RC22146 (W0108-016) August 9, 2001
Computer Science

IBM Research Report

Static Datarace Analysis for Multithreaded Object-Oriented Programs

Jong-Deok Choi, Alexey Loginov, Vivek Sarkar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Static Datarace Analysis for Multithreaded Object-Oriented Programs

Jong-Deok Choi[†*] Alexey Loginov[†] Vivek Sarkar[†]

IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract

This paper presents a novel analysis framework and algorithm for statically identifying dataraces in multithreaded object-oriented programs. The framework shows how *datarace analysis* can be formulated as a conjunction of *interthread control flow analysis* and *points-to analysis* of *thread objects*, *synchronization objects* and *access objects*. This formulation can be used to identify a spectrum of dataraces depending on the precision of points-to and control flow information received as input. The framework can be used for datarace analysis of programs written in any multithreaded object-oriented language that supports creation of thread objects, monitor-like synchronization of threads via object-based locking, and global memory accesses via static and instance fields.

Our datarace analysis algorithm operates interprocedurally over the *interthread call graph (ICG)* and extracts ordering information by examining the thread creation sites and the object references used for monitor synchronization. The algorithm computes two classes of data races — *potential* (no false negatives) and *definite* (no false positives). We present experimental results obtained from a preliminary implementation of the datarace analysis algorithm in the Jalapeño JVM. The results show that our preliminary implementation can be effective in narrowing down the set of statement pairs that may particpate in a datarace. This information could be used in a static analysis tool, or as a filter for reducing the overhead of dynamic datarace detection.

1 Introduction

Multithreaded programs can be classified into three categories. *Deterministic* multithreaded programs exhibit the same functional behavior whenever they are executed with the same set of inputs[1]. *Nondeterministic datarace-free* programs exhibit the same functional behavior whenever the same program is executed with the same set of inputs *and* the same execution order of synchronization operations among threads. We refer to programs that do not belong to either of the previous two categories as "*racy*" programs.

A racy program can exhibit different functional behaviors when it is executed repeatedly with the same set of inputs and the same execution order of synchronization (acquire/release) operations. This degree of nondeterminism is caused by events in different threads such that (1) they access the same memory location without ordering constraints among them, and (2) at least one of them writes into the memory location[2]. These events are called *dataraces.* In almost all cases, a datarace is a programming error. Since racy programs are notoriously difficult to debug, it has been recognized that tools for automatic detection of dataraces in multithreaded programs can be extremely valuable. As a result, there has been a substantial amount of past work in building tools for analysis and detection of dataraces [19, 16, 29, 21, 27, 34, 42, 28, 37, 20, 35].

Dynamic datarace detection tools detect dataraces at runtime for a specific program execution [19, 16, 29, 21, 27, 35]. *Static datarace analysis* tools, on the other hand, consider the space of all possible program executions and identify dataraces that might occur in one of them [44, 30, 42, 26, 3, 20]. The output of static datarace analysis is a set of statement pairs that can generate the identified dataraces. Static datarace analysis and dynamic datarace detection are complementary techniques, and should ideally be used in conjunction. For example, static datarace analysis can help reduce the runtime cost of dynamic datarace detection by filtering out memory accesses that are known to not participate in any datarace. The specific benefits of static datarace analysis compared to dynamic datarace detection are as follows: (1) its results are execution-independent; (2) it does not perturb the timing of the execution with in-

*Contact author.

[†]e-mail addresses: jdchoi@us.ibm.com, alexey@cs.wisc.edu, vsarkar@us.ibm.com

[1] For convenience, we consider all interactions between the program and its external world as "inputs" *e.g.,* file I/O and system calls such as `timeofday()` and `random()`.

[2] Under certain memory models, two read accesses can also generate a datarace [36]. The framework presented in this paper can be easily applied to such models by dropping the requirement for a write access in a datarace.

strumentation; and (3) its results can be used to safely optimize certain memory accesses (since a memory access that does not participate in a datarace can usually follow a more relaxed memory model than the default memory model [36]).

The bulk of past work on static datarace analysis was targeted to fork-join programs, in which concurrency is specified by parallel control structures such as cobegin-coend and parallel loops [11, 28, 3]. However, those results are not applicable to the object-based concurrency models present in Java [2] and other multithreaded object-oriented programming languages. A recent exception is the work on type-based datarace detection by Flanagan and Freund [20], in which a combination of user annotations and type-based analysis is used to identify potential dataraces. Object-based concurrency has also been considered in recent work on dynamic datarace detection [35]. However, we are unaware of any other past work on static datarace analysis of object-based concurrent programs that goes beyond type analysis.

This paper presents a novel analysis framework and algorithm for statically identifying dataraces in multithreaded object-oriented programs. The framework shows how *datarace analysis* can be formulated as a conjunction of *interthread control flow analysis* and *points-to analysis* of *thread objects*, *synchronization objects* and *access objects*. Both path-specific and all-path formulations of points-to analyses are used as foundations for the framework. The framework employs two different types of *points-to* information, namely *may points-to* and *must points-to*, to identify a spectrum of dataraces. At one end of the spectrum are *potential dataraces*, which include no false negatives — this set must include all event pairs that may generate dataraces in any program execution in which they both get executed. At the other end are *definite dataraces*, which include no false positives — this set only includes event pairs that must generate dataraces in all program executions in which they both get executed. We also provide the correctness proofs of the formulations for computing the potential and the definite dataraces.

We present experimental results obtained from a prototype implementation of our datarace analysis algorithm in the Jalapeño Java Virtual Machine (JVM) [9]. The datarace analysis algorithm operates interprocedurally over the *interthread call graph (ICG)* and extracts ordering information by examining thread creation sites and object references used for monitor synchronization.

The rest of the paper is organized as follows. Section 2 illustrates datarace examples. Section 3 defines the conditions for a datarace to occur and their relationship with points-to analysis. Section 4 presents the program representations we use in our analysis framework. Section 5 presents the formulation of the path-specific points-to analyses, and Section 6 presents the formulation of the path-specific datarace analyses. Section 7 presents all-path points-to analyses and all-path datarace analyses. Section 8 describes the datarace analysis algorithm we have developed and implemented for the all-path datarace analyses. Section 9 presents the experimental results of applying our framework and algorithm to Java benchmark programs. Section 10 describes related work, and Section 11 concludes the paper.

2 Datarace Examples

In this section, we discuss an example multithreaded Java program and two different execution instances of this program. The program and its instances will also be used as a running example in the rest of the paper. Though our approach is applicable to multithreaded object-oriented programs in general, our implementation experience has been specifically with Java programs so Java is the programming language that we will use when discussing examples.

Figure 1 shows an example program with two classes: `MainThread` and `ChildThread`. Statements are labeled with statement numbers such as `S11`, the first statement of the `main` thread. We will also use the notation *thread.stmt* to denote a statement instance in a thread. For convenience, statements that are not relevant to dataraces have been elided from this program. After creating and starting threads `T1` and `T2`, thread `main` invokes method `m1()` in a synchronized block starting at `S15`. The synchronized block is guarded by the *synchronization object* pointed to by static field `MainThread.p`. (A synchronization object is an object whose lock is used to guard a synchronized block or method.) Thread `main` then waits at `S17` for thread `T2` to terminate, after which it executes `S18`.

Threads `T1` and `T2` execute the same code — the `run()` method in class `ChildThread`. They both attempt to enter the synchronized block starting at `S31`, guarded by the synchronization object pointed to by instance field `a`. If both thread objects have the same value of `a`, they will execute the synchronized block in mutual exclusion but the order in which they acquire the lock is nondeterministic. Also, whether or not the two synchronized blocks at `S15` and `S31` are guarded by the same synchronization object will depend on whether or not fields `p` and `a` point to the same object. This is one of the reasons why points-to analysis is an essential foundation for datarace analysis.

Figures 2-(A) and 2-(B) show the timing diagrams for two execution instances of the example program. The solid vertical edges (labeled *thread edge*) represent serial execution within a thread. The dashed cross edges represent interthread event ordering relationships arising

```
// thread main
class MainThread {
   static Obj p, q, x;
   public static void main(String args[]) {
      . . .
S11:  MyThread T1 = new ChildThread(...);
S12:  MyThread T2 = new ChildThread(...);
S13:  T2.start();
S14:  T1.start();
      . . .
S15:  synchronized(p) {
S16:     m1(q);
      } // synchronized
      . . .
S17:  T2.join(); // wait until T2 terminates
S18:  x.f = 200;
   }

   public static void m1(Obj y) {
S20:  y.f = 100;
   }
} // class MainThread
```

```
// thread T1, T2
class ChildThread implements Runnable {
   Obj a, b, c
   public void run() {
S30:
S31:  synchronized (a) {
S32:     MainThread.m1(b);
S33:     m2(c);
      } // synchronized
S34:
   }

   public void m2(Obj w) {
S40:  . . .
S41:  Obj z = new Obj(...);
      . . .
S42:  z.g = ...
   }
} // class ChildThread
```

Figure 1: Example Program with Three Threads.

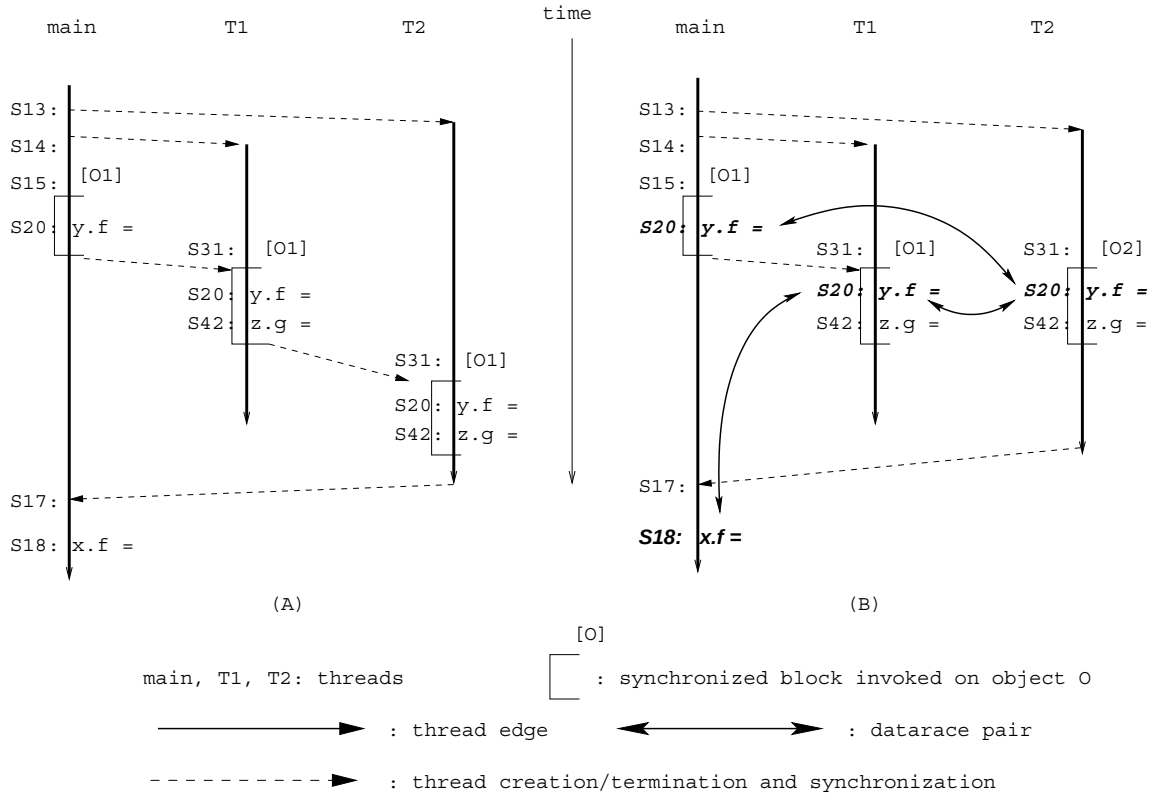

Figure 2: Two Example Execution Instances of the Program of Figure 1.

from thread creation, termination, and synchronization. In both instances, the cross edges from `main.S13` and `main.S14` represent the event ordering arising from the main thread's starting up of threads `T2` and `T1` respectively. The cross edge from the end of `T2` to `main.S17` is due to the `join()` call at `S17`. The remaining cross edges represent event ordering due to execution of synchronized blocks. Together, the thread edges and the cross edges define a partial order or a "happens-before" relationship among the statement instances.

The execution instance in Figure 2-(A) corresponds to the case when `MainThread.p` = `T1.a` = `T2.a` = `O1`, and thread `main` enters synchronized block `S15` before thread `T1` enters `S31`, and `T1` enters `S31` before thread `T2`. In this instance, all three synchronized block executions are guarded by the same object, `O1`. Further, the order in which the three blocks acquires `O1`'s lock causes all (relevant) statement instances in the three threads to be *totally ordered*. Therefore, the instance in Figure 2-(A) does not exhibit a datarace.

Now, consider a variation of this instance that is not shown in the figure. If thread `T2` enters synchronized block `S31` before `T1`, then there is a possibility of a datarace between `T1.S20` and `main.S18`. For the datarace to occur, it is necessary for fields `T1.b` and `MainThread.x` to point to the same object (another application of points-to analysis!) Such a datarace would be detected by a dynamic datarace detection tool if this variation in execution were to occur, but not if the tool was presented with the original execution instance in Figure 2-(A). This illustrates a key limitation of a purely dynamic approach to datarace detection.

The execution instance in Figure 2-(B) corresponds to the case when `MainThread.p` = `T1.a` = `O1`, but `T2.a` = `O2` and `O1` and `O2` are distinct objects. As in the previous instance, the main thread enters synchronized block `S15` before thread `T1` enters `S31`. However, since thread `T2` uses a different synchronization object than the main thread or thread `T1`, this instance exhibits dataraces that were not present in Figure 2-(A). Specifically, there is a datarace between `T1.S20` and `T2.S20`, assuming that `T1.b` = `T2.b`. There's also a datarace between `main.S20` and `T2.S20`, assuming that `MainThread.q` = `T2.b`. Finally, since `MainThread.q` = `T1.b` must also be true, there is a datarace between `T1.S20` and `main.S18`. The three dataraces are shown as double arrows in Figure 2-(B).

We now briefly discuss why some other pairs of statements were not identified as dataraces. Consider statement instance `T2.S42`, which writes to field `z.g`. There will not be a datarace between this statement and any access of field `f`, because a strongly-typed object-oriented language guarantees that accesses to distinct fields can never be aliased. Further, there will not be a datarace between `T1.S42` and `T2.S42` because `T1.z` [3] and `T2.z` point to distinct objects. This fact is hard to discern by (context-insensitive) points-to analysis, but can be easily determined by *escape analysis* [14, 5, 6, 38]. Since object `z` allocated in `S41` *does not escape* method `m2`, `z` must be a thread-local object and `T1.z` and `T2.z` must point to distinct objects. This observation shows how escape analysis can supplement points-to analysis when analyzing a program for static dataraces.

3 Datarace Conditions, Points-To Analysis, and Escape Analysis

Using the insights from the previous section, we identify the following four conditions, called *datarace conditions*, for two events to cause a datarace in a multithreaded object-oriented program: (1) the two events access the same memory location (*i.e.,* the same field/element in the same class/object); (2) the two events are executed by different thread objects; (3) the two events are not guarded by the same synchronization object; and (4) there is no execution ordering enforced between the two events by thread creation or termination.

The objective of static datarace analysis is to identify statements for which some execution instances may satisfy all four datarace conditions. We use *points-to analysis* of reference variables as the basis for static datarace analysis. Specifically, points-to analysis of access objects, thread objects, and synchronization objects contributes to static analysis of conditions (1), (2), and (3) respectively. Points-to analysis of thread objects also contributes to analysis of condition (4).

Points-to analysis in general subsumes escape analysis in determining whether two statements executable by different threads can generate a datarace. A separate escape analysis, however, can be useful for determining whether two statements reachable from the same `run()` method can generate a datarace. Consider `S42`, which has two execution instances: one in `T1` and the other in `T2`. Since only a single statement is involved, a sophisticated points-to analysis may be needed to determine whether execution instances of the statement in different threads may point to the same object or not. However, a simple escape analysis might be able to recognize that the object pointed to by `z` never escapes the creating thread, and, therefore, execution instances of `S42` in different threads can never point to the same object.

[3] `T1.z` is shorthand for local `z` in method `m2` under the control of thread `T1`.

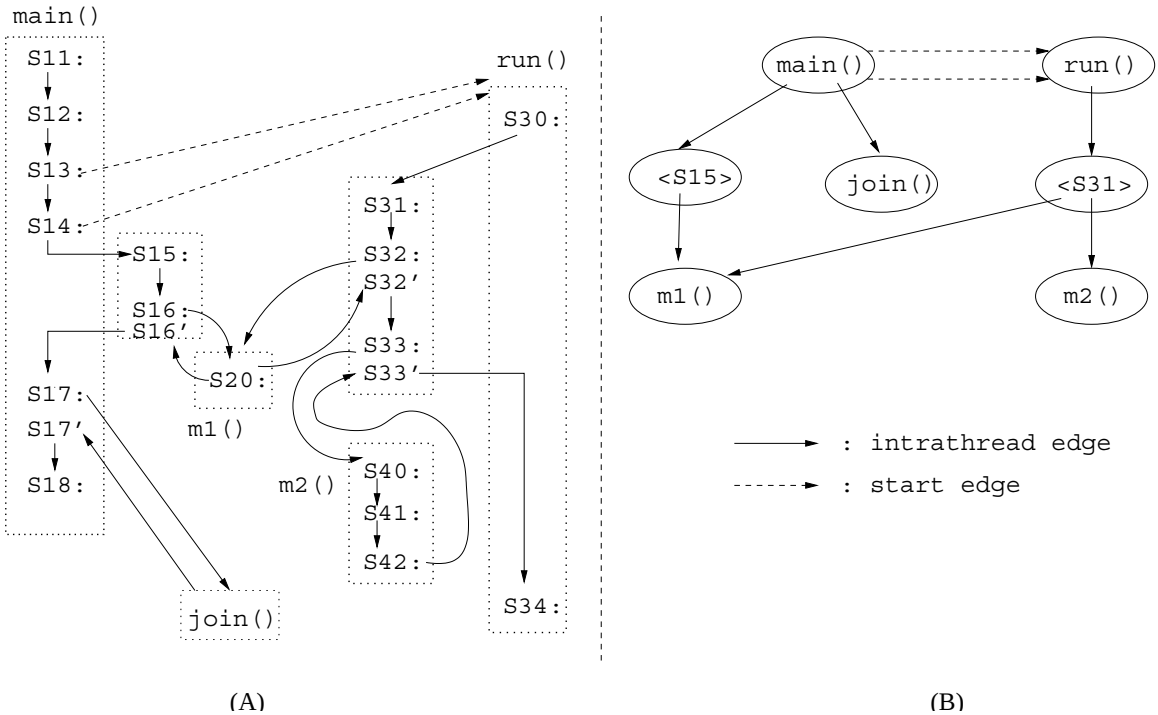

Figure 3: Interthread Control Flow Graph (A) and Interthread Call Graph (B) of the Example Program in Figure 1.

4 Foundations

In this section, we introduce the *interthread control flow graph (ICFG)* and the *interthread call graph (ICG)*. The ICFG serves as the foundation for other definitions in the paper, but is too detailed for effective use in analysis of large programs. The ICG is an abstraction of the ICFG, and is the representation that is actually employed for datarace analysis.

Our approach is applicable to programs written in any multithreaded object-oriented language that supports thread creation, and monitor-like synchronization via objects such as *synchronized methods* and *synchronized blocks*. The only thread operation not currently supported by our framework is `join`. Ignoring joins can be a source of spurious dataraces in our analysis. However, we do not expect this to be a major factor in practice because monitors are used more frequently than join operations for serializing concurrent data accesses.

that might not otherwise

4.1 Interthread Control Flow Graph (ICFG)

The ICFG is a detailed interprocedural representation of a multithreaded program in which nodes represent instructions and edges represent control flow. Without loss of generality, we assume that instructions are *fine-grained*— *i.e.,* each instruction contains at most one use (read) or one def (write) of a memory location. We will use the term *access* to refer to a use or a def. Each method and each synchronized block has distinguished *entry* and *exit* nodes in the ICFG. We refer to the entry node of the method that starts program execution (*e.g.,* the entry node of the main method in a Java program) as the *root* of the ICFG.

An ICFG contains four types of control flow edges: *intraprocedural* [4], *call*, *return*, and *start*. The first three types are present in a standard *interprocedural control flow graph* [10]. Start edges are unique to the ICFG, and are defined as follows. A start edge is included from each *thread-start node* to the entry node of all compatible target methods. In a Java program, a thread-start node is a node containing a `Thread.start()` method call site, and a target method is a compatible `run()` method[5]. The assumption is that a thread start call will initiate execution of a new thread at its `run()` method; all other invocations of a `run()` method execute as part of the calling thread. Start edges are referred to as *interthread edges*, while all other edges in the ICFG are called *intrathread edges*. Further, any edge that is not an intraprocedural edge is called

[4] We assume that the intraprocedural edges capture all intraprocedural control flow, including control flow arising from exceptions.

[5] In this paper, the term "`run()`" for a Java program refers to the `run()` method in a class that directly or indirectly implements the `Runnable` interface.

an *interprocedural* edge. The entry node that is a target of a start edge is called a *thread-root node*.

Figure 3-(A) shows the Interthread Control Flow Graph (ICFG) for the example program in Figure 1. Each node represents a distinct statement. The dotted lines are used to group together ICFG nodes that get combined into the same ICG node in Figure 3-(B).) Solid arrows represent intrathread edges, and dashed arrows represent interthread edges.

An ICFG path without any interprocedural edges is an *intraprocedural path*, and an ICFG path with one or more interprocedural edges is an *interprocedural path*. An ICFG path without any interthread edges is an *intrathread path*, and an ICFG path with one or more interthread edges is an *interthread path*.

A *realizable ICFG path* is defined below as an ICFG path with matching call and return edges that starts at the root node. Unless otherwise specified, we will use the term "path" or "ICFG path" in the remainder of this paper to denote a realizable ICFG path.

Definition 4.1 *A* matching path *is defined recursively as follows: (1) An intraprocedural edge is a matching path. (2) A start edge is a matching path. (3) A path of length 0 is a matching path. (4) A path that is the concatenation of two matching paths is also a matching path. (5) Let* $\stackrel{c}{\rightarrow}$ *and* $\stackrel{r}{\rightarrow}$ *be a matching pair of invocation and return edges. Then, an ICFG path* $\pi : \stackrel{c}{\rightarrow} \pi' \stackrel{r}{\rightarrow}$ *is a matching path if ICFG path* π' *is a matching path.*

Definition 4.2 *A* realizable ICFG path *is a directed path that starts at the root node of the ICFG, and is a prefix of a matching path.*

We observe that each *dynamic execution instance* of an ICFG node from the input program can be characterized by a unique ICFG path, which represents the instruction sequence that was executed to cause the dynamic execution instance to be reached. This path may consist of intraprocedural, call, return, and start edges. Different dynamic instances will be characterized by different ICFG paths. (Note that ICFG paths can contain cycles to represent different loop iterations and recursive calls.) For example, the three possible dynamic instances of statement `S20` in Figure 3-(A) can be identified by the ICFG paths $\langle S11, S12, S13, S30, S31, S32, S20\rangle$, $\langle S11, S12, S13, S14, S30, S31, S32, S20\rangle$, and $\langle S11, S12, S13, S14, S15, S16, S20\rangle$.

Finally, we introduce the notion of *mutually sequential* path. Our goal is to identify cases when two dynamic instances of ICFG nodes are guaranteed to not have executed in parallel. The predicate corresponds to the negation of the fourth datarace condition in Section 3.

Definition 4.3 *Two ICFG paths* π *and* π' *are* mutually sequential*, written as* $MuSeq(\pi, \pi')$*, if one is a prefix of the other.*

4.2 Interthread Call Graph (ICG)

Figure 3-(B) shows the Interthread Call Graph (ICG) of the ICFG in Figure 3-(A). The ICG is a reduced abstraction of the ICFG, designed to be practical and scalable for analysis of large programs. For convenience, we define an ICG by specifying how it can be constructed from an ICFG. However, in practice, an ICG is constructed directly from the program, without the need to first build an ICFG.

An ICG node is created for each method and each synchronized block in the ICFG. The inclusion of separate ICG nodes for synchronized blocks is a notable difference between the ICG and standard call graphs.We call a node in the ICG a *synchronized node* if it represents either a synchronized method or a synchronized block.

An ICG contains two types of edges: *call*, and *start*. For each *call* edge in an ICFG, an ICG *call* edge is created from the ICG node for the caller method to the ICG node for the callee method (as in standard call graphs). Similarly, for each *start* edge in the ICFG, an ICG *start* edge is created from the ICG node for the caller method to the ICG node for the target `run()` method. Finally, edges are created in the ICG to reflect control flow into synchronized blocks: for each *intraprocedural* ICFG edge that targets the *entry* node of a synchronized block, an ICG *call* edge is created from the ICG node representing the method containing the source of the ICFG edge to the ICG node representing the synchronized block. Each edge in the ICG is labeled with the instruction number representing the transfer of control (a `call`, `start`, or `synchronized` instruction).

5 Path-Specific Points-To Analysis

In this section, we present a formulation for *path-specific points-to analysis*. This formulation provides the foundation for defining *path-specific datarace analysis* in Section 6. Path-specific points-to analysis computes dataflow facts on reference (i.e., pointer) variables along a specific ICFG path. We are unaware of any prior formulations of path-specific points-to analysis.

Path-specific analyses are more precise, but usually more expensive to compute, than *all-path* analyses. Path-specific datarace analysis can be useful for identifying specific execution paths in a multithreaded program that exhibit dataraces. This information could be useful in

determining the input data or thread schedules necessary to force a program to exhibit certain dataraces [43]. However, since path-specific analyses can be expensive, we show in Section 7 how the path-specific analyses presented in this section and Section 6 can be widened to *all-path points-to analysis* and *all-path datarace analysis*.

A dynamic execution of a multithreaded object-oriented program involves allocating *concrete* (runtime) objects, and storing references to the objects in local, global or field reference variables. When a reference variable has a non-null value, we say that it *points to* the object for which it contains a reference.

The goal of static points-to analysis is to statically approximate relationships among concrete objects [13, 23, 18]. One approach, which forms the basis of our datarace analysis framework, is to statically create *abstract* objects during analysis, associate them with accesses (defs/uses) of reference variables in the program. We say that the abstract objects associated with an access are being *pointed to* by the access. Each abstract object pointed to by an access represents one or more concrete objects pointed to by instances of the access during dynamic execution.

If access x is a dereference, e.g., `x.g` to access field `g`, $Field(x)$ identifies the field accessed at $Node(x)$: i.e., $Field(x) = \mathtt{g}$. If x is not a dereference, $Field(x) = \perp$ by convention.

For a program path π on the ICFG, we define $end(\pi)$ to be the end node of π. (Recall that all program paths start at the root node, by definition.) In this paper, we will limit our attention to paths with end nodes that contain an access. For convenience, we define $Access(\pi) = x$ where x is the access contained in $end(\pi)$. We also define $Field(\pi)$ to be a shorthand for $Field(Access(\pi))$.

Let π be an ICFG path and $x = Access(\pi)$. The *may points-to set* of x along π, $MayPT(\pi)$, denotes the set of abstract objects that a points-to analysis computes to be pointed to by x along path π. We define the *must points-to set* of x along π, $MustPT(\pi)$, as follows:

$$MustPT(\pi) = \begin{cases} MayPT(\pi) & \text{if } singleton(MayPT(\pi)) \\ \emptyset & \text{otherwise} \end{cases}$$

where $singleton(MayPT(\pi)) = true$ if $MayPT(\pi)$ is a singleton set of an abstract object that represents at most one concrete object.

Let $\pi' = thStart(\pi)$ be the longest prefix of π such that $end(\pi')$ is a thread-start node. (In Java, this means that $end(\pi')$ is the thread-start node of a new thread, and this new thread executes $end(\pi)$ along π. By convention, if $end(\pi)$ is in the main thread, we define $end(\pi')$ to be the hypothetical access of the main thread by the system thread that starts the main thread.) We define the path-specific *thread points-to sets* of x along π as follows:

$$MayThread(\pi) = MayPT(\pi') \quad (1)$$

$$MustThread(\pi) = MustPT(\pi') \quad (2)$$

Next, let $\{\pi_1, \pi_2, \ldots, \pi_n\}$ be the set of all subpaths of π such that $end(\pi_k), 1 \leq k \leq n$, corresponds to an invocation of a synchronized method or block. We define the path-specific *synchronization points-to sets* of access x along π as follows:

$$MaySync(\pi) = \bigcup_{1 \leq k \leq n} MayPT(\pi_k) \quad (3)$$

$$MustSync(\pi) = \bigcup_{1 \leq k \leq n} MustPT(\pi_k) \quad (4)$$

Finally, for consistency, the path-specific *accessed-object points-to sets* of x along π are defined trivially as follows:

$$MayAccess(\pi) = MayPT(\pi) \quad (5)$$

$$MustAccess(\pi) = MustPT(\pi) \quad (6)$$

6 Path-Specifc Datarace Analysis

In this section, we present our framework for path-specific static datarace analysis. The framework builds on the formulation of path-specific points-to analysis introduced in the previous section, and provides formal path-specific definitions for definite and potential dataraces using may/must points-to sets. Path-specific dataraces will be used as a building block in defining all-path dataraces in Section 7. However, they may also serve as a usual foundation in formalizing dynamic datarace detection, since there's a one-to-one correspondence between an ICFG path and the implied dynamic execution instance of the last instruction in the path. We are unaware of any prior formulations of path-specific datarace analysis.

6.1 Path-Specific Points-To Predicates

Let π and π' be two program paths from the root to nodes n and n' containing accesses $x = Access(\pi)$ and $x' = Access(\pi')$ respectively. We will now introduce predicates for the cases when x and x' are references to the same access object, are executed by the same thread object, and are guarded by the same synchronization objects. These predicates are defined using the points-to sets introduced in Section 5 and are related to the first three datarace conditions in Section 3.

The *conflicting-access predicates* determine when x and x' access the same memory location, and are defined as follows:

$$MayConfAcc(\pi, \pi') = (MayAccess(\pi) \cap MayAccess(\pi') \neq \emptyset) \ \wedge$$

$$(Field(\pi) = Field(\pi'))$$

$$MustConfAcc(\pi, \pi') = (MustAccess(\pi) \cap MustAccess(\pi') \neq \emptyset) \wedge (Field(\pi) = Field(\pi'))$$

The *same-thread predicates* determine when x and x' are executed by the same thread object (these predicates correspond to the negation of the second datarace condition in Section 3):

$$MaySameTh(\pi, \pi') = MayThread(\pi) \cap MayThread(\pi') \neq \emptyset$$

$$MustSameTh(\pi, \pi') = MustThread(\pi) \cap MustThread(\pi') \neq \emptyset.$$

Finally, the *mutually-synchronized predicates* determine when paths π and π' contain a common synchronization object (these predicates correspond to the negation of the third datarace condition in Section 3):

$$MayMuSync(\pi, \pi') = MaySync(\pi) \cap MaySync(\pi') \neq \emptyset$$

$$MustMuSync(\pi, \pi') = MustSync(\pi) \cap MustSync(\pi') \neq \emptyset$$

6.2 Path-Specific Datarace Predicates

We now define the path-specific *potential datarace* ($PotentialDR$), and *definite datarace* ($DefiniteDR$) predicates for accesses $x = Access(\pi)$ and $x' = Access(\pi')$ as follows. For convenience, the predicates are actually defined in terms of π and π':

$$PotentialDR(\pi, \pi') = MayConfAcc(\pi, \pi') \wedge (\neg MustSameTh(\pi, \pi')) \wedge (\neg MustMuSync(\pi, \pi')) \wedge (\neg MuSeq(\pi, \pi')) \quad (7)$$

$$DefiniteDR(\pi, \pi') = MustConfAcc(\pi, \pi') \wedge (\neg MaySameTh(\pi, \pi')) \wedge (\neg MayMuSync(\pi, \pi')) \wedge (\neg MuSeq(\pi, \pi')) \quad (8)$$

6.3 Correctness

Recall from Section 4 that each dynamic instance e of an ICFG node n can be characterized by a unique ICFG path π that ends with node n. (Path π may contain multiple occurrences of node n, but the end node is the one that corresponds to instance e.) The following theorem states that $PotentialDR(\pi_1, \pi_2)$ is a necessary condition for two execution instances to generate a datarace, where π_1 and π_2 are the ICFG paths corresponding to the two instances.

Theorem 6.1 *Let e_1 and e_2 be two execution instances with ICFG paths π_1 and π_2, respectively, so that e_1 is the last instruction instance on π_1 and e_2 is the last instruction instance on π_1. Then, e_1 and e_2 can generate a datarace* only if $PotentialDR(\pi_1, \pi_2) = true$.

Proof (by contradiction): Assume that e_1 and e_2 generate a data race, but $PotentialDR(\pi_1, \pi_2) = false$. Then, at least one of the four sub-predicates of Equation 7 is false. Assume $MayConfAcc(\pi_1, \pi_2) = false$. Then, the object accessed by $Access(\pi_1)$ and the object accessed by $Access(\pi_2)$ cannot be the same. Without accessing the same object, the two execution instances cannot generate a datarace, thus leading to a contradiction. Similar arguments can be applied to the other three sub-predicates. □

The following theorem states that $DefiniteDR(\pi_1, \pi_2)$ is a *sufficient condition* for a datarace to be generated by two executions instances with ICFG paths π_1 and π_2.

Theorem 6.2 *Let e_1 and e_2 be two execution instances with ICFG paths π_1 and π_2, respectively, so that e_1 is the last instruction instance on π_1 and e_2 is the last instruction instance on π_1. Then, e_1 and e_2 generate a datarace* if $DefiniteDR(\pi_1, \pi_2) = true$.

Proof (by contradiction): Assume $DefiniteDR(\pi_1, \pi_2) = true$, but e_1 and e_2 do not generate a datarace. This is possible only when at least one of the four datarace conditions in Section 3 does not hold between π_1 and π_2. Assume the first condition that the two events access the same object does not hold. This assumption, however, contradicts the first sub-predicate of $DefiniteDR(\pi_1, \pi_2)$: $MustConfAcc(\pi_1, \pi_2)$). Similar arguments can be applied to the other datarace conditions and their corresponding sub-predicates of $DefiniteDR(\pi_1, \pi_2)$. □

Note that the specific use of must vs. may points-to information for access, thread and synchronization objects in the definitions of $PotentialDR$ and $DefiniteDR$ was critical for establishing the proofs of Theorems 6.1 and 6.2.

7 All-Path Analysis

In this section, we present a formulation of *all-path points-to analysis* and *all-path datarace analysis*.

7.1 All-Path Points-to Analysis

Let $\Pi^*(x)$ be the set of all program paths from the root to the ICFG node of an access x. We define the may points-to and must points-to sets of x over $\Pi^*(x)$ as follows:

$$MayPT_\perp(x) = \bigcup_{\pi \in \Pi^*(x)} MayPT(\pi)$$

$$MustPT_\perp(x) = \bigcap_{\pi \in \Pi^*(x)} MustPT(\pi)$$

In other words, $MayPT_\perp(x)$ is the *non-path-specific* (i.e, all-paths) may points-to set of x, and $MustPT_\perp(x)$ is the non-path-specific must points-to set of x.

We define the all-path *accessed-object points-to sets* of x as follows:

$$\begin{aligned} MayAccess_\perp(x) &= MayPT_\perp(x) \\ MustAccess_\perp(x) &= MustPT_\perp(x) \end{aligned}$$

For program path π from the root to access x, recall that $thStart(\pi)$ is the longest prefix of π to access y on π such that the ICFG node of y is a thread-start node. We define the all-path *thread points-to sets* of x as follows:

$$ThStart(x) = \{Access(thStart(\pi)) \mid \pi \in \Pi^*(x)\} \quad (9)$$

$$MayThread_\perp(x) = \bigcup_{y \in ThStart(x)} MayPT_\perp(y) \quad (10)$$

$$MustThread_\perp(x) = \bigcap_{y \in ThStart(x)} MustPT_\perp(y) \quad (11)$$

We define the all-paths *synchronization points-to sets* of x as follows:

$$MaySync_\perp(x) = \bigcup_{\pi \in \Pi^*(x)} MaySync(\pi) \quad (12)$$

$$MustSync_\perp(x) = \bigcap_{\pi \in \Pi^*(x)} MustSync(\pi) \quad (13)$$

Note that Equation (13) computes the set intersection of $MustSync$ sets, each of which, however, is computed as the union of the must points-to synchronization objects along the path (Equation (4)).

7.2 All-Path Datarace Analysis

Let x and y be two accesses. We define the all-paths conflicting-access predicates for accessing the same memory location by x and y as follows:

$$\begin{aligned} MayConfAcc_\perp(x,y) &= \\ &(MayAccess_\perp(x) \cap MayAccess_\perp(y) \neq \emptyset) \ \wedge \\ &(Field(x) = Field(y)) \quad (14) \\ MustConfAcc_\perp(x,y) &= \\ &(MustAccess_\perp(x) \cap MustAccess_\perp(y) \neq \emptyset) \ \wedge \\ &(Field(x) = Field(y)) \quad (15) \end{aligned}$$

We define the all-paths same-thread predicates for accesses by the same thread as follows:

$$\begin{aligned} MaySameTh_\perp(x,y) &= \\ &MayThread_\perp(x) \cap MayThread_\perp(y) \neq \emptyset \quad (16) \\ MustSameTh_\perp(x,y) &= \\ &MustThread_\perp(x) \cap MustThread_\perp(y) \neq \emptyset \quad (17) \end{aligned}$$

We define the all-path mutually-synchronized predicates as follows:

$$\begin{aligned} MayMuSync_\perp(x,y) &= \\ &MaySync_\perp(x) \cap MaySync_\perp(y) \neq \emptyset \quad (18) \\ MustMuSync_\perp(x,y) &= \\ &MustSync_\perp(x) \cap MustSync_\perp(y) \neq \emptyset \quad (19) \end{aligned}$$

We now define the all-paths predicates for potential, and definite dataraces between x and y as follows:

$$\begin{aligned} PotentialDR_\perp(x,y) &= \\ &MayConfAcc_\perp(x,y) \wedge (\neg MustSameTh_\perp(x,y)) \ \wedge \\ &(\neg MustMuSync_\perp(x,y)) \wedge PotentialPar_\perp(x,y) \quad (20) \\ DefiniteDR_\perp(x,y) &= \\ &MustConfAcc_\perp(x,y) \wedge (\neg MaySameTh_\perp(x,y)) \ \wedge \\ &(\neg MayMuSync_\perp(x,y)) \wedge DefinitePar_\perp(x,y), \quad (21) \end{aligned}$$

where

$$\begin{aligned} PotentialPar_\perp(x,y) &= \\ &\exists \pi \in \Pi^*(x) \exists \pi' \in \Pi^*(y). \neg MuSeq(\pi, \pi') \\ DefinitePar_\perp(x,y) &= \\ &\forall \pi \in \Pi^*(x) \forall \pi' \in \Pi^*(y). \neg MuSeq(\pi, \pi'). \end{aligned}$$

7.3 Complete and Sound Datarace Sets

The following theorem states that $PotentialDR_\perp(x,y)$ is a *necessary condition* for an execution instance of access x and an execution instance of access y to generate a datarace.

Theorem 7.1 *Let e_x and e_y be execution instances of accesses x and y, respectively. Then, e_x and e_y generate a datarace* only if $PotentialDR_\perp(x,y) = true$.

Proof: Similar to the proof for Theorem 6.1. □

We call the set of statement pairs of a program that satisfy the $PotentialDR_\perp$ predicate the *complete datarace set* of the program because any actual datarace that occurs during any execution has a corresponding statement pair in the set.

The following theorem states that $DefiniteDR_\perp(x,y)$ is a *sufficient condition* for an execution instance of access x and an execution instance of access y to generate a datarace.

Theorem 7.2 *Let e_x and e_y be execution instances of accesses x and y, respectively. Then, e_x and e_y generate a datarace* if $DefiniteDR_\perp(x,y) = true$.

Proof: Similar to the proof for Theorem 6.2. □

We call the set of statement pairs of a program that satisfy the $DefiniteDR_\perp$ predicate the *sound datarace set* of the program because any statement pair in the set will generate a datarace during any execution.

8 All-Path Datarace Analysis Algorithm

In this section, we describe the all-path datarace analysis algorithms that we have implemented in Jalapeño. The algorithms consist of three phases. First, we perform escape analysis to identify abstract objects that may be accessed by more than one thread. Second, we perform all-path points-to analysis that computes the set of abstract objects pointed to by each access in the program. Finally, we compute the all-path synchronization points-to sets, and also the predicates for the all-path dataraces, such as Equations (20)-(21), all in the same phase.

Escape analysis computes the set of *escaping objects*, which are objects (or memory locations) that may be accessed by more than one thread [14]. Note that an access can generate a datarace only if it accesses an escaping object or a static field. By applying escape analysis before and considering only accesses of escaping objects for datarace analysis, we can improve the results of datarace analysis.

8.1 All-Path Points-to Analysis

The all-paths points-to analysis we have implemented is a flow-insensitive, whole program analysis, similar to [7]. In the analysis, a distinct abstract object is created for each allocation site in the program. Each abstract object represents all the concrete objects created at the same site. The points-to analysis computes for each access in the program the set of abstract objects it points to along some path.

8.2 Must Points-to Information

A precise must points-to analysis is generally expensive. We have devised a simple and conservative must points-to analysis based on the notion of *single-instance* statements, each of which executes at most once during an execution. An abstact object created at a single-instance statement represents at most one concrete object. We call such an object a *single-instance* object. If an access points to only one abstract object and that abstract object is a single-instance object, then the relation between the access and the object is a must points-to relation.

We compute predicate $SingleInstance(n)$ for an ICFG node n conservatively as follows: First, we compute *strongly connected components (SCCs)* of the ICFG[6]. Then, we mark as a *multiple-instance method* every ICG method node that (1) has multiple invocation edges onto it, or (2) has at least one invocation edge onto it from a node in a SCC. Next, we set $SingleInstance(n)$ to false for ICFG node n if and only if it is in a method marked as multiple-instance method or is in an SCC. If n is a thread-root node with a unique (thread) start edge onto it, which originates in a single-instance ICFG node then we set $SingleThreadInstance(n)$ to true.

[6] We only include in an SCC nodes that are on a cyclic ICFG path.

8.3 All-Path Thread Points-To Analysis

We compute $MayThreadPT_\perp(x)$ for access x (Equation 10) as a forward dataflow problem over ICG. Given a node n in the ICFG, let $n.id$ be the unique identifier of node n in the ICFG. Also, let $threadRoot(n)$ be true if and only if n is a thread-root node. We use $x \in n$ if x is an access in ICG node n. The thread set at entry to n (denoted as TS_i^n) and the thread set at exit from n (denoted as TS_o^n) are defined by the following dataflow equations:

$$\begin{aligned} TS_o^n &= TF(n) \\ TS_i^n &= \bigcup_{p \in Pred(n)} TS_o^p \end{aligned}$$

The *transfer funcion* $TF(n)$ is defined as follows:

$$TF(n) = \begin{cases} \{n.id\} & \text{if } threadRoot(n) \\ \iota & \text{if } \neg threadRoot(n), \end{cases}$$

where ι is the identity function. Now for all $x \in n$,

$$MayThreadPT_\perp(x) = TS_o^n.$$

8.4 All-Path Synchronization Points-To Analysis

We compute $MaySync_\perp$ and $MustSync_\perp$ of Equations (12) and (13) as a forward dataflow problem over the ICG. For node $n \in ICG$, let $Synch(n) = true$ if n is a synchronized method or block, and let x_n be the access of the synchronization object if $Synch(n) = true$. Also, let $Pred(n)$ be the set of *intrathread* predecessor nodes of n.

$MaySync_\perp(y), y \in n$, can be computed by the following set of dataflow equations:

$$\begin{aligned} Gen(n) &= \begin{cases} MayPT_\perp(x_n) & \text{if } Synch(n) \\ \emptyset & \text{otherwise} \end{cases} \\ SO_o^n &= SO_i^n \cup Gen(n) \\ SO_i^n &= \bigcup_{p \in Pred(n)} SO_o^p \\ MaySync_\perp(y) &= SO_o^n, \forall y \in n \end{aligned}$$

$MustSync_\perp(y), y \in n$, can be computed by the following set of dataflow equations:

$$Gen(n) = \begin{cases} MustPT_\perp(x_n) & \text{if } Synch(n) \\ \emptyset & \text{otherwise} \end{cases}$$

Input: A multithreaded (Java) program, along with its Interthread Call Graph.
Output: A *statement conflict set* (SCS) containing pairs of statements that exhibit a datarace.
Method:
// Steps 1, 2, 3 compute the *node conflict set*, NCS.
// it contains all pairs of ICG nodes that could exhibit a datarace.
1: /* Step 1: */
2: $NCS := \emptyset$;
3: **foreach** thread-root node R1 in the ICG **do**
4: Propagate synchronization objects to callees
5: **endfor**

6: initializeVisitMarkers() ; /* to prevent visiting node more than once per traversal */

7: /* Step 2: */
8: **foreach** distinct pair of thread-root nodes (R1, R2) in the ICG **do**
9: /* The root methods of R1 and R2 must be invoked by distinct threads. */
10: $NCS := NCS \cup$ OuterTrav(R1, R2);
11: **endfor**

12: /* Step 3: */
13: **foreach**thread-root node R1 in the ICG **do**
14: **if** $\neg SingleThreadInstance(R1)$ **then**
15: /* The root method of R1 can be invoked by more than one thread in a single program execution. */
16: $NCS := NCS \cup$ OuterTrav(R1, R2);
17: **endif**
18: **endfor**

19: /* Step 4: */ // Compute SCS by enumerating pairs of conflicting statements
// within each pair of conflicting ICG nodes in NCS.

20: $SCS := \emptyset$;
21: **foreach** pair of ICG nodes (N1, N2) in NCS **do**
22: **foreach** access u1 in N1 **and** access u2 in N2 **do**
23: **if** u1 is a write operation **or** u2 is a write operation **then**
24: **if** $SameAccessObj(u1, u2)$ **then** // see Table 1
25: $SCS := SCS \cup \{(u1,u2)\}$;
26: **endif**
27: **endif**
28: **endfor**
29: **endfor**
30: **end procedure**

Figure 4: Meta-algorithm for static data race analysis

Datarace kind	$SameSyncObj(n_1, n_2)$	$SameAccessObj(u_1, u_2)$	$SameThreadRoot(n_1, n_2)$
Definite	$MayMuSync_{\perp}(n_1, n_2)$ ($Equation$ 18)	$MustConfAcc_{\perp}(u_1, u_2)$ ($Equation$ 15)	$n_1.Root = n_2.Root$
Potential	$MustMuSync_{\perp}(n_1, n_2)$ ($Equation$ 19)	$MayConfAcc_{\perp}(u_1, u_2)$ ($Equation$ 14)	false

Table 1: Table illustrating multiple accesses of the meta-algorithm in Figure 4

```
/* Returns the set of conflicting ICG node pairs reachable from n1 and R2. n.SyncSet is the set of */
/* synchronization objects for which locks must have been obtained before execution reaches ICG node n. */
1:  procedure OuterTrav(n1, R2)
2:    if (n1.outerHasBeenVisited()) then
3:      return; // already visited
4:    endif
5:    n1.outerMarkVisited(); /* mark the node to prevent revisiting during outerTrav*/

6:    ConflictSet conflictSet = InnerTrav(n1, R2);
7:    foreach callee c1 of n1 in the ICG do
8:      conflictSet = conflicSet ∪ OuterTrav(n1, R2);
9:    endfor
10:   return conflictSet;
11: end procedure

12: /* Returns the set of conflicting ICG node pairs reachable from n1 and n2. */
13: procedure InnerTrav(n1, n2)
14:   if (SameThreadRoot(n1, n2)) then // see Table 1
15:     return;
16:   endif

17:   if (n2.innerHasBeenVisited()) then
18:     return; // already visited
19:   endif
20:   n2.innerMarkVisited(); /* mark the node to prevent revisiting during innerTrav*/

21:   if SameSyncObj(n1, n2) then // see Table 1/
22:     return ∅; /* no conflict possible along this path */
23:   endif

24:   ConflictSet conflicSet = {(n1, n2)};
25:   foreach callee c2 of n2 in the ICG do
26:     conflicSet = conflicSet ∪ InnerTrav(n1, c2);
27:   endfor
28:   return conflictSet;
29: end procedure
```

Figure 5: Procedures OuterTrav and InnerTrav

$$
\begin{aligned}
SO_o^n &= SO_i^n \cup Gen(n) \\
SO_i^n &= \bigcap_{p \in Pred(n)} SO_o^p \\
MustSync_\perp(y) &= SO_o^n, \forall y \in n
\end{aligned}
$$

be computed as follows:

$$
\begin{aligned}
PotentialPar_\perp(x,y) &= \\
&\quad \neg(dom(n_x, n_y) \vee dom(n_y, n_x)) \\
DefinitePar_\perp(x,y) &= par(n_x, n_y)
\end{aligned}
$$

8.5 All-Path Thread Creation Ordering

Let $N1$ and $N2$ be two nodes in ICFG. $N1$ *dominates* $N2$, written $dom(N1, N2)$, if every path from the root to $N2$ includes $N1$. $N1$ and $N2$ are *parallel*, written $par(N1, N2)$, if no ICFG paths include both $N1$ and $N2$. Let x and y be two accesses in the program such that N_x and N_y are the corresponding nodes in the ICFG, respectively. $PotentialPar_\perp(x,y)$ and $DefinitePar_\perp(x,y)$ can

(We have not implemented these two predicates for our experimental results.)

8.6 All-Path Datarace Analysis

Figure 4 outlines our *meta-algorithm* for static data race analysis. As we will see, this same algorithmic structure can be used to compute definite and potential dataraces by selecting the precision assumed in computing the

SameSyncObj, *SameAccessObj*, and *SameThreadRoot* relations. The input for the algorithm is a multithreaded (Java) program, along with its Interthread Call Graph (ICG) representations. The output is a set of *statement pairs* that exhibit dataraces. We call this set a *Statement Conflict Set* (SCS). We describe the algorithm generically in terms of "statements" for convenience; these may be statements in the source code or instructions in the bytecode. The algorithm performs a "whole program analysis" to identify dataraces, and assumes that no dynamic class loading will occur.

The algorithm in Figure 4 has three major phases. The first step initializes the synchronization object sets of each ICG node. This step implements the all-path synchronization points-to analysis described in Section 8.4.

Next, a nested traversal is performed (in Steps 2 and 3) on the ICG to identify pairs of ICG nodes which are not mutually synchronized, and so could exhibit a datarace. The traversal in Step 2 begins with pairs of distinct thread-root nodes in the ICG, which implies that they must be executed by distinct threads. The traversal in Step 3 begins with a single thread-root node, and uses the *SingleThreadInstance* predicate/relation to determine if the thread-root node can be invoked more than once (as a new thread) in a single program execution. Both traversals accumulate node-pairs in the *node conflict set*, NCS. For each pair of ICG nodes in NCS, the object references within each node are examined (in Step 4) to determine if they represent a datarace or not by using the *SameAccessObj* relation. (We use a pair of markers to prevent visiting an ICG node more than once during outerTrav, and visiting an ICG node more than once during innerTrav for a given outerTrav. The algorithm does not show the details of these parts.)

Figure 5 outlines the algorithms for procedures *OuterTrav* and *InnerTrav*, which are used to perform the nested traversals of the ICG in Steps 2 and 3 of Figure 4. The algorithms traverse the ICG in a simple depth-first order and compute the set of "conflicting" ICG node pairs. Two ICG node $N1$ and $N2$ conflict each other if (1) there exist a path π from a thread-start node $R1$ to $N1$, and a path π' from another thread-start node $R2$; and (2) path π and π' do not have a common synchronization object. Since multiple visits to an ICG node in a path do not provide any more synchronization objects than a single visit, a simple depth-first visit of ICG nodes is sufficient.

Table 1 shows how the meta-algorithm can be used for different purposes. The entries in the second and the third columns (*SameSyncObj* and *SameAccessObj*) come straightforwardly from the definitions of definite and potential dataraces. The equation numbers identify the corresponding equations in Section 7. Note that Equation 16 and Equation 17 are implicitly implemented by Step 2 and Step 3 in the algorithm.

9 Experimental Results

We have completed a preliminary implementation of the static datarace analysis algorithms described in this paper in the Jalapeno optimizing compiler [8]. Since this compiler does not currently perform any points-to analysis, we implemented conservative may and must points-to alias analyses to use as building blocks for datarace analysis. Note that conservative analysis leads to larger may points-to sets and smaller must points-sets, compared to a more precise analysis. The may-alias implementation is a hybrid of two flow-insensitive points-to algorithms described in previous work [7, 41]. The must-alias implementation is based on the *SingleInstance* predicate defined in Section 8 for ICFG nodes: a points-to set is a must points-to set if it is a singleton set of an abstract object created at a single-instance statement. Our results also use an escape analysis implementation based on the approach described in [14].

To test the effectiveness of our approach, we analyzed the results we obtained by running our implementation on eight multi-threaded Java programs. Seven of them are small programs borrowed from the authors of [32]. The eighth is a larger program, `mtrt`. It is the only multithreaded benchmark in the specJVM98 suite. `Phil` is a Java implementation of the dining philosophers problem. `BridgeTest` and `OneCarBridgeTest` are bridge operation simulations originally taken from [1]. `Chiron` is a Java simulation of the Chiron user interface framework originally written in Ada at UC Irvine. The remaining benchmarks were originally taken from Doug Lea's book on Java concurrency [25] and its online supplement [24].

Table 2 shows the names and (static) characteristics of the benchmark programs we used. The second column shows the number of static accesses (*e.g.,* `getstatic`, `putstatic`) encountered in each program. The third column shows the number of heap accesses (*e.g.,* `getfield`, `putfield`). The fourth column shows the number of heap accesses of escaping objects, as computed by our escape analysis. The escaping accesses are a subset of the heap accesses. It is encouraging to note that the largest fraction of *non-escaping* heap accesses (780 out of 990 accesses) was encountered in the largest program, `mtrt`.

Table 3 presents the results of running our analysis, in *number of statement pairs* that can generate dataraces. The second and the third columns show the results for definite datarace, the first of which shows the results of points-to analysis only and the second of which shows the results of both points-to analysis and escape analysis. The large number of zeroes in these columns is disappointing, and indicate that more precise points-to analyses (especially must points-to analysis of accessed objects) is necessary for identifying definite dataraces.

The remaining columns show the results for potential

	number of static accesses	number of heap accesses	number of escaping heap accesses
MessagePrinter	22	7	7
PictureRenderer	21	3	3
PrintService	14	11	11
OneCarBridgeTest	5	17	11
BridgeTest	6	28	22
Phil	45	9	9
Chiron	13	48	43
mtrt	248	990	210

Table 2: Benchmark programs and their characteristics

	definite datarace		*potential datarace*			
benchmark programs	points-to only	points-to & escape	points-to & escape	points-to only	escape only	no analysis
MessagePrinter	0	0	0	0	$29 \times 30/2$	$29 \times 30/2$
PictureRenderer	0	0	0	0	$24 \times 25/2$	$24 \times 25/2$
PrintService	0	0	0	0	$25 \times 26/2$	$25 \times 26/2$
OneCarBridgeTest	0	0	4	4	$16 \times 17/2$	$22 \times 23/2$
BridgeTest	0	0	9	13	$28 \times 29/2$	$34 \times 35/2$
Phil	6	6	30	30	$54 \times 55/2$	$54 \times 55/2$
Chiron	0	0	27	31	$56 \times 57/2$	$61 \times 62/2$
mtrt	0	0	587	3030	$458 \times 459/2$	$1238 \times 1239/2$

Table 3: Datarace pairs of statements found by our analyses

dataraces. The fourth column shows the results of points-to analysis and escape analysis, the fifth shows the results of points-to analysis only, the sixth is for escape analysis only, and the last shows the total number of pairs of statements that access a static or instance field. The number of pairs should be montonically non-decreasing as we read each row from left to right. (Recall that a larger size indicates better precision for definite dataraces, and a smaller size indicates better precision for potential dataraces.) For potential dataraces, the results show that our analysis can yield significant improvements compared to examination of all accesses or all accesses that escape. Further, combining escape anlysis with points-to analysis yields notable improvements for potential dataraces, especially for `mtrt` where the number of statement pairs was reduced from 3030 to 587.

Our preliminary results show more promise for potential dataraces than definite dataraces. We expect further improvements after implementing more precise points-to algorithms. However, these results show that even our preliminary implementation can be effective in narrowing down the set of statement pairs that may particpate in a datarace. This information could be useful for use in a static analysis tool, or as a filter for reducing the overhead of dynamic datarace detection.

10 Related Work

Flanagan and Freund's static datarace detection tool for Java [20] is based on a type system, and employs type-based equivalence of lock variables: two locks of identical types are regarded as the same lock. This is similar to type-based alias analysis, and can result in overly optimistic results in datarace analysis. Their work relies on user annotations for its precision when the type system proves too restrictive. The authors claim that the annotation burden is not excessive, typically requiring fewer than 20 annotations per 1000 lines of code. Their system successfully found a few dataraces in well tested and relatively mature Java programs. Warlock is an annotation-based static datarace detection tool for ANSI C programs [42], which also supports lock-based synchronization. It also assumes without verification that user annotations on thread-local accesses are correct. It has been used successfully to catch dataraces in several programs.

Aiken and Gay's work statically detects dataraces in SPMD programs [3]. Since SPMD programs employ barrier-style synchronizations, they need not track locks held at each statement.

Eraser is similar to our approach in that it is based on lock-based synchronization in determining whether two statement executions are concurrent or not [39].[7] However, it is a dynamic tool, and as such suffers from insufficient test coverage inherent in any purely dynamic tool: it only detects dataraces that actually occur in an execution. It does not consider thread ordering due to thread creation or termination in determining whether two statement executions can be concurrent. Eraser works independently of the input source language by instrumenting binary code.

Praun and Gross improves on Eraser by applying escape analysis to filter out the statements that cannot generate dataraces [35]. Their work specifically detects dataraces in accessing Java objects. Their work for detecting dynamic dataraces will benefit from our datarace analysis because, as shown in Section 9, combining both points-to analysis and escape analysis is more effective than escape analysis alone for identifying statements that cannot generate dataraces.

Most dynamic datarace detection techniques for SPMD programs work either as a post-mortem tool or as an on-the-fly tool [15, 21, 27, 19, 34], by collecting information from actual executions with software instrumentation. The information is analyzed off-line in post-mortem analysis and on-line in on-the-fly analysis. Min and Choi's hardware-based scheme [29] uses the cache coherence protocol, and Richards and Larus' work [37] uses the *Distributed Shared-Memory (DSM)* computer's memory coherence protocol, respectively, in collecting information for on-the-fly datarace detection.

Netzer and Miller categorize dynamic dataraces into *actual*, *apparent*, and *feasible* dataraces [33]. The event pair of `MainThread.S18` and `T1.S20` in Figure 2-(B) is both an actual and an apparent datarace. It is, however, only an apparent datarace in the execution instance of Figure 2-(A) since there exists a happened-before relation between the two. A feasible datarace is an apparent datarace that can happen even when control and data dependences of the program are considered. Choi and Min describe how to identify and reproduce the *race frontier*, which is the set of dataraces not affected by any other dataraces [16]. By repeatedly reproducing and correcting the dataraces in the race frontier, one can identify all the dataraces that occur in executions.

Previous work on static concurrency analysis attempted to compute the set of statement pairs that can never happen in parallel (*nhp*) or that may happen in parallel (*mhp*) for low-level semaphore style synchronizations [11], for rendezvous style synchronizations of Ada [44, 30, 26], or Java [31]. Note that mhp, which is a compliment of nhp, is equivalent to our $PotentialPar_{\perp}(x, y)$ in Section 8.5. Computing the con-

[7] There exists a compile-time tool, also named Eraser, that uses program analysis to optimize instrumentation for dynamic data race detection [28].

currency in the presence of Java's synchronization requires identifying aliasing of synchronization objects. It is not clear how aliasing of synchronization objects is considered in the work by Naumovich et. al [31].

Guava is a dialect of Java that statically disallows dataraces by preventing concurrent accesses to shared data [4]. Only instances of classes belonging to the *class category* called *monitor* can be shared by multiple threads. By serializing all accesses to fields or methods of the same shared data, Guava can prevent dataraces.

11 Conclusions

In summary, the major contributions of this paper are as follows: (1) a novel framework for static datarace analysis based on points-to analysis of multithreaded object-oriented programs; (2) formulation of path-specific and all-paths static datarace analysis problems based on path-specific and all-paths formulation of points-to analysis; (3) correctness proofs of the path-specific and all-path static datararce analyses; (4) an algorithm for efficient all-paths datarace analysis; and (5) a prototype implementation of the algorithm, and preliminary experimental results showing its effectiveness for a small set of multithreaded programs.

The results are obtained with a very conservative, flow-insensitive points-to analysis, However, they still show the effectiveness of our approach. We expect better results with more precise, albeit more expensive, points-to analyses such as flow-sensitive and context-sensitive interprocedural analyses. Considering that dataraces make multithreaded-program development notoriously difficult, static datarace analysis can become a "killer application" of precise but expensive points-to analyses.

We plan to use static datarace analysis to enhance various tools built on the *DejaVu* infrastructure for record and replay of general multithreaded programs. For example, the knowledge that a memory access does not participate in a datarace can help reduce the cost of dynamic datarace detection, and also the cost of record and replay of multithreaded applications on shared-memory multiprocessors [17, 22, 12].

Finally, we plan to extend our framework to incorporate join operations, and fully implement event ordering due to thread creation and termination. We also plan to extend our work by using techniques such as extant analysis [40] to enable datarace analysis to be performed in the context of dynamic class loading as well.

References

[1] http://vislab-www.nps.navy.mil/java/course/sourcecode.

[2] Java Development Kit 1.1. Technical report, Sun Microsystems.

[3] A. Aiken and D. Gay. Barrier interference. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*, pages 342–354, January 1998.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of java without data races. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[5] B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[6] J. Bodga and U. Hölzle. Removing unnecessay synchronization in Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[7] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *7th International Workshop on Languages and Compilers for Parallel Computing*, 1994. Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

[8] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[9] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Java Grande*, pages 129–141, 1999.

[10] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, July 1988. *SIGPLAN Notices, 23(7).*

[11] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Conference Record of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1989.

[12] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th IEEE International Parallel & Distributed Processing Symposium*, April 2001.

[13] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, Jan. 1993.

[14] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[15] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[16] J.-D. Choi and S. L. Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[17] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, August 1998.

[18] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE'92 International Conference on Computer Languages*, Apr. 1992.

[19] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Seattle, Washington, Mar. 1990. ACM Press.

[20] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, June 2000.

[21] Y.-K. Jun and K. Koh. On-the-fly detection of access anomalies in nested parallel loops. *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging (Also available as ACM SIGPLAN Notices Vol. 28, No. 12)*, pages 107–117, May 1993.

[22] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, May 2000.

[23] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 93–108, Jan. 1991.

[24] D. Lea. Concurrent programming in java: Design principles and patterns, online supplement. http://gee.cs.oswego.edu/dl/cpj/index.html.

[25] D. Lea. *Concurrent Programming in Java.* Addison-Wesley, November 1999.

[26] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on on Principles and Practices of Parallel Programming*, pages 129–138, May 1993.

[27] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91*, pages 24–33, November 1991.

[28] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging (Also available as ACM SIGPLAN Notices Vol. 28, No. 12)*, pages 129–139, May 1993.

[29] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[30] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering (FSE)*, pages 24–34, November 1998.

[31] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. Technical report 98-44, Department of Computer Science, University of Massachusetts at Amherst, 1998.

[32] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. In *Proceedings of the Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 338–354, September 1999.

[33] R. H. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[34] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 133–144, April 1991.

[35] C. v. Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[36] W. Pugh. Fixing the Java memory model. In *ACM 1999 Java Grande Conference*, pages 89–98, June 1999.

[37] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 40–47, August 1998.

[38] E. Ruf. Effective synchronzation removal for Java. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

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

[40] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[41] B. Steensgaard. Points-to analysis in almost linear time. In *In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41, January 1996.

[42] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[43] J. M. Stone. Debugging concurrent processes: A case study. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 14–153, June 1988.

[44] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.