation), $\bowtie$ (thread sibling) and $\succ$ (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 $(V, \sqcap, F)$. Here, V represents the set of all thread interleaving facts, $\sqcap$ is the meet operator (set union $\cup$). $F : V \to V$ 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)

Algorithm 2: Interleaving Analysis

Data: TCT
Result: Context-sensitive interleaving information $\mathcal{I}$

1 Let $Entry(sr_t)$ be the entry of start routine function of t
2 Let sa_t be the statement immediately after t's fork site
3 **foreach** *context-sensitive thread t in TCT* **do**
  Let (c, fk) be the context-sensitive fork site which creates t
4 $W := W \cup \{(t, c, Entry(sr_t))\}$;
  `/* update ascendant threads */`
5 **foreach** $t_{asc} \in Asc(t)$ **do**
6 Let (c', fk') be the context-sensitive fork site which creates t_{asc}
7 $\mathcal{I}(t_{asc}, c', sa_{t_{asc}}) := \mathcal{I}(t_{asc}, c', sa_{t_{asc}}) + < t >$;
8 **if** $\mathcal{I}(t_{asc}, c', sa_{t_{asc}})$ *changes* **then**
9 $W := W \cup \{(t_{asc}, c', sa_{t_{asc}})\}$
  `/* update sibling threads */`
10 **foreach** $t_{asc} \in Asc(t)$ **do**
11 **foreach** $t_{sib} \in Sib(t_{asc})$ **do**
12 **if** $t_{sib} \not> t_{asc}$ *and* $t_{sib} \not< t_{asc}$ **then**
13 $(c'', fk'') = TCT(t_{sib})$
14 $\mathcal{I}(t_{sib}, c'', sr_{t_{sib}}) := \mathcal{I}(t_{sib}, c'', sr_{t_{sib}}) + < t >$;
15 **if** $\mathcal{I}(t_{sib}, c'', sr_{t_{sib}})$ *changes* **then**
16 $W := W \cup \{(t_{sib}, c'', sr_{t_{sib}})\}$
17 **while** $W \neq \emptyset$ **do**
18 $(t, c, s) := SELECT(W)$;
19 **if** *s is fork site* $t \xrightarrow{(c,s)} t'$ **then**
20 $\mathcal{I}(t', c \oplus i, s_{t'}) := \mathcal{I}(t', c \oplus i, s_{t'}) \sqcap \mathcal{I}(t, c, s) + < t >$
21 **if** $\mathcal{I}(t', c \oplus i, s_{t'})$ *changes* **then**
22 $W := W \cup \{(t', c \oplus i, s_{t'})\}$
23 **if** *s is join site of* $t \xleftarrow{(c,s)} t'$ **then**
24 $\mathcal{I}(t, c, s) := \mathcal{I}(t, c, s) \;/\!/ < t' >$
25 **foreach** *outgoing control flow edge* $ss \leftarrow s$ *from s* **do**
26 **if** $ss \xleftarrow{intra} s$ **then**
27 $c' := c$
28 **else if** $ss \xleftarrow{call[i]} s$ **then**
29 $c' := c \oplus i$
30 **else if** $ss \xleftarrow{ret[i]} s$ **then**
31 $c' := c \ominus i$
32 $\mathcal{I}(t, c', ss) := \mathcal{I}(t, c', ss) \sqcap \mathcal{I}(t, c, s)$;
33 **if** $\mathcal{I}(t, c', ss)$ *changes* **then**
34 $W := W \cup \{(t, c', ss)\}$

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** $t' \xrightarrow{(c,fk)} t$ **then**
4 $ASC := ASC \cup \{t'\}$
5 **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** $t \bowtie t'$ **then**
4 $SIB = SIB \cup \{t'\}$
5 **return** SIB;

which creates thread t, the algorithm first analysis the interleaving information of its ascendant threads by updating $\mathcal{I}$ of the statements that locate immediately after the fork sites of t's ascendant threads t_{asc} with t (line 5-9).

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 $\mathcal{I}$ 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 spawnee'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: $\mathcal{I}(t_0, c, s_1) = \{\}$, $\mathcal{I}(t_0, c, s_2) = \{t_1\}$, $\mathcal{I}(t_0, c, s_3) = \{t_1\}$, $\mathcal{I}(t_1, c, s_4) = \{t_0\}$, $\mathcal{I}(t_1, c, s_5) = \{t_0\}$, $\mathcal{I}(t_1, c, s_6) = \{t_0\}$. 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

Algorithm 5: Thread-Local Analysis

Data: Points-to information
Result: Set_{ntl}: the set of all escaped objects

1 **foreach** *argument pointer ap passing into a spawnee procedure of a forksite* **do**
2 $Pts(ap)$ represents the points-to targets of pointer argument ap
3 $W := W \cup Pts(ap)$
4 **foreach** *global pointer gp* **do**
5 $Pts(gp)$ represents the points-to targets of global pointer gp
6 $W := W \cup Pts(gp)$
7 **while** $W \neq \emptyset$ **do**
8 $o := SELECT(W)$
9 $Set_{ntl} := Set_{ntl} \cup \{o\}$
10 $Pts(o)$ represents the points-to targets of o
11 **foreach** $o' \in Pts(o)$ **do**
12 **if** $o' \notin Set_{ntl}$ **then**
13 $W := W \cup \{o'\}$

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 is 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

```
1081 for(i=0;i<nprocs-1;i++){ //fork loop
1082  Error=pthread_create(&PThreadTable[i],
                NULL,(void*)(slave),NULL);
1087 }
     ......

1100 for(i=0;i<nprocs-1;i++){ //join loop
1101  Error=pthread_join(PThreadTable[i],NULL);
1106 }
     ......
```

Figure 6: A multi-forked thread example in `ocean_ncp` from the SPLASH2 benchmark suite.

Original *TSAN*'s strategy is to check all variables and statements in a program, expect *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 pairs 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 spawnee 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

Table 1: Benchmarks statistics (under compiler option -O0).

	Pthread API Invocations					Instrumented by *TSAN*			Instrumented by Ours		
	Create	Join	Lock	Trylock	Unlock	Read	Write	Ignored	Read	Write	Ignored
barnes	1	1	12	0	12	2188	1222	395	982	601	2222
fft	1	1	8	0	8	1048	387	88	576	261	686
lu_cb	1	1	6	0	6	1097	408	81	396	199	991
lu_ncb	1	1	6	0	6	840	303	73	392	182	642
ocean_cp	1	1	24	0	24	9531	2301	410	5722	1809	4711
ocean_ncp	1	1	23	0	23	5465	1381	301	2909	1103	3135
radiosity	3	3	38	0	50	5250	1917	269	3006	1500	2930
radix	1	1	13	0	13	777	354	120	329	208	714
raytrace	1	1	13	0	16	6049	2368	359	2865	1057	4854
water_nsquared	1	1	18	0	18	2188	703	181	1313	510	1249
water_spatial	1	1	19	0	19	2437	833	211	1168	440	1873

3.3.1 *Static Analysis*

Table 1 gives the static statistic. For each benchmark, we list its 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.

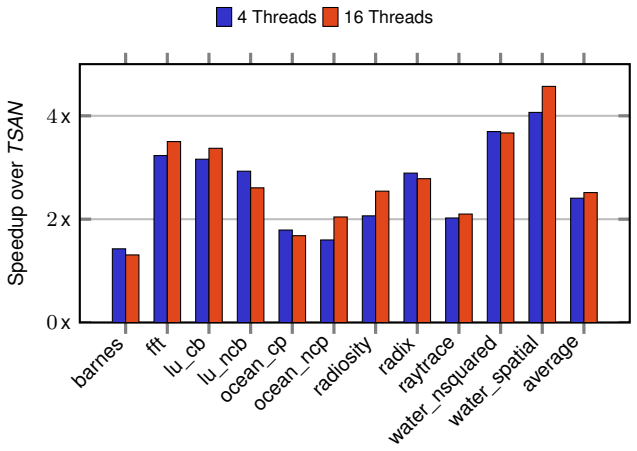


Figure 7: Speedups our static analyzer over original *TSAN* (under compiler option -O0).

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 date 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 focuse 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

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2), Mar. 2006.

[2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP '07*, page 183, 2007.

[3] K. Apinis, H. Seidl, and V. Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *PLDI '13*, pages 377–386, 2013.

[4] R. Barik. Efficient computation of May-Happen-in-Parallel information for concurrent Java programs. In *LCPC'05*, pages 152–169, 2005.

[5] C. Boyapati and M. Rinard. A Parameterized Type System for Race-free Java Programs. In *OOPSLA '01*, pages 56–69, 2001.

[6] G. Bristow, C. Drey, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *ICSE '79*, pages 265–273. IEEE Press, 1979.

[7] D. Callahan. Static Analysis of Low-level Synchronization. In *PADD'88*, pages 100–111, 1892.

[8] C. Chen, W. Huo, L. Li, X. Feng, and K. Xing. Can we make it faster? Efficient may-happen-in-parallel analysis revisited. In *PDCAT'12*, pages 59–64, 2012.

[9] M. Christiaens and K. D. Bosschere. Trade: Data race detection for java. In *ICCS '01*, pages 761–770, 2001.

[10] P. Di, Y. Sui, D. Ye, and J. Xue. Region-Based May-Happen-in-Parallel Analysis for C Programs. In *ICPP' 15*, 2015.

[11] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4*, pages 36–48, 1991.

[12] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: interference-free regions for dynamic data-race detection. In *ACM international conference on object oriented programming systems languages and applications - OOPSLA'12*, volume 47, pages 467–484. ACM, 2012.

[13] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *PLDI '07*, pages 245–255, 2007.

[14] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP '03*, pages 237–252, 2003.

[15] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI '09*, 2009.

[16] C. Flanagan and S. N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP'13*, pages 255–280, 2013.

[17] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298, 2011.

[18] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. *PLDI'14*, pages 337–348, 2014.

[19] M.-Y. Hung, P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J.-K. Lee. Support of probabilistic pointer analysis in the ssa form. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2366–2379, 2012.

[20] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, pages 1–34, 2014.

[21] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL'11*, pages 623–636, 2011.

[22] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07*, pages 226–239, 2007.

[23] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural aliasing. *PLDI '92*, 27(7).

[24] J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *PPoPP '10*, pages 25–36, 2010.

[25] P. Masticola and G. Ryder. Non-concurrency Analysis. In *PPOPP'93*, pages 129–138, 1993.

[26] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06*, volume 41, page 308, 2006.

[27] G. Naumovich. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs Information for Concurrent Java Programs. In *ESEC/FSE-7*, pages 338–354, 1999.

[28] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[29] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[30] D. Pearce, P. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.

[31] F. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135, 2009.

[32] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.

[33] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection. In *PLDI '06*, pages 320–331, 2006.

[34] R. R. Rick Hank, Loreena Lee. Implementing next generation points-to in open64. In *Open64 Developers Forum*, 2010.

[35] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *PLDI '99*, volume 34, pages 77–90. ACM, 1999.

[36] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *PPOPP '01*, 36(7):12–23, 2001.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[38] K. Sen. Race directed random testing of concurrent programs. In *PLDI'08*, volume 43, page 11, 2008.

[39] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[40] W.-L. Shih, Y.-P. You, C.-W. Huang, and J. K. Lee. Compiler optimization for reducing leakage power in multithread bsp programs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(1):9, 2014.

[41] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL'12*, volume 47, page 387, 2012.

[42] Y. Sui, P. Di, and J. Xue. Sparse Flow-Sensitive Pointer Analysis for Multithreaded C Programs. In *CGO '16*, 2016.

[43] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11.

[44] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171.

[45] Y. Sui, S. Ye, J. Xue, and J. Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *SPE '13*, 2013.

[46] R. N. Taylor. A General Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, 1983.

[47] V. Vojdani and V. Vene. Goblint: Path-Sensitive Data Race Analysis. 2009.

[48] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA '01*, pages 70–82, 2001.

[49] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC-FSE '07*, pages 205–214, 2007.

[50] X. Xie and J. Xue. Acculock: Accurate and efficient detection of data races. In *CGO '11*, pages 201–212, 2011.

[51] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. 2014.

[52] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP '05*, pages 221–234, 2005.