SmallRace: Static Race Detection for Dynamic Languages - A Case on Smalltalk

Siwei Cui
Texas A&M University
College Station, Texas
siweicui@tamu.edu

Yifei Gao
Texas A&M University
College Station, Texas
keizoku-pravda@tamu.edu

Rainer Unterguggenberger
Lam Research
rainer.unterguggenberger@lamresearch.com

Wilfried Pichler
Lam Research
wilfried.pichler@lamresearch.com

Sean Livingstone
Texas A&M University
College Station, Texas
seanlivingstone@tamu.edu

Jeff Huang
Texas A&M University
College Station, Texas
jeff@cse.tamu.edu

***Abstract*—Smalltalk, one of the first object-oriented programming languages, has had a tremendous influence on the evolution of computer technology. Due to the simplicity and productivity provided by the language, Smalltalk is still in active use today by many companies with large legacy codebases and with new code written every day.**

A crucial problem in Smalltalk programming is the race condition. Like in any other parallel language, debugging race conditions is inherently challenging, but in Smalltalk, it is even more challenging due to its dynamic nature. Being a purely dynamically-typed language, Smalltalk allows assigning any object to any variable without type restrictions, and allows forking new threads to execute arbitrary anonymous code blocks passed as objects. In Smalltalk, race conditions can be introduced easily, but are difficult to prevent at run time.

We present SmallRace, a novel static race detection framework designed for multithreaded dynamic languages, with a focus on Smalltalk. A key component of SmallRace is SmallIR, a subset of LLVM IR, in which all variables are declared with the same type—a generic pointer `i8*`. This allows SmallRace to design an effective interprocedural thread-sensitive pointer analysis to infer the concrete types of dynamic variables. SmallRace automatically translates Smalltalk source code into SmallIR, supports most of the modern Smalltalk syntax in Visual Works, and generates actionable race reports with detailed debugging information. Importantly, SmallRace has been used to analyze a production codebase in a large company with over a million lines of code, and it has found tens of complex race conditions in the production code.

I. Introduction

Race conditions are among the worst types of bugs to encounter in software programs. Due to their non-deterministic nature, race conditions often lead to unpredictable behaviors and it is notoriously difficult to reproduce and debug a race in production environments. The race detection problem has inspired a significant amount of research efforts in different languages and platforms, such as Java, JavaScript, Android, C/C++, and OpenMP [1]–[12]. This work focuses on race detection for Smalltalk, an object-oriented dynamic language.

Developed in the 1970s, Smalltalk was one of the most popular programming languages, and is still actively used today in the semiconductor, manufacturing, financial industries, and many others [13]. These industries all have core business applications written in Smalltalk with large legacy codebases that must be maintained and developed to meet business needs.

Smalltalk programs are prone to race conditions but are extremely challenging to debug due to the high flexibility of the language and the complex runtime. Everything in Smalltalk is an object, a chunk of code that manages a specific piece of data. Other objects act upon that data by passing messages to its object. Smalltalk allows assigning any object to any variable without type restrictions. The Smalltalk virtual machine performs the type check only at runtime. It considers the type of an object as valid if it implements the message sent to it, otherwise, it throws an exception. Smalltalk incorporates green thread, and it emulates multi-threaded environments without relying on the native OS. Moreover, an object can re-compile itself by executing new code every time it is updated, and can work with other objects as a program by exchanging messages. While this approach makes code reusable and easy to test, it also makes it easy to introduce errors at runtime. This is especially true when race conditions are introduced by dynamic code that forks new threads at bad timing, with an incorrect priority, or without proper lock protection on shared variable accesses.

In this paper, we present SmallRace, an end-to-end static analysis framework for detecting race conditions in multithreaded dynamic languages, with a focus on Smalltalk. SmallRace addresses many technical and practical challenges that are essential in handling dynamic language features such as dynamic types, closures, and variable scopes. SmallRace is based on LLVM [14], a popular compiler toolchain that contains many modular and reusable libraries and tools for building compilers. LLVM is widely adopted for

© © 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

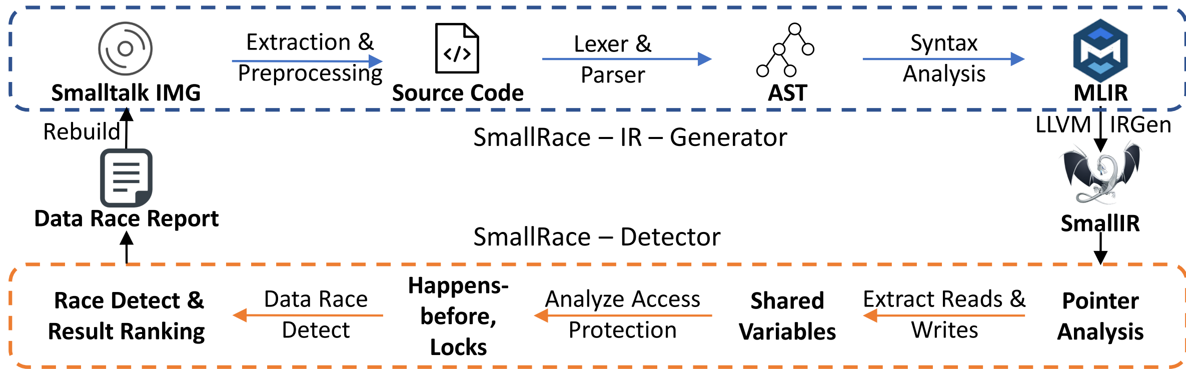

Fig. 1. An Overview of the SmallRace Framework.

program optimization [15]–[17], code generation [18]–[20], and vulnerability detection [21], [22]. Central to LLVM is the LLVM IR, a low-level intermediate representation that serves as the common interface to work with other LLVM components.

An overview of SmallRace is depicted in Fig. 1. SmallRace works by first compiling a Smalltalk image containing all the source code into Smalltalk LLVM IR (SmallIR), which follows the same format as LLVM IR (SmallRace-IR-Generator phase). A main technical challenge is that standard LLVM IR must have a type, but Smalltalk variables do not have a static type. To address this problem, our novel `SmallIR` translates all variables in the IR into generic pointer types `i8*`. It exclusively uses pointers in IR generation to preserve all Smalltalk semantics, allowing faster pointer analysis and type inference.

With `SmallIR`, we can offload type inference into the analysis (SmallRace-Detector phase). We propose demand-driven type inferences and reduce the overall workload by only considering variables that need type assignments. This phase has multiple technically interesting components, which are built on top of an interprocedural thread-sensitive data flow analysis on pointers. We generate a ranked data race report by analyzing memory accesses on shared variables that do not have a happens-before relation and are not protected by the same locks or synchronizations. With static analysis, the runtime environment is not required for SmallRace. Compared with dynamic analysis, SmallRace achieves high code coverage, and explores code paths independent of the number of threads and input.

The main contributions of this work are:

- To our knowledge, we are the first to build an end-to-end static race detector for Smalltalk, a dynamically-typed language with challenging features such as dynamic types, closures, and code blocks.
- We propose a novel approach to generate LLVM IR for dynamically-typed languages, and introduce SmallIR designed for Smalltalk. The key novelty of SmallIR is that it exclusively uses pointers in our IR generation phase to preserve all Smalltalk semantics.
- SmallRace is fully automated and scales to large complex codebases containing millions of lines of code.
- SmallRace has been evaluated extensively on benchmarks from small tests to a large-scale production code base, all provided by our industry partner. It has revealed 37 real race conditions in the production code confirmed by the developers.
- We have released a version of SmallRace as open source and publicly available on GitHub[1].

II. Background and Motivation

In this section, we review important language features of Smalltalk, and illustrate the technical challenges of Smalltalk race conditions with examples.

A. Smalltalk Language

Smalltalk is a dynamically-typed reflective programming language with a unique feature: everything in Smalltalk is an object. There are no constructors, static methods, type declarations (dynamic), interfaces, or package/private/protected modifiers in Smalltalk. All methods are public virtual, and all attributes are protected and private to the object while accessible from subclasses. There is only single inheritance between classes.

Smalltalk provides a clean and elegant syntax that can be extended easily [13]. There are multiple standards and dialects with variations in Smalltalk grammar, such as Visual Works [23], GNU Smalltalk [24], Pharo [25], and Squeak [26]. In this work, we focus on Visual Works. Visual Works consists of two key components: (1) a virtual image contains all objects in the system; (2) a virtual machine consists of hardware and software to give dynamics to objects in an image. This design ensures the portability of the virtual image.

There are three key concepts in Smalltalk related to race conditions: threads, code blocks, and semaphores.

- *Threads*. Smalltalk allows multiple threads to execute (pseudo) concurrently. Similar to Java threads and `await` in JavaScript, all threads in Smalltalk run within the same address space (a single operating system process) and can

[1]https://github.com/parasol-aser/smallrace-open-source

communicate via shared objects [27]. Thread scheduling is priority-driven, with preemptive time slicing within each priority level. Scheduling is implemented fully in Smalltalk.

- *Code Blocks*. In Smalltalk, a closure is a code block, a self-contained code snippet enclosed in the square brackets for computation. The statements in a code block are not immediately evaluated, and can be activated by sending the `#value` message to the code block object. Code blocks are responsible for thread creation. In Smalltalk, a thread is created when a code snippet is wrapped around a code block and a fork message is sent to that block. (i.e., call `fork` or `forkAt: p`, where `p` holds the value of priority).

```
1  | test |
2  test := SharedState new.
3  test initialize.
4  test start.
5  (Delay forSeconds: 10) wait.
6  test stop.
7
8  <class>
9  <name>SharedState</name>
10 <super>Core.Object</super>
11 <class-vars></class-vars>
12 <inst-vars>max shared thread1 thread2 </inst-vars>
13 </class>
14 <body package="SharedState"
       selector="initialize">initialize
15   max := 1000000.
16   shared := Array new: max.
17   ^self
18 </body>
19 <body package="SharedState" selector="start">start
20     self startThread1.
21     self startThread2.
22 </body>
23 <body package="SharedState"
       selector="startThread1">startThread1
24   | race |
25   thread1 := [
26   [race := 0.
27   1 to: max do: [:index|shared at: index put:1].
28   shared do:
29       [:each|each=1 ifFalse: [race := race+1]].] ]
       forkAt: 30
30 </body>
31 <body package="SharedState"
       selector="startThread2">startThread2
32   | race |
33   thread2 := [
34   [race := 0.
35   1 to: max do: [:index | shared at: index put: 2].
36   shared do:
37       [:each|each=2 ifFalse: [race := race + 1]].]
       ] forkAt: 31
38 </body>
```

Listing 1. A simplified race condition in Smalltalk

- *Semaphores*. In Smalltalk, semaphores are used to implement lock mutual exclusion and other synchronization mechanisms. For example, to create a critical section, a semaphore object can be instantiated by `sem := Semaphore forMutualExclusion` and used by sending a `critical:` message: `sem critical: codeBlock`.

In Smalltalk, a higher-priority process will preempt all lower-priority processes. (1) A new process will start evaluating if all other processes with a priority equal or higher to the new process are waiting on a preemption point (`Semaphore wait`); (2) A new process will preempt all other processes with a lower priority.

Smalltalk implements a green thread system, and threads are scheduled by the Smalltalk virtual machine instead of the OS. The Smalltalk virtual machine executes one instruction at a time, ensuring a sequentially consistent view of all memory accesses. However, due to the priority-driven nature, read-modify-write sequences can race. Any read-modify-write on the instance variables of shared objects is a potential race condition with different order of execution on reads and writes.

B. Smalltalk Race Conditions

We use an example in Lst. 1 to illustrate race conditions. Lines 1-6 show a test script, i.e., the main entry point. It declares a local variable `test` (Line 1), creates a new `SharedState` object and assigns it to the local variable `test`, and sends the messages `initialize` and `start` to the object (similar to calling `test.initialize()` and `test.start()` in Java). It waits for 10 seconds and then sends the `stop` message to `test`.

Lines 8-13 define the `SharedState` class and variables. `SharedState` has four instance variables `max`, `shared`, `thread1` and `thread2`. It does not have any class variable and its superclass is `Core.Object`, the root of the class hierarchy.

Lines 14-38 define the Smalltalk methods of `SharedState`. The `initialize` method defined at Line 14 sets the value of `max`, and initializes a shared variable `shared` to be an array of size `max`. The method calls `startThread1` and `startThread2` in the `start` method to create new threads by calling `forkAt: priority`. In each thread, the `shared` variable is updated in a loop by writing a different value into the shared array, which is then used for counting the number of races. The race condition is caused by the two threads trying to write to the same `shared` array at the same index, while not being protected by a lock.

Detecting the race condition in Smalltalk has several challenges compared to a statically typed language.

- First, all variables, including local variables (`test`, `race`) and class instance variables (`max`, `shared`, `thread1`, and `thread2`), are declared without types. The types are important to recognize the method called by a message. For instance, `test start` calls the method `start` (Line 19). Without knowing that the type of `test` is `SharedState` statically, it is difficult to determine the call target, since a method `start` may be defined in many other classes.
- Second, threads are created by sending a `forkAt:` message to a code block, which defines the thread body. However, the code block can be defined elsewhere and passed to the fork site.

- Third, the variables used in a code block can be defined in outer scopes. Code blocks can be nested, and inner blocks can refer to any statically visible outer scope's variables. For example, both thread1 and thread2 use a local variable race defined outside the code block and use the class instance variable shared of the test object. It is difficult to find which variable is thread local and which is shared by multiple threads.

```
"workspace script"
|x y|
x := 0.
y := 0.
[
  |t1 t2|
  t1 := x.
  t2 := t1.
  [
    y := t1 + t2.
  ] forkAt: 20.
] forkAt: 10.
x := y.
```

Listing 2. An illustration of races in Smalltalk closures

Technical Challenges on Closures. In Lst. 2, there are three threads: the main thread (T0), the thread (T1) created by T0 (Line 12) executing the code block between Lines 5-12, and the thread (T2) created by T1 (Line 11) executing the code block between Lines 9-11. The local variables x and y defined in T0 are used in T1 and T2 respectively, and the local variables t1 and t2 defined in T1 are both used in T2. Although x and y are defined as local variables in T0, there exist race conditions on them because x and y are also implicitly passed to the code blocks and used in other threads. The races on x are between Line 13 (written by T0) and Line 7 (read by T1). The races on y are between Line 13 (read by T0) and Line 10 (written by T2). However, there does not exist any race on the local variables t1 and t2, because their accesses by T1 and T2 are all happens-before ordered.

III. SmallRace in a Nutshell

In this section, we present an overview of SmallRace (Fig. 1) and illustrate how it works on the motivating examples in Lsts. 1-2.

SmallRace has two major components: (1) SmallRace-IR-Generator generates SmallIR from source code; (2) SmallRace-Detector generates a data race report based on the SmallIR. In IR-Generator, the input is a collection of source code file out files (.st files) containing program libraries, and optionally together with a workspace file (.ws file) for the entry point. We define the language grammar for Visual Work Smalltalk in ANTLR [28], and parse both the file out source code and workspace file. An abstract syntax tree (AST) is generated, and then translated into our SmallIR instruction by instruction.

We design the SmallIR, a dialect of LLVM IR, to preserve all Smalltalk semantics while parsing. Type inference for each variable is required to build LLVM IR, as standard LLVM IR requires a type. For example, to emit a load or store instruction, the type specified must be a first-class type of known size. For a dynamically-typed language such as Smalltalk, inferring type statically is challenging and computationally expensive. SmallIR models types of variables exclusively as pointers. It represents types at the IR level as *generic pointers*, and models Smalltalk semantics with *call* instructions. SmallIR can preserve all Smalltalk semantics into IR, and allows for offloading the type inference as demand-driven in the analysis phase. As we do not need typing for every variable to detect race conditions, the novel SmallIR design improves the analysis efficiency with demand-driven type inference.

SmallRace-Detector analyzes the SmallIR, and performs a series of analyses to detect race conditions: (1) identifying threads, (2) finding conflicting accesses from multiple threads, (3) lockset [29], and (4) happens-before. The race detection algorithm is illustrated in Alg. 3. SmallRace-Detector incorporates pointer analysis to traverse instructions of each thread, extracts the memory accesses (variable reads and writes), and synchronizations (locks and semaphores). It analyzes all the knowledge we gathered from SmallIR, checks happens-before and lockset conditions, and generates a race report.

Lst. 3 shows the SmallIR generated for the code in Lst. 2. To detect races based on the IR, we first perform an interprocedural thread-sensitive pointer analysis to determine the type of each variable, which is marked by i8* in the IR. The type inference is important for determining the call target of any call instruction st.call.*. For example, the source code "test start" on Line 4 of Lst. 1 will generate a call instruction call @st.call.start(i8* %1), in which %1 refers to the variable test. The pointer analysis will identify that the type of %1 is SharedState and, therefore, we can determine the call target is start$SharedState. In addition to type inference, pointer analysis is used to determine the object of every memory access. For example, in st.model.opaqueAssign(i8* %dst, i8* %src), if pointer analysis infers that %src may point to a certain object o1, and %dst may point to a certain object o2, we can extract a read access to o1 and a write access to o2.

Being thread-sensitive, the pointer analysis uses the thread creation site (e.g. the instruction st.forkAt:) to represent the context of each pointer variable. This leads to a highly precise and scalable pointer analysis. We will elaborate on our pointer analysis algorithm in more detail in Section V-A.

After pointer analysis, we launch our static tracing from an entry point, traverse instructions, track thread creations and data flows, and extract race-relevant information including memory accesses (initialization, read, and write) and synchronizations. To detect a race, we check if there are two or more threads accessing the same variable, at least one is a write, and two accesses are not protected by a common lock (lockset) and without a Happens-Before (HB) relationship. We will report a potential race for those cases.

We illustrate our analysis approach for the race condition (Lst. 1) in Fig. 2. There are user-defined methods start,

```
1  define i8* @"st.anonfun.1*main"(i8* %0, i8* %1,
      i8* %2) !dbg !3 {
2    %4 = call i8* @st.model.newTemp(@t1), !dbg !7
3    %5 = call i8* @st.model.newTemp(@t2), !dbg !9
4    call void @st.model.opaqueAssign(i8* %4, i8*
       %1), !dbg !10
5    call void @st.model.opaqueAssign(i8* %5, i8*
       %4), !dbg !11
6    %6 = call i8*
       @st.model.blockParam4(@"st.anonfun.2*
7    st.anonfun.1*main.1", i8* %4, i8* %5, i8* %2),
       !dbg !12
8    %7 = call i8* @"st.forkAt:"(i8* %6, 20), !dbg
       !12
9    ret i8* %0, !dbg !13
10 }
11
12 define i8* @"st.anonfun.2*st.anonfun.1*main"(i8*
      %0, i8* %1, i8* %2, i8* %3) !dbg !14 {
13   %5 = call i8*
       @st.model.binaryop(@"global_op_+", i64 0,
       i64 0), i8* %1, i8* %2), !dbg !15
14   call void @st.model.opaqueAssign(i8* %3, i8*
       %5), !dbg !17
15   ret i8* %0, !dbg !18
16 }
17
18 define i64 @main() !dbg !19 {
19   %1 = call i8* @st.model.newTemp(@x), !dbg !20
20   %2 = call i8* @st.model.newTemp(@y), !dbg !22
21   call void @st.model.opaqueAssign(i8* %1, 0),
       !dbg !23
22   call void @st.model.opaqueAssign(i8* %2, 0),
       !dbg !24
23   %3 = call i8*
       @st.model.blockParam3(@"st.anonfun.1*main.2",
       i8* %1, i8* %2), !dbg !25
24   %4 = call i8* @"st.forkAt:"(i8* %3, 10), !dbg
       !25
25   call void @st.model.opaqueAssign(i8* %1, i8*
       %2), !dbg !26
26   ret i64 0, !dbg !27
27 }
```

Listing 3. Partial SmallIR generated for Lst.2

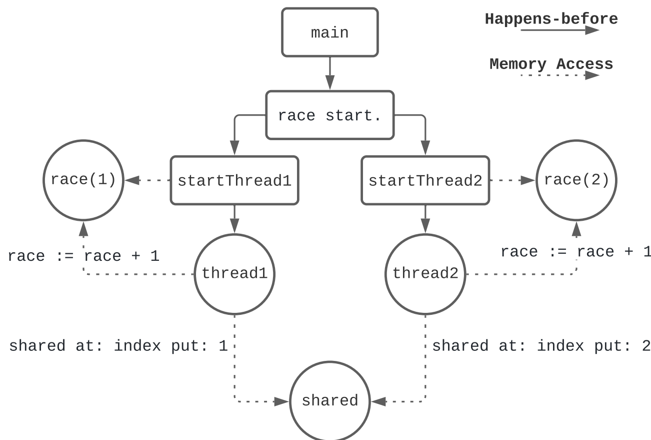

Fig. 2. Visualization for the race in Lst. 1.

startThread1/2, and multiple anonymous methods representing code blocks. To detect races, threads are identified in the IR, created by fork or forkAt:. We find two threads created at startThread1 and startThread2, respectively. The next step is to check memory accesses. The thread1 spawn from startThread1 modifies the instance variable shared, while thread2 also accesses the same variable and writes a different value. Since there is no lock protection, we will report a race.

IV. SmallRace-IR-Generator

A. Lexing, Parsing, and AST Design

The Smalltalk image contains all the source code in XML format using the file out option in Visual Works. We define our grammar in ANTLR [28] to parse the Smalltalk source code, and produce a parse tree for each function. Although ANTLR offers a standard Smalltalk grammar [30] from the Redline Smalltalk Project, differences in syntax between Visual Works and Redline Smalltalk result in many parsing errors. To address this, we developed a dedicated grammar for Visual Works. Fixing the grammar issues is a labor-intensive effort. Our Visual Works Smalltalk grammar can successfully parse all the 190,467 functions in the production code base with only 24 errors, for a success rate of over 99%. We translate the parse trees into customized AST nodes preserving all Smalltalk class metadata, and generate SmallIR based on our AST with debugging symbols to facilitate the data race detection.

Our AST design follows ideas from Clang and Flang. We use ParseTreeVisitor in ANTLR to traverse the parse tree and generate the AST. In SmallRace, all AST nodes are designed to be subclasses of the BaseAST class, capturing general information (source code location, AST level, and AST types). Extended from BaseAST, each AST class captures a specific parsing unit, and is later responsible for generating SmallIR with the carried information. For example, BlockExprAST represents the expression class for code blocks, e.g., [Transcript show: race]. The class fields include the parameters passed to the code block and local variables defined in the code block, both of which are modeled as VarDeclExprAST. BlockExprAST also contains a parent scope, which is either a FunctionAST or another BlockExprAST. With this design philosophy, our AST can be easily extended to support increasing features and semantics in Smalltalk.

Our AST is designed to capture most semantics that is critical for race detection and other analyses for Smalltalk. For example, when the AST visitor encounters a temporary variable, we do not have the type declaration as Smalltalk is dynamically typed. We create a VarDeclExprAST without any type information, and postpone the type inference to the analysis phase. Function calls will produce a FunctionCallAST, modeling a message being sent to an object. In Smalltalk. A unary message "new" sent to an object would call the constructor function of objects if not overloaded. At the AST level, initializing an object is represented similarly to other unary messages, and a FunctionCallAST is emitted. For example *"SharedState new"* at line 2 (Lst. 1) will generate a FunctionCallAST with callee

as "new" targeting "SharedState". Different messages will be translated into subclasses of FunctionCallASTs, such as `KeywordFunctionCallAST` of `Delay forSeconds: 10` (Line 5 of Lst. 1).

B. SmallIR Generation

We construct `SmallIR`, a Smalltalk intermediate representation compatible with LLVM IR, using Multi-Level Intermediate Representation (MLIR) [31]. MLIR has a standard dialect and many other dialects to support different IRs, giving us more flexibility in IR generation. To generate SmallIR, we define a new Smalltalk dialect, and lower our AST to first the MLIR and then the `SmallIR` in LLVM IR forms. We lower most Smalltalk code into function calls, including creating new variables, objects, and assignments between variables.

TABLE I
SMALLIR DEFINITION

Smalltalk code	SmallIR Model Function
|a|	st.model.newTemp(i8* a)
Collection new.	st.model.newObject(i8* @Collection)
a + b	st.model.binaryop(i8* op_+,i8* a, i8* b)
a var	st.model.instVar(i8* @var)
[:a|...]	st.model.blockParam(i8* a, ...)
a := b	st.model.opaqueAssign(i8* a, i8* b)

In SmallIR, almost all the IR instructions (except function return `ret`) are LLVM `call` instructions, and every call instruction will call a function name that starts with either `st.model.*` (denoting special model functions in SmallRace), `st.anonfun.*` (denoting code blocks), or a normal function defined in the Smalltalk source code (`st.call.*`). All variables are represented by a universal pointer type `i8*`.

The list of model functions in SmallIR is shown in Table I. Each captures a certain semantic of Smalltalk. We omitted the `st.model.` for simplification.

- `newTemp(i8* @x)` creates a local variable `x`.
- `newObject(i8* @x)` creates a object named `x`.
- `binaryop(i8* %st.global_op, i8* %1, i8* %2)` invokes a binary operation (e.g. +, −, *, /) on two variables `%1` and `%2` and returns the value.
- `instVar(i8* @x)` creates a reference to a class instance variable named `x`.
- `blockParam(i8* %st.anonfunc, ...)` links the parameters of a code block from its outer scope.
- `opaqueAssign(i8* %dst, i8* %src)` reads a variable `%src`, assigns it to another variable `%dst`.

Code blocks are represented by anonymous functions. For example, the call instruction `st.forkAt:(i8* %st.anonfun.main.1, i8* @"10")` creates a thread with priority `10`. The new thread executes the code block represented by function `st.anonfun.main.1`.

With SmallIR, SmallRace has the ability to deal with Smalltalk language features. Our novel IR represents types as *generic pointers*, and models Smalltalk semantics with *call* instructions. We exclusively use pointers in SmallIR generation to preserve all the semantics and perform on-demand type inference to improve efficiency.

Handling Nested Code Blocks. Code blocks are one of the key components in Smalltalk semantics related to race detection. Smalltalk supports nested blocks. The variables used inside a code block can be defined in an outer scope, either a function or another code block. As we lower code blocks to anonymous functions, the pointers to variables defined in outer scopes are added to the function signatures. Variables that do not appear in block parameters and are not defined inside the current block are modeled as parameters of anonymous functions. We only add the variables that are used inside a code block.

For functions that are called with a block as an argument, we pack the block along with all outer scope variables by generating a `st.model.blockParam` instruction and map the corresponding arguments in the outer scope. For example, when coming across functions creating new threads, we create a context `%c` for the anonymous function `func` wrapped with actual parameters $v_0 \cdots v_{n-1}$, and feed into the fork function. We emit two SmallIR instructions: $\%c = blockParam(func, l_0, \cdots, l_{n-1}); st.forkAt(\%c, priority);$ The linking between formal parameters and actual parameters is conducted in the analysis phase.

V. SMALLRACE-DETECTOR

To detect race conditions, SmallRace-Detector has three whole program passes. The first two passes perform an inter-procedural thread-sensitive pointer analysis to infer types for the pointer variables. The third pass extracts static thread traces, finds shared variables, and performs race checking using a hybrid happens-before and lockset analysis. We examine each pair of read/write on a shared variable. If one of the memory accesses is a write, and two memory accesses are executed on different threads without proper protections (e.g., are protected by a common lock, or have a happens-before order), we will report a race.

To build stack traces for a piece of code, we build two graphs, i.e., a `call graph` and a `constraint graph`. The constraint graph captures the memory correlation between two LLVM values over five types of constraints: load, store, copy, address-of, and offset. We run pointer analysis on the constraint graph to derive the `pointsTo set` for each memory location. We build the stack trace statically by traversing the call graph, identifying each API call, and conducting interprocedural analysis.

A. Thread-sensitive Pointer Analysis

Alg. 1 describes our pointer analysis. We traverse the SmallIR twice. First, we iterate through all instructions from the entry point, add forked functions to the call graph, and add new objects or assignment constraints to the constraint graph. However, most function calls depend on the type of caller to resolve the call target. To address this problem, we store these function calls as indirect function calls in $funPtrs$, and compute the correct call target with the `pointsTo set`

in the second pass. The `pointsTo set` contains all the potential types of an object. We resolve the function call to the correct call target, and expand the newly discovered function body for recursive iteration.

```
Algorithm 1: Pointer Analysis in SmallRace
   Input: SmallIR for all functions module, a list of function pointers
          funPtrs
   Output: Call Graph callgraph; Constraint Graph consgraph
1  Function solve(module, entry):
2      foreach Instruction inst ∈ entry do
3          switch typeof(inst) do
4              case fork(callee) do
5                  callgraph.addCallEdge(entry, inst.callee)
6                  solve(module, inst.callee)
7              case newObject(obj) do
8                  consgraph.addConsEdge(inst.obj, inst, addr_of)
9              case assign(lhs, rhs) do
10                 consgraph.addConsEdge(inst.rhs, inst.lhs, copy)
11             case call(callee, args) do
12                 if isDirectCall(inst) then
13                     callgraph.addCallEdge(entry, callee)
14                     solve(module, callee)
15                 else
16                     funPtrs.append(inst, args(0))
17 Function updateFunPtrs():
18     foreach Instruction inst, Pointer p ∈ funPtrs do
19         prefix = inst.getName()
20         foreach suffix ∈ p.pointsTo() do
21             func = module.getFunc(prefix+suffix)
22             callgraph.addCallEdge(inst.getCaller(), func)
23             solve(func, module.getFunc(inst))
24 solve(module, module.getFunc("main"))
25 updateFunPtrs()
```

We adopt Andersen's pointer analysis [32] to resolve the pointsTo set of variables iteratively. Pointer analysis is a static analysis technique to derive the mapping between a pointer and a variable or heap allocation. In `SmallIR`, all variables are defined using a pointer type. We heavily use the `address-of` and `copy` constraints in our constraint graph to facilitate pointer analysis and infer types on demand to allow faster pointer analysis and type inference. The call graph is built up in the same process of constructing the constraint graph for efficiency.

Our pointer analysis is thread-sensitive. Compared to traditional k-CFA [33] that uses k method call sites as the context, we use k thread creation sites (i.e., creating a new thread by `st.fork*`) as the context. We demonstrate the feasibility of using pointer reasoning to statically improve the accuracy of type inference and polymorphism.

Besides type inference, pointer analysis is also good for identifying shared objects. For example, variables with the same name but defined under different scopes are eliminated from race checking. For heap allocation APIs, we capture the function call to `st.model.instVar` and `st.model.newObject`. We add to our constraints graph in three situations. (1) **newObject b = obj()**: we add an `address-of` constraint from the pointer to the actual object being allocated. (2) **OpaqueAssign a=b**: we add a *copy* edge in our constrain graph from a to b. (3) **Anonymous Functions**: for each function call to *st.anonfun*, we add a *copy* edge from the call site of the anonymous function to the actual function declaration.

B. Program and Thread Traces

A program trace is a sequence of important events related to race conditions, including memory read/write, thread fork, lock/unlock, and wait/signal. We build program traces by traversing the call graph, and capturing key events related to each thread, as shown in Alg. 2.

```
Algorithm 2: Event Trace Building
   Input: SmallIR module;
   Output: Thread Trace threads
1  Function genFuncSummary(node, module):
2      foreach Instruction inst ∈ node do
3          switch typeof(inst) do
4              case assign(lhs, rhs) do
5                  summary.add(load, rhs)
6                  summary.add(store, lhs)
7              case call(callee(arg)) do
8                  if callee.isRead() then
9                      summary.add(load, arg)
10                 if callee.isWrite() then
11                     summary.add(store, arg)
12             case fork(callee) do
13                 summary.add(fork, callee)
14     return summary
15 Function traverse(node, thread):
16     summary = genFuncSummary(node, module)
17     foreach Instruction inst ∈ summary do
18         switch typeof(inst) do
19             case load(obj) do
20                 trace.add(read, obj)
21             case store(obj) do
22                 trace.add(write, obj)
23             case call(callee) or fork(callee) do
24                 traverse(callee, fork?threads.new():thread)
25 threads.add(thread0)
26 traverse(module.getFunc("main"), thread0)
```

Starting from the entry function, we process each SmallIR instruction and use the semantics to produce events. For example, for a function call `st.anonfun*`, we construct a Smalltalk Fork IR, and a ForkEvent is generated. A new ThreadTrace will be built for each fork event. For function call `st.model.opaqueAssign(a,b)`, we generate two events, i.e., read for pointer *b* and write to pointer *a*.

Fig. 3 shows the program trace constructed for Lst. 3. The main thread (`Thread0`) has 5 events, including forking a new thread at the third block. The thread created on Line 24 (`Thread1`) forks a new thread on Line 8 (`Thread2`). There are two data races in this listing. The first race involves a read of local variable `local_y` in `Thread0` (T0.e4) and a write to `local_y` at `Thread2` (T2.e3), without proper protections. The second race is related to a write to `local_x` in `Thread0` (T0.e5) and a read of `local_x` in `Thread1` (T1.e1).

C. Race Detection

The race detection report is generated with Alg. 3. We first analyze shared memory accesses by processing read and write

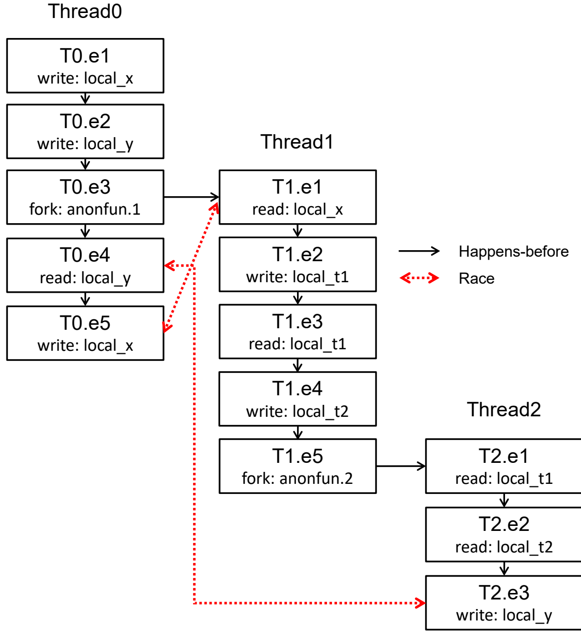

Fig. 3. Generated program trace for Lst. 3.

Algorithm 3: Race Detection in SmallRace

Input: Write Events $writes$, All Read or Write Events $others$
Output: Race Report $report$

```
1  Function checkRace(write, other):
2      if isLocalVar(write) and isLocalVar(other) then
3          return false
4      if happensBefore(write, other) or sharesLock(write, other) then
5          return false
6      return true
7  foreach write ∈ writes do
8      foreach other ∈ others do
9          if checkRace(write, other) then
10             report.collect(write, other)
```

events of each thread and organizing them based on the pointsTo set of each shared memory access. As two memory accesses could be protected by a common lock or have a Happens-Before (HB) order, we conduct a series of analyses to filter out false positives. Specifically, we handle the HB relationship, lock set algorithm, and local variable filtering. We construct a `happens-before graph` from the events currently stored in the program. Successfully building the HB graph depends on the heuristic that each event has an increasing ID per thread. Edges are added between program-ordered events and **sync events** (fork/wait/notify).

Fig. 3 illustrates our happens-before analysis. At `Thread1`, the event `T1.e5` is a fork event spawning `Thread2`. We add one edge from `T1.e5` to the first event `T2.e1` in `Thread2` in the HB graph. We define the sync events as the events with an HB edge on them. Inside the same thread, we model the HB relationship by the event ID instead of explicitly creating an HB edge. To check if an event `Tx.ey` (Thread x, event y) can reach an event `Ta.eb`, we find the closest sync event after `Tx.ey` and find the closest sync event before `Ta.eb`. We conclude that there is an HB order if the sync event after `Tx.ey` can reach the sync event before `Ta.eb` by traversing the HB graph on sync event edges.

To handle synchronization operations, we employ an event-driven `lockset algorithm` instead of adding an HB edge. SmallRace will not report a race if the shared memory is protected by a shared lock or other synchronization techniques.

D. Result Ranking

We incorporate several heuristics to rank the data race report, and assign a score to each data race as a confidence level. Our heuristics are extracted from observations on the production code base and developers' feedback. We assign a higher score to the two memory accesses if both of their stack traces contain a thread creation (i.e., `st.fork*`) or contain a special signature indicating that one function can be called remotely in parallel. We will also increase the score if there is a critical section protecting one memory access, while the other memory access of the same variable is not protected. This is often the case where a programmer forgot to add protection to a variable accessed concurrently. Heuristics in function names are also considered and can be configured in our framework. Function names to initialize the system, such as *startup*, and *setup*, or to terminate the system, such as *clearall*, *shutdown*, *terminate*, and *deactivate*, are being called mostly once per lifetime in our code base. We assign a lower score to a reported race where the stack traces contain those function keywords.

E. Extensibility and Entry Points

SmallRace is highly configurable with a number of command-line arguments as well as a configuration file. It can potentially be extended to be applied in other programming languages with similar characteristics to Smalltalk and creates a compiler infrastructure that facilities other researchers. With SmallIR, SmallRace can be extended to detect other types of errors for other dynamically typed languages, such as Python and JavaScript. With the same abstract machine in Java threads and JavaScript's await operator, the techniques in SmallRace are broadly applicable to many other languages using green threads or native threads.

While adapting SmallRace to a new language is kept straightforward, certain adjustments are necessary. Modifications include translating the language into LLVM-like IR (frontend) and modeling the semantics of the new language (analysis backend). Generally, the effort required for these adaptations is significantly less than the effort to create a new race detector from scratch.

With numerous interfaces in Smalltalk libraries that can be called by multiple threads, we model those APIs in our configuration. There are two main types of APIs specific to Smalltalk. First, we model the read and write functions provided by core libraries. Second, application-specific classes are supposed to be thread-safe. For example, in our production code base, any "ClassName" used in "`CTRemote export #ClassName`" is designed to run in parallel. Therefore, all

TABLE II
BENCHMARK STATISTICS

Image	LoC	#Classes	#Functions	#SO[2]	#Locks	#Threads	#EP[2]
Test 1	85	1	7	5	0	5	1
Test 2	134	2	16	5	0	2	1
Test 3	86	1	8	9	0	3	1
MemoryMonitor	940	1	131	57	1	136	126
AlarmMonitor	15,387	62	2,071	49	1	56	55
Image 1	266,242	3,233	58,526	4,106	22	6,087	5,945
Image 2	109,309	1,922	25,980	1,037	8	1,865	1,741
Image 3	41,171	469	8,143	1,187	6	1,813	1,757
Image 4	45,626	813	12,150	35	0	71	70
Image 5	35,400	246	5,074	395	7	483	437
Image 6	40,551	265	4,336	20	0	28	26
Image 7	591,219	3,237	74,025	6,332	25	11,381	11,184

[2]SO stands for shared objects, and EP stands for entry points.

functions of these classes can be considered as entry points, which can be potentially invoked by threads in parallel.

TABLE III
TIME EFFICIENCY EVALUATION RESULTS

Image	IR-Gen(s)	PTA(s)	Trace(s)	Detect(s)
Test 1	2.48	0.42	0.01	0.01
Test 2	2.80	0.44	0.01	0.01
Test 3	1.20	0.43	0.01	0.01
MemoryMonitor	22.99	1.32	0.07	0.17
AlarmMonitor	319.64	3.62	0.26	0.50
Image 1	22,309.39	15,322.93	14,469.46	20,711.06
Image 2	3,853.27	757.89	182.48	348.09
Image 3	1,367.14	197.10	41.94	80.42
Image 4	1,255.91	8.59	1.12	1.73
Image 5	609.35	30.44	6.31	12.05
Image 6	291.59	2.14	0.14	0.24
Image 7[3]	41,982.00	46,155.71	9,936.85	48,720.08

[3]Image 7 statistics are collected by summing chunks splits.

VI. EVALUATION

We evaluated SmallRace extensively on an AWS instance (`m5 2xlarge`) with a range of benchmarks provided by our industry partners that are executed concurrently for various applications, including three small Smalltalk tests extracted from real-world applications, two medium-sized Smalltalk applications (MemoryMonitor and AlarmMonitor), and a large-scale production code. All benchmarks and their statistics are reported in Table II.

The three small tests illustrate three common patterns of race conditions that had happened in the production code before. Test 1 contains a data race caused by two threads accessing the same instance variable without being protected by the critical section. Test 2 uses a queue to pass through a shared object and execute the code block in a different thread. Test 3 is presented in Lst 1. The large-scale production code contains 22 files, and we generate our test workspace file with all functions with `CTRemote` signature as entry points.

A. *Experimental Results*

We collect the time consumption of the SmallRace-IR-Generator (IR-Gen), time for pointer analysis (PTA), building thread trace (Trace), data race detection (Detect) in Table III,

TABLE IV
EVALUATION RESULTS ON NUMBER OF RACE.

Image	#Total	#Real	#Mild	#FP
Test 1	9	9	0	0
Test 2	3	3	0	0
Test 3	1	1	0	0
MemoryMonitor	10	8	2	0
AlarmMonitor	5	4	1	0
Image 1	63	4/10	2/10	4/10
Image 2	56	2/10	1/10	7/10
Image 3	16	2/10	2/10	6/10
Image 4	100	7/10	1/10	2/10
Image 5	59	3/10	6/10	1/10
Image 6	23	4/10	1/10	5/10
Image 7	100	3/10	4/10	3/10

and the reported races in Table IV. SmallRace took around 64 hours to finish analyzing all the benchmarks, on an average of 200 seconds per KLOC. After ranking, SmallRace reports the top 100 races at most for each image. For each race report, we inspected the races from the top until at most 10 races (manually by both the authors and the developers). With the detailed call stack and racy variable provided by SmallRace, the effort of analyzing a race report is minimized. In total, SmallRace detected about 400 races, and most of them are considered as either real races or mild races. A real race indicates that the race can lead to bad consequences in production, and should be fixed. A mild race indicates that the race is a valid race, but the consequence is either intended or benign, and developers will not fix it immediately. A common case of mild races is setter/getter races on boolean variables, where setters or getters can be invoked by a UI thread at any time.

B. *One example of the Real Race*

Lst. 4 shows a race found by SmallRace in MemoryMonitorApp, which is a long-running component involving multiple threads started at the beginning of the application. The race is on `loggerLock`, a lock used to protect logging operations from multiple threads. On Line 350, the lazy initialization of `loggerLock` may race with a concurrent write to `loggerLock` on Line 356. The consequence of this race is that multiple different `loggerLock` may be created

```
line 356, column 1 in MemoryMonitor.st AND line
    350, column 3 in MemoryMonitor.st
Shared variable: at line 350 of MemoryMonitor.st
 350| ( loggerLock isNil ) ifTrue: [ self
    loggerLock: RecursionLock new ].
Thread 1 (write):
>356| loggerLock := aSemaphore</body>
>>>Stack Trace:
>>>    st.startUp$MemoryMonitor
>>>      st.startLoggerProcess
    [MemoryMonitor/MemoryMonitor.st:979]
>>>        st.startLoggerProcess$MemoryMonitor
>>>          st.forkAt:
    [MemoryMonitor/MemoryMonitor.st:1125]
>>>            st.loggerLock
    [MemoryMonitor/MemoryMonitor.st:1131]
>>>              st.loggerLock$MemoryMonitor
>>>                st.loggerLock:
    [MemoryMonitor/MemoryMonitor.st:350]
Thread 2 (read):
>350| ( loggerLock isNil ) ifTrue: [ self
    loggerLock: RecursionLock new ].
>>>Stack Trace:
>>>    st.imageStartUpComplete$MemoryMonitor
>>>      st.fork
    [MemoryMonitor/MemoryMonitor.st:877]
>>>        st.startImageLockupProcess
    [MemoryMonitor/MemoryMonitor.st:890]
>>>
    st.startImageLockupProcess$MemoryMonitor
>>>            st.forkAt:
    [MemoryMonitor/MemoryMonitor.st:1070]
>>>              st.imageLockupDetected:
    [MemoryMonitor/MemoryMonitor.st:1085]
>>>
    st.imageLockupDetected:$MemoryMonitor
>>>                  st.logMessage:
    [MemoryMonitor/MemoryMonitor.st:1055]
>>>                    st.logMessage:$MemoryMonitor
>>>                      st.loggerLock
    [MemoryMonitor/MemoryMonitor.st:1119]
```

Listing 4. Races found in MemoryMonitorApp

resulting in the lock protection. From the developer: "*This means that the* `loggerLock` *is "almost" created during instantiation, but because it is within the fork and at a lower priority, the loggerProcess is executed much later. This is a potential timing issue, depending on where the execution is when loggerProcess task is pre-empted, but I can see opportunities for failure. This is a good catch.*"

C. Limitations and Future Work

SmallRace does not offer soundness or completeness guarantees. SmallRace can introduce false positives as it does not reason about branch conditions. We perform multiple analyses to eliminate false positives. Specifically, we utilize the happens-before relationship, lock set algorithm, and local variable filtering. Despite these efforts, a significant number of false positives occur due to the lack of condition reasoning in infeasible stack traces.

Also, our tool has false negatives due to the over-approximation of pointer analysis and the trade-off between efficiency and precision (the number of pointsTo variable for each memory access). Considering the incompleteness of the code base, information may be absent such as some super class's definitions, which can introduce imprecision. As we do not have access to the version history of the code base, it is not feasible to calculate the false negatives rate.

There is still room for improvement. Some semantic and language features are yet to be implemented due to implementation costs. The precision of the underlying data flow analysis and type inference can be further improved with the implementation of more language features in Smalltalk (e.g., heterogeneous types). We will leave those as future work.

VII. Related Work

Although race detection has been a fruitful research topic, we are not aware of any prior research on detecting race conditions in Smalltalk. There are limited studies available that focus on the static analysis of dynamic languages. Besides the design-level novelty, what's distinguished in SmallRace is the ability to deal with dynamic language features in Smalltalk such as code blocks and type inferences. Most prior work on race detection has focused on statically typed programs, such as Java or C/C++.

The LanguageKit framework developed by David Chisnall [34] is the closest to our work. It implements a reusable AST, interpreter, JIT, and static compiler for dynamic object-oriented languages, including a Smalltalk front-end. It also generates LLVM IR and allows the mixing of Smalltalk and C code. LanguageKit proposed a frontend for Pragmatic Smalltalk, which is a different dialect of Smalltalk Language closely tied to Étoilé OS. Different from SmallRace, it does not have pointer analysis and does not address the various technical challenges in detecting race conditions, such as type inference, code blocks, scopes, thread traces, modeling synchronization semantics, and identifying shared variables.

Banerjee, Utpal, et al. [35] propose a rigorous mathematical theory for data race detection. Most race detectors adopt either static analysis [2], [3], [36] or dynamic analysis [37]–[44], and use happens-before [37], [39], [42], [45]. Using a static approach allows for the analysis of source code without the need to set up the execution environment. Chord [2] is a static race detection tool for Java combining a series of static analysis techniques to reduce false positives. D4 [3] presents a fast concurrency analysis framework that detects concurrency bugs in real time. RacerD [36] detects data races in Java efficiently and is scalable with continuous integration.

A dynamic data race analyzer processes the stack trace on a particular execution path, and captures necessary information during runtime. ThreadSanitizer [37] adopts a hybrid algorithm [38] based on the happens-before and locksets algorithm [29]. Helgrind+ [39] is an efficient dynamic race detector to detect synchronization defects in parallel threads, which incorporates condition variables.

VIII. Conclusions

We present the design and implementation of SmallRace, the first end-to-end static race detection framework for Smalltalk. SmallRace addresses several technical challenges in analyzing dynamic language features and in reasoning about threads and

race conditions. This framework has demonstrated real value in an industrial setting for addressing a highly challenging problem. Our evaluation shows that SmallRace is fast, precise, and scalable. It is easy to use, achieves high coverage, and has been applied to a large-scale complex Smalltalk codebase from our industry partners, discovering 37 new race conditions confirmed by the developers.

References

[1] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler," in *International Conference on Runtime Verification*. Springer, 2011, pp. 110–114.

[2] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 308–319.

[3] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 359–373, 2018.

[4] Y. Li, B. Liu, and J. Huang, "Sword: A scalable whole program race detector for java," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 75–78.

[5] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang, "Ompracer: A scalable and precise static race detector for openmp programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[6] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: effectively spotting data races in large openmp applications," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 53–62.

[7] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[8] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: Compositional static race detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 144:1–144:28, 2018.

[9] B. Liu, P. Liu, Y. Li, C.-C. Tsai, D. Da Silva, and J. Huang, *When Threads Meet Events: Efficient and Precise Static Race Detection with Origins*. New York, NY, USA: Association for Computing Machinery, 2021, p. 725–739. [Online]. Available: https://doi.org/10.1145/3453483.3454073

[10] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 205–214.

[11] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 33, no. 1, pp. 1–55, 2011.

[12] C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen, "Repairing event race errors by controlling nondeterminism," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 289–299.

[13] D. Ingalls, "The evolution of smalltalk: From smalltalk-72 through squeak," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386335

[14] "The LLVM Compiler Infrastructure Project," https://llvm.org/, 2003.

[15] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[16] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, pp. 1–24, 2015.

[17] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[18] D. A. Terei and M. M. Chakravarty, "An llvm backend for ghc," in *Proceedings of the third ACM Haskell symposium on Haskell*, 2010, pp. 109–120.

[19] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[20] S. Wanderman-Milne and N. Li, "Runtime code generation in cloudera impala." *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 31–37, 2014.

[21] K. Dharsee, E. Johnson, and J. Criswell, "A software solution for hardware vulnerabilities," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 27–33.

[22] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 71–86.

[23] "Custom Software Application Development Services - Cincom VisualWorks® — Cincom Smalltalk®," https://www.cincomsmalltalk.com/main/products/visualworks/, 1999.

[24] "GNU Smalltalk - GNU Project - Free Software Foundation (FSF)," https://www.gnu.org/software/smalltalk/, 2003.

[25] "Pharo - Welcome to Pharo!" https://pharo.org/, 2008.

[26] "Squeak Smalltalk," https://squeak.org/, 1996.

[27] "SmalltalkX Basic classes Processes," https://live.exept.de/doc/online/english/overview/basicClasses/process.html, 1996.

[28] "ANTLR," https://www.antlr.org/index.html, 1992.

[29] 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 (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[30] "grammars-v4/Smalltalk.g4 at master · antlr/grammars-v4," https://github.com/antlr/grammars-v4/blob/master/smalltalk/Smalltalk.g4, 2013.

[31] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[32] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[33] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 305–315. [Online]. Available: https://doi.org/10.1145/1806596.1806631

[34] D. Chisnall, "Smalltalk in a c world," in *Proceedings of the International Workshop on Smalltalk Technologies*, 2012, pp. 1–12.

[35] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, 2006, pp. 69–78.

[36] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[37] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[38] R. O'callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2003, pp. 167–178.

[39] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–13.

[40] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: Effective sampling for lightweight data-race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 134–143.

[41] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel." in *OSDI*, vol. 10, no. 10, 2010, pp. 1–16.

[42] J. Thalheim, P. Bhatotia, and C. Fetzer, "Inspector: data provenance using intel processor trace (pt)," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 25–34.

[43] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 251–262, 2012.

[44] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proceedings of the 2013 ACM SIGPLAN*

international conference on Object oriented programming systems languages & applications, 2013, pp. 151–166.

[45] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang, "Detecting atomicity violations for event-driven node. js applications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 631–642.