Dataflow Analysis for Datarace-Free Programs

Arnab De Deepak D'Souza Rupesh Nasre

IISc-CSA-TR-2010-8
http://archive.csa.iisc.ernet.in/TR/2010/8/

Computer Science and Automation
Indian Institute of Science, India

December 2010

Dataflow Analysis for Datarace-Free Programs

Arnab De[*] Deepak D'Souza[†] Rupesh Nasre[‡]

Abstract

Memory models for shared-memory concurrent programming languages typically guarantee sequential consistency (SC) semantics for datarace-free (DRF) programs, while providing very weak or no guarantees for non-DRF programs. In effect programmers are expected to write only DRF programs, which are then executed with SC semantics. With this in mind, we propose a novel scalable solution for dataflow analysis of concurrent programs, which is proved to be sound for DRF programs with SC semantics. We use the synchronization structure of the program to propagate dataflow information among threads without requiring to consider all interleavings explicitly. Given a dataflow analysis that is sound for sequential programs and meets certain criteria, our technique automatically converts it to an analysis for concurrent programs.

1 Introduction

In recent years several new semantics based on relaxed memory models have been proposed for concurrent programs, most notably the Java Memory Model [21], and the C++ Memory Model [3]. While the aim of the relaxed semantics is to facilitate aggressive compiler optimizations and efficient execution on hardware, the semantics they provide can be quite different from the standard "Sequentially Consistent" (SC) semantics. A common guarantee that they typically provide however is that programs *without* dataraces will run with SC semantics. For programs *with* dataraces there are very weak guarantees: the Java Memory Model [21] essentially ensures that there will be no "out-of-thin-air" values read, while the C++ memory model specifies no semantics [3] for such programs.

[*] arnabde@csa.iisc.ernet.in

[†] deepakd@csa.iisc.ernet.in

[‡] nasre@csa.iisc.ernet.in

The prevalence of this so-called "SC-for-DRF" semantics makes the class of datarace-free programs with sequentially consistent semantics, an important one from a static analysis point of view. An analysis technique that is sound for this class of programs can in principle be used by a compiler-writer for the *general* class of programs, as long as the ensuing transformation preserves the weak guarantees described above. From a verification point of view as well, the class of racy programs is unlikely to require sophisticated analysis due to the loose semantics for this class of programs, while a sound analysis for datarace-free programs can be used to prove non-trivial properties for the class of datarace-free programs.

With this in mind in this paper we propose a novel and scalable dataflow analysis technique for concurrent programs that is sound for datarace-free programs under the SC semantics. Given a sequential dataflow analysis that meets certain criteria, our technique automatically produces an efficient and fairly precise analysis for concurrent programs. The criteria that the underlying analysis must meet is that each dataflow fact should be dependent on the contents of some associated lvalues (an *lvalue* is an expression that refers to some memory locations at runtime). Several sequential dataflow analyses such as null-pointer analysis, interval analysis, and constant propagation satisfy this criteria. Our technique gives useful information (in terms of precision of the inferred data-flow facts) at points where the corresponding lvalue is *read*. For example, in the case of null-pointer analysis, the dataflow fact "*NonNull*(`p`)" is dependent on the contents of the lvalue "`p`" and is relevant before a statement that dereferences (reads) "`p`". Similarly, the fact that an lvalue has a constant value at a program point is dependent on the contents of the lvalue and is relevant at statements that read that lvalue.

The main challenge in lifting an analysis for sequential programs to concurrent programs is that multiple threads can simultaneously modify a shared memory location. Traditionally the analysis techniques for concurrent programs address this problem in one of the following ways: they either invalidate the analyzed fact if there is any possible interference from any other thread [4, 16], making the analysis very imprecise, or they exhaustively explore all possible interleavings [28], leading to poor scalability. In contrast, our analysis technique uses the synchronization structure of the program to propagate dataflow facts between threads. The main insight we use is that it is sufficient to propagate dataflow facts between threads only at corresponding synchronization points (like from an "`unlock(l)`" statement to a "`lock(l)` statement"). We also show how our framework can be integrated with a context-sensitive analysis.

We have implemented our technique in a framework for automatically converting dataflow analyses for sequential Java programs to sound analyses for concurrent programs and instantiated it for a null-reference analysis. Our initial experience with the tool shows that the analysis runs in a few seconds on real benchmark programs, and is able to prove a high percentage of dereferences to be safe. We have also developed a prototype implementation for concurrent C programs which use the pthreads library [12]. This allows us to compare our technique empirically with the state-of-the-art Radar tool [4], and show that our tool is more precise on a few medium-sized benchmarks.

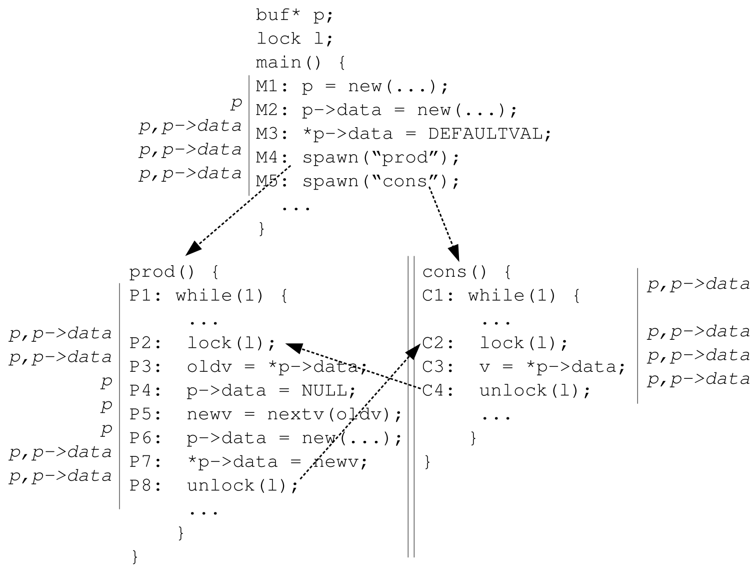
2 Overview of Our Approach

In this section we informally illustrate our technique with the help of a few examples. We consider the null-pointer analysis where the goal is to compute a set of dataflow facts for each edge of the program which tell us which lvalues are non-null along all executions reaching that edge. Examples of such dataflow facts can be *NonNull*(`p->data`) for the program in Figure 1.

Note that value of the dataflow fact *NonNull*(`p->data`) at runtime depends on the contents of the memory location corresponding to the lvalue `p->data`. Hence, at runtime, the value of this fact can only be modified by writing to the memory locations corresponding to `p->data` or `p`, possibly through some alias. Moreover, the value of the fact *NonNull*(`p->data`) is relevant only before the statements where `p->data` is dereferenced or `p->data` is assigned to some other pointer or `p->data` is compared to `NULL`. For example, in Figure 1, this fact is relevant before the statements `M3`, `P3`, `P7` and `C3`, but not before `P6` or `M2`. Note that at all edges where this fact is relevant, the successor statements read `p->data`. Our analysis guarantees that for a given datarace-free program, if a fact is computed to be true at a program edge where the fact is relevant, then it is indeed true at that program edge in all executions of the program.

Figure 1 shows a simple concurrent producer-consumer program, where data is shared through a shared location, pointed to by `p`. The call to `new` returns newly allocated memory. Note that, the `main` thread sets the pointers `p` and `p->data` to non-null values. The `prod` thread sets `p->data` to null after locking `l`, but restores its non-nullity before unlocking `l`. As a result, the `cons` thread can dereference `p` and `p->data` without checking for non-nullity after locking `l`. This code has no null-pointer dereferences in any of its executions. Clearly, the threads in this code depend on each other to make the pointers non-null before any other thread can access them. We also note the the program has no data-races.

Let us again consider the dataflow fact *NonNull*(`p->data`) in the program of Figure 1. As the program is datarace-free, if a thread writes to `p->data` or



```
                              buf* p;
                              lock l;
                              main() {
                              M1: p = new(...);
                           p  M2: p->data = new(...);
                   p,p->data  M3: *p->data = DEFAULTVAL;
                   p,p->data  M4: spawn("prod");
                   p,p->data  M5: spawn("cons");
                                ...
                              }

               prod() {                          cons() {
               P1: while(1) {                    C1: while(1) {                p,p->data
                     ...                               ...
   p,p->data   P2:   lock(l);                    C2:   lock(l);                p,p->data
   p,p->data   P3:   oldv = *p->data;            C3:   v = *p->data;           p,p->data
           p   P4:   p->data = NULL;             C4:   unlock(l);              p,p->data
           p   P5:   newv = nextv(oldv);               ...
           p   P6:   p->data = new(...);             }
   p,p->data   P7:   *p->data = newv;            }
   p,p->data   P8:   unlock(l);
                     ...
                   }
               }
```

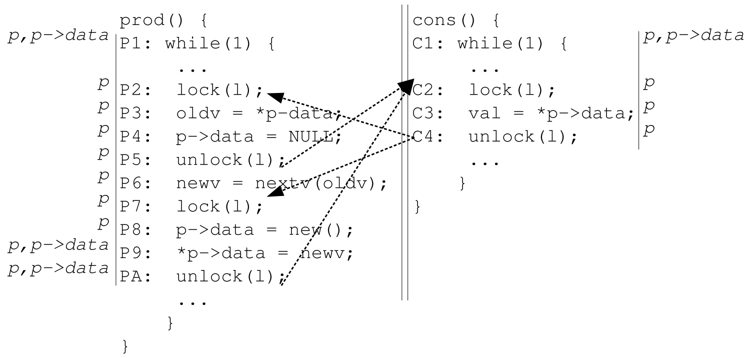
Figure 1: Program 1

p and some other thread reads p->data later in the execution, then these accesses must be synchronized, i.e. there must be a release action (e.g. unlock or spawn) by the first thread, followed by an acquire action (e.g. lock or first action of a thread) by the second thread, between the write and the read. In other words, in any execution of the program, the action that modifies the dataflow fact and the action before which it is relevant either belong to the same thread or are synchronized.

As the first step of our analysis, we introduce new edges between nodes of the control-flow graphs (CFGs) representing different threads. These edges correspond to possible "release-acquire" pairs at runtime. We refer to this unified set of CFGs with added edges as the *sync-CFG*. Figure 1 shows the edges we add for this program as dashed arrows — from spawn to the first instruction of the child thread and from the unlock to lock statements if they access same lock variable and if they can possibly run in parallel.

In the next step of our analysis, we perform a sequential dataflow analysis on this *sync-CFG* to compute a set of dataflow facts at each program edge that conservatively approximates the *join-over-all-paths* (JOP) solution over the *sync-CFG*.

In Figure 1, we show the lvalues discovered to be non-null by our analysis at different program points in *italics*. As p->data is non-null at point M5 in the main thread before spawning the cons thread, this fact gets propagated to the first instruction C1 of the cons thread though one of the added edges,



```
             prod() {                              cons() {
p,p->data   |P1: while(1) {                        C1: while(1) {         |p,p->data
                  ...                                     ...
          p |P2:  lock(l);                         C2:  lock(l);          |p
          p |P3:  oldv = *p-data;                  C3:  val = *p->data;   |p
          p |P4:  p->data = NULL;                  C4:  unlock(l);        |p
          p |P5:  unlock(l);                              ...
          p |P6:  newv = nextv(oldv);                  }
          p |P7:  lock(l);                         }
          p |P8:  p->data = new();
p,p->data   |P9:  *p->data = newv;
p,p->data   |PA:  unlock(l);
                  ...
                }
             }
```

Figure 2: Program 2

and from there to the lock instruction at C2. Similarly, although p->data is set to null in the prod thread at P4, it is set back to non-null at P6 before the unlock. This facts also gets propagated to the lock statement of the cons thread through the edge P8 to C2. As p->data is non-null in both the paths joining at the C2 of the cons thread, we can determine p->data to be non-null before the lock statement in all executions. This makes the fact *NonNull*(p->data) to be true before the deference of p->data at C3.

The reason why our analysis works is that if, in an execution, an action modifies the dataflow fact *NonNull*(p->data) and it is relevant at some later action, then there exists a static path from the statement of the first action to the statement of the second action in the *sync-CFG* and the static dataflow function corresponding to this path will conservatively approximate the effect of the execution path segment from the first action to the second action on the dataflow fact. As an example, consider the interleaved execution path fragment [P6, C1, P7, P8, C2, C3] where P6 modifies *NonNull*(p->data) and it is relevant at C3. There is a static path in the *sync-CFG* [P6, P7, P8, C2, C3] which has the same effect on this dataflow fact as the execution path segment.

We note that at points where a fact is *not* relevant our analysis may compute incorrect values. For example our analysis computes *NonNull*(p->data) to be true at C1 although the interleaved execution path segment [P4,C1] can make it false. However, the fact *NonNull*(p->data) is not relevant at C1.

Let us now consider a buggy version of the program, presented in Figure 2. The main thread is the same as Figure 1. This program is also DRF, but the prod thread releases the lock after setting p->data to null at P4, and acquires the lock again before setting it to non-null. If the cons thread dereferences p->data in between these two actions, it will dereference a null-pointer. For example, the execution path segment [`P4`, `P5`, `C2`, `C3`] will result in null-pointer dereference. Note that there is a static path [`P4`, `P5`, `C2`, `C3`] in the *sync-CFG* that also sets the fact *NonNull*(`p->data`) to false before `C3`. Hence our analysis will detect that `p->data` can be null before the dereference at `C3`. Note that here also we incorrectly compute *NonNull*(`p->data`) to be true at `C1` as the modification of this fact at `P4` is not propagated to `C1`. Nevertheless, as the program is datarace-free, before the `cons` thread reads `p->data`, it must synchronize with the `prod` thread and the modified value for the fact *NonNull*(`p->data`) is propagated to the `cons` thread through the corresponding static edge ([`P5`, `C2`] in this case).

3 Related Work

There are quite a few works on dataflow analysis of concurrent programs in the literature and they differ considerably in terms of technique, precision and applicability. Some works [17, 11, 6] create parallel flow graphs similar to our technique and perform a modified version of sequential analysis on them, but unlike us, their techniques are applicable to very specific analyses, such as bit-vector analysis or gen-kill analysis. In particular, they do not handle the analyses where the value of a dataflow fact can depend on some other dataflow fact. For example, in null-pointer analysis, `p` is non-null after a statement `p = q` only if `q` is non-null before the statement. Unlike our technique, they also do not consider many features of modern concurrent programs such as unbounded threads, synchronization using locks/volatiles etc. For example, the pointer-analysis algorithm presented in [24] considers only structured par-begin/par-end like synchronization constructs.

On the other hand, there are a few works such as [16] that kill the dataflow facts whenever there is a possible interference. Similarly, Radar [4] uses a datarace detection engine to conservatively kill a dataflow fact whenever there is a possible race on the lvalues corresponding to the fact. Our technique is more precise than theirs as we propagate the dataflow facts precisely. For example, in Figure 1, Radar cannot detect the dereference of `p->data` in the `cons` thread to be safe. Recently Farzan et al [7] presented a compositional technique for dataflow analysis, but it is applicable to only bit-vector analyses.

Model checkers such as [28] provide an alternative technique to find if a property holds at a particular program point. They typically exhaustively enumerate all interleavings of a program, resulting in poor scalability. CHESS [22] prunes the number of interleavings by context switching only at the synchronization points, assuming the program is datarace-free, but scalability still remains an issue. In contrast ours is a static analysis which does not explore interleavings explicitly. Moreover, due to infinite state-spaces, model checking of real programming languages cannot cover all program behaviors. Thread modular analyses [8, 9, 10] can analyze each thread separately, but either require user-defined annotations denoting some invariants or try to infer them automatically, limiting their scalability and precision. Recently, Malkis et al. [20] proposed a thread-modular abstraction refinement technique where the set of reachable "global states" is computed as the cartesian abstraction of sets of reachable "local" states. If a global state is infeasible, an abstraction refinement step excludes it from the cartesian abstraction. This technique assumes the number of dynamic threds to be statically bound. It is not implemented for real programs and the analysis-refinement cycle limits its scalability.

4 Preliminaries

4.1 Program Structure

In this section we formalize the structure of the subject programs for our analysis. For ease of presentation, we use a simple core language that has the representative features of real programming languages with shared-memory concurrency.

The program is composed of a finite number of named thread codes[1], one of which is designated as the *main thread.* The program is denoted as $P = (T_0, \ldots, T_k)$, where each T_i is name of a static thread. Each thread T_i is represented as a control flow graph (CFG) C_i where each node represents a statement in the program. We do not consider procedures at this point (context-sensitive inter-procedural analysis is described in Section 8). In the rest of the paper, we use the terms *nodes* and *statements* interchangeably to refer to the static statements in the program.

Figure 3 defines the syntax of the language partially. Variables are declared globally. The non-terminal *Decl* in Figure 3 describes a variable declaration. A regular (non-synchronization) variable can be of some basic type or structure type or pointer type. A *synchronization variable* is either a lock or a thread identifier.

Statements (*Stmt* in Figure 3) are of following types: *assignment*, *branch*, *synchronization* and *skip*. Assignment statements (*AsgnStmt* in Figure 3) assigns the value of an expression to an lvalue, which is either a declared

[1] We refer the code of a thread as a *static thread* and the runtime instance of a thread as a *dynamic thread.*

```
Decl      ::=  VarType <var> | Lock <lockvar>
               | ThreadId <tid>
VarType   ::=  BasicType | VarType*

Stmt      ::=  AsgnStmt | BranchStmt | SyncStmt
               | skip
AsgnStmt  ::=  Lval := Expr
Lval      ::=  <var> | *Lval
SyncStmt  ::=  lock <lockvar> | unlock <lockvar>
               | <tid> := spawn <T> | join <tid>
               | start | end
```

Figure 3: Partial syntax of the language

variable or dereference of an lvalue. Expressions are arithmetic or logical expressions over constants and lvalues or “address of” expressions. Branch conditions can be any Boolean expression.

For an lvalue l, we define $\mathit{deref}(l)$ to be the set of lvalues that are dereferenced in the expression of l. Formally,

$$\mathit{deref}(l) = \begin{cases} \{l'\} \cup \mathit{deref}(l') & \text{if } l \text{ is of the form } *l' \\ \emptyset & \text{otherwise} \end{cases}$$

For example, if `p` is a variable and `**p` is an lvalue, then $\mathit{deref}(**\texttt{p}) = \{\texttt{p}, *\texttt{p}\}$.

We call an lvalue l *relevant* at a program edge E and the node in $\mathit{nsucc}(E)$ if l is syntactically part of the expression read at the node $\mathit{nsucc}(E)$. Note that, if l is relevant at a program edge/node, all lvalues in $\mathit{deref}(l)$ are also relevant at that program edge. In the program of Figure 1, at `C3`, the relevant lvalues are `p`, `p->data` and `*p->data`. We consider only well-typed programs without pointer arithmetic.

Synchronization statements (*SyncStmt* in Figure 3) are of special interest to us. Each thread has a *start* node and an *end* node, containing special `start` and `end` statements, respectively. Threads are spawned by `spawn` statements that take static thread names as parameters and return thread ids of the child threads. A parent thread waits for a child thread to finish using a `join` statement. The `lock` and `unlock` statements have the standard semantics for reentrant locks. Only synchronization statements can access synchronization variables. Although we consider only these synchronization statements in this paper, our technique can be applied to programming languages with other synchronization statements that have acquire/release semantics (described in Section 4.2), such as read/write of volatiles in the Java programming language [13].

For a CFG $C = (\mathit{Nodes}, \mathit{Edges}, E_0, E_\sharp)$, *Nodes* denotes the set of nodes, $\mathit{Edges} \subseteq \mathit{Nodes} \times \mathit{Nodes}$ denotes the set of edges, $E_0 \notin \mathit{Edges}$ denotes a special *start edge* with no predecessor node and $E_\sharp \notin \mathit{Edges}$ denotes a special *end edge* with no successor node in C. For a node N, $\mathit{epred}(N) = \{E \in \mathit{Edges} \mid \exists N' \in \mathit{Nodes} : E = \langle N', N\rangle\}$ denotes the set of predecessor edges of N and $\mathit{npred}(N) = \{N' \in \mathit{Nodes} \mid \langle N', N\rangle \in \mathit{Edges}\}$ denotes the set of predecessor nodes of N. For an edge $E = \langle N, N'\rangle$, $\{N\}$ is the singleton set of predecessor node of E, denoted by $\mathit{npred}(E)$ and the set $\mathit{epred}(\mathit{npred}(E))$ is the set of predecessor edges of E, denoted by $\mathit{epred}(E)$. Similarly, *esucc* and *nsucc* denote the sets of successor edges and successor nodes for an edge or a node, respectively. Although we overload these notations, the meaning should be clear from the context. Each CFG has a *start node* N_0 which is the successor node of E_0 and an *end node* $N_\sharp$ which is the predecessor node of $E_\sharp$. Let N_0^M and E_0^M denote the start node and the start edge of the main thread and $N_\sharp^M$ and $E_\sharp^M$ denote the end node and the end edge of the main thread, respectively.

A *path* Π in a CFG C is defined as a sequence of nodes $\langle N'_0, \ldots, N'_n\rangle$ of C, such that there is an edge in C between N'_i and N'_{i+1} for every i, $0 \leq i < n$. A path Π is called an *initial path* in C if the first node of the path is the node N_0, the start node in C.

4.2 Execution

Let P be a program written in the language described in Section 4.1. An *action* is a dynamic instance of a statement in an execution. For an action a, $\mathit{stmt}(a)$ denotes the corresponding static statement or node and $\mathit{thread\_id}(a)$ denotes the dynamic thread id of the thread performing the action.

An *interleaving* of P is a sequence of actions $\langle a_0, \ldots, a_n\rangle$, $\mathit{stmt}(a_0) = N_0^M$, possibly from different dynamic threads, such that the projection of the sequence to any thread id is consistent with the sequential semantics of that thread, given the values of reads of shared variables. If I is an interleaving of P, $I[i]$ denotes the *ith* action in the interleaving. Let a be an action in an interleaving I. By $\mathit{eprev}(a)$ and $\mathit{enext}(a)$ we denote the program point (CFG edge) reached in the thread executing a just before and after executing a, respectively. Similarly, by $\mathit{next}(a)$ we mean the next action in I that belongs to the same dynamic thread as a. Thus, $\mathit{next}(a) = I[j]$ if $a = I[i]$ and $\mathit{thread\_id}(a_i) = \mathit{thread\_id}(a_j)$, $i < j$ and there is no k, $i < k < j$ such that $\mathit{thread\_id}(a_i) = \mathit{thread\_id}(a_k)$. If $a' = \mathit{next}(a)$, then we say $a = \mathit{prev}(a')$.

Synchronization actions are of two types: `spawn`, `end` and `unlock` actions are the *release* actions, where as `join`, `start` and `lock` actions are the *acquire* actions.

An interleaving I of program P is *synchronization-valid* if

- Each `unlock` action is preceded by a *matching* `lock` action. For every prefix of I, number of unlock actions on a lock variable by a dynamic thread must be less than or equal to the number of lock actions performed by the same dynamic thread on the same lock.

- Locks maintain mutual exclusion property. If a is a `lock` action performed by a dynamic thread t on a lock `l`, then for any thread $t' \neq t$, the number of `unlock` actions performed on `l` by t' before a in I must be exactly equal to the number of `lock` actions on `l` by t' before a in I.

- The `start` action of any thread (except the `main` thread) is preceded by a corresponding `spawn` action that returns a thread id which is the same as the started thread.

- Each `join` action is preceded by the `end` action of the thread it waits for.

An interleaving is *sequentially consistent* (SC) if every read of a memory location reads the value written by the last preceding write to the same memory location in the interleaving. We assume that there is an initial write to every memory location whenever the memory is allocated in an execution.

An *sc-execution* is simply a synchronization-valid and sequentially consistent interleaving.

4.3 Datarace-free Programs

Two non-synchronization actions in an sc-execution are *conflicting* if they both access a common memory location and at least one of them writes to that memory location.

Given an sc-execution $\mathcal{E}$ of a program P, we say a release action *synchronizes-with* subsequent acquire actions corresponding to it. More specifically, an `unlock` action synchronizes with any subsequent `lock` action on the same lock variable, a `spawn` action synchronizes with the `start` action of the thread it spawns and an `end` action synchronizes with the `join` action that waits for the thread to finish. If in $\mathcal{E}$, an action a synchronizes with an action b, it is denoted by $a <^{\mathcal{E}}_{sw} b$.

Similarly, if in an sc-execution $\mathcal{E}$, $a = \mathcal{E}[i]$ and $b = \mathcal{E}[j]$ are two actions such that $\mathit{thread\_id}(a) = \mathit{thread\_id}(b)$, $i < j$ and there is no k, $i < k < j$, such that $\mathit{thread\_id}(\mathcal{E}[k]) = \mathit{thread\_id}(\mathcal{E}[i])$, then there is a *program-order* relation between a and b, denoted by $a <_{po}^{\mathcal{E}} b$. Note that if $a <_{po}^{\mathcal{E}} b$, then there is an edge from $\mathit{stmt}(a)$ to $\mathit{stmt}(b)$ in the CFG of the corresponding static thread.

The *happens-before* order induced by an sc-execution $\mathcal{E}$, is a partial-order on the actions of $\mathcal{E}$, denoted by $\leq_{hb}^{\mathcal{E}}$, and is defined as the reflexive transitive closure of $<_{sw}^{\mathcal{E}}$ and $<_{po}^{\mathcal{E}}$ relations.

An sc-execution $\mathcal{E}$ is *datarace-free* if every pair of conflicting actions are related by the happens-before order. A program is datarace-free if all sc-executions of the program are datarace-free. This definition of datarace-freedom is equivalent to the more intuitive definition [25] — in any sc-execution of a datarace-free program, two conflicting actions from different dynamic threads cannot happen immediately after one another.

Many programming languages such as Java [21] and C++ [3] and threading libraries such as pthreads [2], guarantee that any execution of a datarace-free program in these languages is equivalent to some sc-execution. We assume that the memory model of our language guarantees sequentially consistent semantics for datarace-free programs and we are only interested in datarace-free programs in this paper. Henceforth we refer to an sc-execution simply as an *execution*.

5 Analysis for Sequential Programs

In this section, we characterize the class of the analyses for sequential programs that can be converted to analyses for concurrent programs using our technique. This class essentially consists of the "value set analysis" (Section 5.1) and any consistent abstraction (Section 5.2) of it.

We assume the sequential program to consist of a single `main` thread. It may not have any synchronization statement except for the `start` and `end` statements of the main thread. Let us denote the sequential program by P and its CFG by $C = (\mathit{Nodes}, \mathit{Edges}, E_0, E_\sharp)$.

5.1 Value Set Analysis

Intuitively, the value set semantics of a program is an abstract semantics where the state at each program edge is a map from the lvalues read or written in the program to a set of values. The analysis characterizes a conservative approximation of such a state for each program edge E, i.e. the set of values corresponding to an lvalue l in the solution should include every value contained in the memory location corresponding to l at E in any execution of the program P reaching E.

Formally, the *value set analysis* $\mathcal{VS}$ for a program P is a tuple $(\mathcal{L}_{\mathcal{VS}}, \mathcal{F}_{\mathcal{VS}})$ where $\mathcal{L}_{\mathcal{VS}}$ is the lattice of abstract states and $\mathcal{F}_{\mathcal{VS}}$ is the set of static flow functions. An abstract state in this semantics is denoted by the map $VS : LVals \rightarrow 2^{Values}$, where $LVals$ is the set of lvalues read/written in program P and $Values$ is the set of values that can be contained in any memory location. The domain of the states is thus $LVals \rightarrow 2^{Values}$, denoted as $ValueSets$. Hence the lattice $\mathcal{L}_{\mathcal{VS}}$ is a join-lattice $(ValueSets, \preceq, \top, \bot, \sqcup)$, where for $vs, vs' \in ValueSets$ and $S \subseteq ValueSets$

- $vs \preceq vs'$ iff $\forall l \in LVals : vs(l) \subseteq vs'(l)$
- $\top = \lambda\, l.\, Values$
- $\bot = \lambda\, l. \emptyset$
- $\bigsqcup S = \lambda\, l. \bigcup\limits_{vs \in S} vs(l)$

We allow the analysis to be flow-sensitive and (partially) path-sensitive. Hence, the static flow function for any node N is of the form $F_N : ValueSets \times Edges \rightarrow ValueSets$, allowing it to propagate different abstract states along different successor edges. The flow functions for different types of statements are defined below. Given an expression e, the denotation $[\![e]\!] : ValueSets \rightarrow 2^{Values}$ is a function that returns a set of values obtained from evaluating e on all possible *concrete states* corresponding to a given value set. For an lvalue l, $AliasSet(l)$ denotes the set of lvalues that may represent the same memory location as l. Note that for sequential programs, the $AliasSet$ can be computed from the value sets itself or from some sound pointer analysis such as [1].

If $N \in AsgnStmt$ and is of the form `l := e`, $F_N(vs, \_) = vs'$, where

$$vs'(l') = \begin{cases} [\![\mathtt{e}]\!](vs) & \text{if } l' = \mathtt{l} \\ [\![\mathtt{e}]\!](vs) \cup vs(l') & \text{if } l' \in AliasSet(\mathtt{l}) \\ Values & \text{if } \mathtt{l} \in AliasSet(deref(l')) \\ vs(l') & \text{otherwise.} \end{cases}$$

Intuitively, we destructively update the value set of the lvalue at the LHS, but conservatively update the value set of an lvalue that may be alias of the LHS. If an lvalue is dependent on some alias of the LHS, the memory location corresponding to that lvalue might change. Hence its value set is set to $\top$.

If $N \in BranchStmt$ and the branch condition is `e`, then $F_N(vs, true\_branch) = vs'$ and $F_N(vs, false\_branch) = vs''$, where

$$\forall l, v : v \in \mathit{vs}'(l) \text{ iff } v \in \mathit{vs}(l) \wedge \exists \hat{\mathit{vs}} : \hat{\mathit{vs}}(l) = \{v\} \wedge \mathit{true} \in [\![\mathsf{e}]\!](\hat{\mathit{vs}})$$
$$\forall l, v : v \in \mathit{vs}''(l) \text{ iff } v \in \mathit{vs}(l) \wedge \exists \hat{\mathit{vs}} : \hat{\mathit{vs}}(l) = \{v\} \wedge \mathit{false} \in [\![\mathsf{e}]\!](\hat{\mathit{vs}}).$$

Intuitively, a value v is included in the value set of an lvalue l along the true branch if e can evaluate to *true* with v contained in l. The false branch is similar. Branch statements do not generate any value that was not there in the input value set. Flow functions for other statements are identity functions.

A concrete state of a program P is a map $\mathit{cs} : \mathit{LVals} \rightarrow \mathit{Values}$. Given an action a from an execution $\mathcal{E}$ of the program P, $\mathit{pre}(a)$ and $\mathit{post}(a)$ denote the concrete states immediately before and after a is executed, respectively. If $a_\sharp$ is the last action of $\mathcal{E}$, $\mathit{post}(\mathcal{E}) = \mathit{post}(a_\sharp)$. Given a program edge E, let $\varXi(E)$ denote the set of executions of the program *up to* E, ie, $\varXi(E) = \{\mathcal{E} \mid \mathcal{E} = \langle a'_0, \ldots, a'_\sharp \rangle \text{ and } E = \mathit{enext}(a'_\sharp)\}$. Then for an edge E the collecting value set at E is defined to be

$$\mathit{CVS}[E] = \lambda\, l. \bigcup_{\mathcal{E} \in \varXi(E)} \mathit{post}(\mathcal{E})(l). \tag{1}$$

Let $\mathcal{E} = \langle a'_0, \ldots, a'_n \rangle$ be an execution of the sequential program P. $\Pi_\mathcal{E} = \langle N'_0, \ldots, N'_n \rangle$ is the path corresponding to $\mathcal{E}$ where for all i, $0 \leq i \leq n$, $N'_i = \mathit{stmt}(a'_i)$. Note that for a sequential program, there is an edge in the CFG between N'_i and N'_{i+1} for all i, $0 \leq i < n$. For any analysis $\mathcal{A} = (\mathcal{L}, \mathcal{F})$, the flow function for the path $\Pi_\mathcal{E}$ with the initial state $d \in \mathcal{L}$ along the edge E is defined by $F_{\Pi_\mathcal{E}}(\mathit{vs}, E) = F_{N'_n}(F_{N'_{n-1}}(\ldots(F_{N'_0}(\mathit{vs}, E'_0)\ldots), E'_{n-1}), E)$, where each $E'_i = \langle N'_i, N'_{i+1} \rangle$, $E \in \mathit{esucc}(N'_n)$ and each $F_{N'_i} \in \mathcal{F}$. Let $\varSigma(E)$ be the set of initial paths up to E. Then the ideal *join-over-all-paths* (JOP) solution of the analysis $\mathcal{A}$ on P, denoted by $J_\mathcal{A}$, is given by

$$\forall E \in \mathit{Edges} : J_\mathcal{A}[E] = \bigsqcup_{\Pi \in \varSigma(E)} F_\Pi(\top, E) \tag{2}$$

For value set analysis, the static flow functions overapproximate the runtime behavior, i.e. $\forall l \in \mathit{LVals}\colon v = \mathit{post}(a_n)(l) \Rightarrow v \in F_{\Pi_\mathcal{E}}(\top, \mathit{enext}(a_n))$. We assume the flow function of an empty path to be identity. Hence for a sequential program, $\mathit{CVS} \preceq J_{\mathcal{VS}}$.

Any dataflow analysis (say $\mathcal{A}$) characterizes a further conservative approximation of the JOP by the least solution $S_\mathcal{A}$ for the following set of equations:

$$
\begin{aligned}
X[E_0] &= \top \\
\forall E \in (\mathit{Edges} - \{E_0\}) : X[E] &= \bigsqcup_{E' \in \mathit{epred}(E)} F_{\mathit{npred}(E)}(X[E'], E)
\end{aligned} \tag{3}
$$

As described in standard literature e.g. [14], if flow functions are monotonic, $J_{\mathcal{A}} \preceq S_{\mathcal{A}}$. In particular, $\mathit{CVS} \preceq J_{\mathcal{VS}} \preceq S_{\mathcal{VS}}$. Note that the least solution always exists, but may not be computable for value set analysis. If the underlying lattice has bounded height, the least solution for $\mathcal{A}$ can be computed using an algorithm like Kildall's [15].

5.2 Abstractions of Value Set Semantics

In this section, we define *consistent abstractions* [5] of the value set semantics. An analysis $\mathcal{A} = (\mathcal{L}, \mathcal{F})$, where $\mathcal{L} = (\mathcal{D}, \preceq)$, is a consistent abstraction of $\mathcal{VS}$ if there are a monotonic abstraction function $\alpha\colon \mathit{ValueSets} \rightarrow \mathcal{D}$ and a monotonic concretization function $\gamma\colon \mathcal{D} \rightarrow \mathit{ValueSets}$, such that

- $\forall x \in \mathcal{D} : x = \alpha(\gamma(x))$.
- $\forall \mathit{vs} \in \mathit{ValueSets} : \mathit{vs} \preceq \gamma(\alpha(\mathit{vs}))$.
- $\forall E \in \mathit{Edges} : S_{\mathcal{VS}}[E] \preceq \gamma(S_{\mathcal{A}}[E])$ and $\alpha(S_{\mathcal{VS}}[E]) \preceq S_{\mathcal{A}}[E]$.

Cousot and Cousot [5] provide a sufficient "local" condition to check that one abstraction is a consistent abstraction of another.

5.3 Null-Pointer Analysis

In this section, we describe a simple *null-pointer analysis NPA* as an example of a consistent abstraction of the value set analysis. This analysis can be used to prove a pointer to be non-null when it is dereferenced. Given a program P, an abstract state is a map of the form $\mathit{LVals} \rightarrow \{\mathit{NonNull}, \mathit{MayNull}\}$, where LVals is the set of lvalues in P. The domain of the analysis $\mathcal{D}_{\mathit{NPA}}$ is a set of all such maps. The concretization function $\gamma : \mathcal{D}_{\mathit{NPA}} \rightarrow \mathit{ValueSets}$ is defined below for $d \in \mathcal{D}_{\mathit{NPA}}$:

$$
\gamma(d)(l) = \begin{cases} \mathit{Values} & \text{if } d(l) = \mathit{MayNull} \\ \mathit{Values} - \{\texttt{NULL}\} & \text{if } d(l) = \mathit{NonNull}. \end{cases}
$$

Similarly, if a value set contains NULL, the abstraction function maps it to *MayNull*, otherwise to *NonNull*.

For $d_1, d_2 \in \mathcal{D}_{NPA}$ and $l \in \mathit{LVals}$, the join operation is defined below:

$$d_1 \sqcup d_2(l) = \begin{cases} \mathit{NonNull} & \text{if } d_1(l) = d_2(l) = \mathit{NonNull} \\ \mathit{MayNull} & \text{otherwise.} \end{cases}$$

The flow functions for a node N, edge E and state d are given below. By $d[l \leftarrow a]$ we denote a map same as d except that $d(l) = a$.

If N is of the form `if (l != NULL)`:

$$F_N(d, E) = \begin{cases} d[l \leftarrow \mathit{NonNull}] & \text{if } E \text{ is the true edge} \\ d & \text{otherwise.} \end{cases}$$

If N is of the form `l := e`:

$$F_N(d, E)(l') = \begin{cases} \mathit{NonNull} & \text{if } l' = \texttt{l},\ e \text{ is an lvalue, and } d(e) = \mathit{NonNull} \\ d(l') & \text{if } l' \notin \mathit{AliasSet}(\texttt{l}) \text{ and } \texttt{l} \notin \mathit{AliasSet}(\mathit{deref}(l')) \\ d(l') & \text{if } l' \in \mathit{AliasSet}(\texttt{l}),\ e \text{ is an lvalue, and } d(e) = \mathit{NonNull} \\ \mathit{MayNull} & \text{otherwise.} \end{cases}$$

The flow functions for all other statements are identity functions. Note that this analysis requires a may-alias analysis. It is easy to see that this is an abstraction of the value set analysis.

6 Analysis for Concurrent Programs

Given a concurrent program P and a dataflow analysis $\mathcal{A}$ for sequential programs, our technique converts $\mathcal{A}$ to an analysis for P that is sound if P is datarace-free and $\mathcal{A}$ falls into the class of analyses described in Section 5. We assume availability of a sound may-alias analysis. For example, flow-insensitive may-alias analyses such as [1] are sound for concurrent programs.

1. **Construction of the sync-CFG:** We first construct an extended CFG C for P, called *sync-CFG*, as follows. We begin by taking the disjoint union of the CFGs of threads of P. We then add the *may-synchronize-with* (msw) edges between nodes of these CFGs as described below. These edges are added between nodes that might participate in a *synchronizes-with* relation at runtime. More specifically, we add the the following types of edges:

    1. From a `spawn` node to the `start` node of the child thread.
    2. From an `end` node of a thread to the corresponding `join` node of the parent thread.

3. From an `unlock` node to a `lock` node, if they access the same lock and if the corresponding threads may run in parallel.

In case the exact set of edges are difficult be compute, we can use any over-approximation of it. For example, if locks can be aliased (not possible in the language described in Section 4.1), we use the may-alias analysis to find out whether a `lock/unlock` pair may access the same lock variable at run-time. Similarly, simple control flow based techniques can be applied to conservatively detect whether two threads can run in parallel. Figure 1 shows the msw edges added for the shown program fragment.

2. Constructing Flow functions: Flow functions of the synchronization statements are simply identity functions. Flow functions of other nodes are same as that of $\mathcal{A}$.

3. Constructing and Solving Flow Equations: The sync-CFG C corresponds to a (non-deterministic) sequential program. We construct the flow equations for our analysis $\mathcal{A}$ over C as given in Equation 3. Finally, we compute the least solution of these set of equations over the sync-CFG C.

Interpreting the Result: As we show in Section 7, the solution given by our technique conservatively approximates the value sets of relevant lvalues at a program edge, while it may not be sound for non-relevant lvalues. Hence the client of the analysis must use the result to reason about only relevant lvalues. For example, in the program of Figure 1, our analysis wrongly concludes that `p->data` must be non-null at `C1`, but `p->data` is not relevant at `C1`. On the other hand, it finds `p->data` to be non-null at `C3` where it is relevant and this fact is sound.

Alternatively, to present a solution that is sound for all lvalues, we define a program dependent operation *havoc* on value set states as follows. For $vs \in \mathit{ValueSets}$, $E \in \mathit{Edges}$ and $l \in \mathit{LVals}$,

$$\mathit{havoc}(vs, E)(l) = \begin{cases} \mathit{Values} & \text{if } l \text{ is not relevant at } E \\ vs(l) & \text{otherwise} \end{cases}$$

Then for an abstract analysis $\mathcal{A}$, $\alpha(\mathit{havoc}(\gamma(S_{\mathcal{A}})[E], E))$ (or any conservative approximation of it) is the final solution at edge E. This step essentially sets the abstract values of non-relevant lvalues at every program point to the most conservative value. Hence, this method produces useful results only for relevant lvalues at each program edge, but is sound for all lvalues.

The alias analysis can be computed in time polynomial in size of the program. The conservative sync-CFG can also be computed in polynomial time. The modified program represented by the sync-CFG is again polynomial in size of the original program and the least solution can be computed in time polynomial in the size of the sync-CFG. Hence, the entire algorithm takes time polynomial in size of the original program.

7 Proof of Soundness

7.1 For Value Set Analysis

In this section we prove that given a datarace-free concurrent program P, the solution characterized by the technique described in Section 6 is a conservative approximation of the collecting semantics defined by Equation 1 for value set analysis with respect to the relevant lvalues at each program edge. Note that the least solution to the equation system 3 is a conservative approximation of the JOP solution over the sync-CFG C of P. Thus it is sufficient for our purpose to argue that if there is an execution of P in which an lvalue l has a value v at a program edge E where l is relevant, then there is an initial path in the *sync-CFG* to E along which the value v is included in the value set of l at E. This is shown in Lemma 7.2 below.

We begin with a lemma that will be useful in proving Lemma 7.2.

Lemma 7.1 *Let $\mathcal{E} = \langle a_0, \ldots, a_j \rangle$ be an execution of the program P. Let l be a relevant lvalue at $stmt(a_j)$ and $v = pre(a_j)(l)$. Let M be the set of memory locations corresponding to the lvalues $\{l\} \cup deref(l)$ at a_j. Let a_i, $i < j$ be the last action before a_j that writes to a memory location in M. Then there exists a static path Π in the sync-CFG C from $stmt(next(a_i))$ to $stmt(prev(a_j))$ such that $\forall vs \in ValueSets\colon v \in vs(l) \Rightarrow v \in F_\Pi(vs, E)(l)$, where $E = eprev(a_j)$.*

Proof As l is relevant at $stmt(a_j)$, a_j reads all the memory locations of M. As a_i is the last action before a_j that writes to one of these memory locations, a_i and a_j are conflicting. As the program is datarace-free, we must have $a_i \leq_{hb}^{\mathcal{E}} a_j$. Recall that the happens-before relation is the reflexive transitive closure of program-order and synchronizes-with relations. It is easy to see that if for two actions b and b' from $\mathcal{E}$, $b <_{po}^{\mathcal{E}} b'$ or $b <_{sw}^{\mathcal{E}} b'$, then there is an edge in C from $stmt(b)$ to $stmt(b')$. Hence, a path Π' from $stmt(a_i)$ to $stmt(a_j)$ in C can be constructed by joining the edges of C corresponding to these *po* and *sw* relations. As neither a_i nor a_j can be synchronization actions (they read/write to lvalues), hence, in Π', $stmt(a_i)$ is succeeded by

$\mathit{stmt}(\mathit{next}(a_i))$ and $\mathit{stmt}(a_j)$ is preceded by $\mathit{stmt}(\mathit{prev}(a_j))$. Clearly, this path is a subsequence of the list of nodes corresponding to $a_i, \ldots, a_j$. We further obtain Π from Π' by excluding $\mathit{stmt}(a_i)$ and $\mathit{stmt}(a_j)$ from Π'.

By contradiction, let vs be a value set state such that $v \in \mathit{vs}(l)$ and $v \notin F_\Pi(\mathit{vs}, E)$. Then there must be a node N and an edge E in Π such that $E \in \mathit{esucc}(N)$ and there is a value set state vs' such that $v \in \mathit{vs}'(l)$ and $v \notin F_N(\mathit{vs}', E)(l)$. From the definition of flow functions from Section 5.1, this can be possible only in the following two cases:

- N is an assignment to l. As a_i was the last assignment to any memory location in M, the memory location corresponding to l does not change after a_i till a_j. If LHS of N was l, then the corresponding action in $a_{i+1}, \ldots, a_{j-1}$ must have written to a memory location in M, which is not possible because of the choice of a_i.

- N is a branch statement and E is the true successor edge and the condition e is such that it does not evaluate to true when l has a value v. This is not possible as the execution took the true branch E with the value v in l. The argument is similar for the false branch.

Hence, there can be no such vs and the lemma is proved. ▮

Lemma 7.2 *Let $\mathcal{E} = \langle a_0, \ldots, a_j\rangle$ be an execution of P. Let l be an lvalue relevant at $\mathit{stmt}(a_j)$ and $v = \mathit{pre}(a_j)(l)$. Let $N = \mathit{stmt}(a_j)$ and $E \in \mathit{epred}(N)$ in C. Then there exists an initial static path Θ in C from N_0^M up to E, such that $v \in F_\Theta(\top, E)(l)$.*

Proof We prove the lemma by induction on the length $k = j + 1$ of the execution $\mathcal{E}$.

Base case: If $k = 0$, $\Theta = \epsilon$ (empty path) and $F_\Theta(\top, E) = \top$. Clearly, $v \in \top(l)$.

Induction step: Let us assume the result for $k < n$ and consider the case for $k = n$.

Let a_i be the last action in $\mathcal{E}$ before a_j which writes to a memory location corresponding to the lvalues in $\{l\} \cup \mathit{deref}(l)$ at a_j. Then we have $v = \mathit{post}(a_i)(l)$ as the value contained in l cannot change after a_i in $\mathcal{E}$. As $\hat{N} = \mathit{stmt}(a_i)$ is an assignment statement, let us denote the singleton edge in $\mathit{esucc}(\hat{N})$ by $\hat{E}$. Then either of the following is true:

1. $\hat{N}$ writes to a memory location corresponding to an lvalue in $\mathit{deref}(l)$ at a_j. In this case, any path $\hat{\Theta}$ from N_0^M to $\hat{N}$ (both inclusive) in C will have $v \in F_{\hat{\Theta}}(\top, \hat{E})(l)$, as the flow function of $\hat{N}$ sets the value set

of l to *Values*. It is easy to see that if a node gets executed, then there is a path from N_0^M to that node in C.

2. $\hat{N}$ writes to the memory location corresponding to l. Let the RHS be the expression e. As the length of $\langle a_0, \ldots, a_i \rangle$ is less than k, by the induction hypothesis, there is a path Θ'' from N_0^M up to but not including $\hat{N}$, such that for all lvalue l' read in e, $v' = pre(a_i)(l') \Rightarrow v' \in F_{\Theta''}(\top, epred(a_i))(l')$. Let $\hat{\Theta} = \Theta''.\hat{N}$. From the definition of static flow function, this implies $v \in F_{\hat{\Theta}}(\top, \hat{E})(l)$.

Now let Π be the path from $stmt(next(a_i))$ to $stmt(prev(a_j))$, excluding both, as given by Lemma 7.1. Clearly, $E = eprev(a_j)$. Let $\Theta = \hat{\Theta} \cdot \Pi$. As $v \in F_{\hat{\Theta}}(\top, \hat{E})(l)$ and $v = post(a_i)(l)$, using Lemma 7.1, we have $v \in F_{\Theta}(\top, E)(l)$. $\blacksquare$

We finally prove the following soundness theorem:

Theorem 7.3 *Let P be a datarace-free concurrent program. Let $S_{\mathcal{VS}}$ be the solution returned by our technique and let CVS be the collecting value set of P. If l is an lvalue relevant at an edge E, then $CVS[E](l) \subseteq S_{\mathcal{VS}}[E](l)$.*

Proof As already observed in the beginning of this section, since our analysis finds a conservative approximation of the *join-over-all-paths* solution over the paths of *sync-CFG* C of P, it is sufficient to show that if there is an execution of P which has a value v in an lvalue l at a program edge E where l is relevant, then there is an initial path in C to E along which the value v is included in the value set of l at E. This is a direct consequence of Lemma 7.2. Hence the theorem is proved. $\blacksquare$

The following corollary is immediate from Theorem 7.3 and definition of *havoc*.

Corollary 7.4 *For a datarace-free program P and for all edges E, $CVS[E] \preceq havoc(S_{\mathcal{VS}}[E], E)$.*

7.2 For Abstractions of Value Set Semantics

We now show that the havoced solution characterized by our technique for any consistent abstraction of value set semantics conservatively approximates the collecting semantics for value set analysis for a datarace-free program.

Theorem 7.5 *Let $\mathcal{A}$ be a consistent abstraction of the value set semantics and $S_{\mathcal{A}}$ be the solution returned by our analysis for a datarace-free concurrent program P. Then for all edges E, $CVS[E] \preceq havoc(\gamma(S_{\mathcal{A}})[E], E)$.*

Proof From definition of consistent abstraction, $S_{\mathcal{VS}} \preceq \gamma(S_{\mathcal{A}})$. As *havoc* is monotonic, $havoc(S_{\mathcal{VS}}[E], E) \preceq havoc(\gamma(S_{\mathcal{A}})[E], E)$. From Corollary 7.4, we have $CVS[E] \preceq havoc(S_{\mathcal{VS}}[E], E)$. Thus, $CVS[E] \preceq havoc(\gamma(S_{\mathcal{A}})[E], E)$. ▮

8 Context-Sensitive Analysis

For programs with procedure calls, context-sensitive analyses are required to improve the precision of sequential analyses. In this section, we describe how one such context-sensitive technique, namely the *call-string approach* [26], can be integrated into our framework. For sake of completeness, we briefly describe the sequential call-string approach here.

We first augment our language described in Section 4.1 with procedure calls. A thread now consists of a number of procedures, each with their own rooted CFGs. Each thread has a *entry procedure* with the same name as the thread. Only the entry procedures have start and end edges and nodes. Execution of a thread starts with the execution of the `start` node of the entry procedure. We define two new types of statements : *CallStmt* of the form `<procname>()`, where `<procname>` is name of some procedure and *ReturnStmt* of the form `return`. The control flow structure of a thread is represented by a *Interprocedural Control Flow Graph* (ICFG), which is obtained by taking disjoint union of all the CFGs of all the procedures that can be called during execution of the thread and adding *call edges* and *return edges*. Call edges are added from call statements to the root nodes of the called procedures' CFGs. Return edges are added from return statements to the statements immediately following the call statements calling the procedures containing the return statements. Note that in any CFG, there are no edges from call statements to the next statements in the same procedures.

8.1 Sequential Analysis with Call-Strings

As before, the sequential program consists of a single `main` thread. Let $\mathcal{A} = (\mathcal{L}, \mathcal{F})$ be the underlying sequential dataflow analysis with $\mathcal{L} = (\mathcal{D}, \preceq)$. Let $C^* = (Nodes, Edges, E_0, E_\sharp)$ denote the ICFG of the program.

We define a *call-string* γ as a (possibly empty) sequence of call statements. Let Γ be the set of all possible call-strings. The empty call-string is denoted by ϵ. The size of a call-string γ is denoted by $|\gamma|$. The ith component of γ is denoted by $\gamma[i]$ and the substring from ith to jth component (both inclusive) is denoted by $\gamma[i..j]$. The operator "$\cdot$" denotes the string append operation.

The call-string approach defines a new dataflow analysis framework $\mathcal{A}^* = (\mathcal{L}^*, \mathcal{F}^*)$, where $\mathcal{L}^* = (\mathcal{D}^*, \preceq)$. The domain $\mathcal{D}^*$ is the space of all maps from Γ into $\mathcal{D}$. The ordering in $\mathcal{L}^*$ is the pointwise ordering on $\mathcal{L}$, i.e. for $\xi_1, \xi_2 \in \mathcal{D}^*, \xi_1 \preceq \xi_2$ iff $\forall \gamma \in \Gamma, \xi_1(\gamma) \preceq \xi_2(\gamma)$. Similarly, the join operation in $\mathcal{L}^*$ is defined as a pointwise join on $\mathcal{L}$, i.e., for $\xi_1, \xi_2 \in \mathcal{D}^*, \gamma \in \Gamma$, $(\xi_1 \sqcup \xi_2)(\gamma) = \xi_1(\gamma) \sqcup \xi_2(\gamma)$. The largest element in $\mathcal{L}^*$ is $\top^* = \lambda\,\gamma.\top$. Similarly, the smallest element is $\bot^* = \lambda\,\gamma \cdot \bot$.

In order to define the flow functions, we first define a partial binary operator $\circ : \Gamma \times \mathit{Edges} \rightarrow \Gamma$ in the following way:

$$\gamma \circ \langle N, N' \rangle = \begin{cases} \gamma \cdot N & \text{if } \langle N, N' \rangle \text{ is a call edge} \\ \gamma[1..|\gamma|-1] & \text{if } \langle N, N' \rangle \text{ is a return edge and } \gamma[|\gamma|] \text{ is the} \\ & \text{corresponding call statement} \\ \gamma & \text{otherwise} \end{cases}$$

A flow function $F^*_N \in \mathcal{F}^*$, where $N \in \mathit{Nodes}$, is a function from $\mathcal{D}^* \times \mathit{Edges}$ to $\mathcal{D}^*$, defined below:

$$F^*_N(\xi, E)(\gamma) = \begin{cases} F_N(\xi(\gamma'), E) & \text{if there exists a unique } \gamma' \text{ such that } \gamma = \gamma' \circ E \\ \bot & \text{otherwise} \end{cases}$$

The analysis characterizes the call-string solution $S^*_{\mathcal{A}}$ as the least solution of the following set of equations:

$$\begin{aligned} X^*[E_0] &= \lambda\,\gamma.\ \text{if } \gamma = \epsilon \text{ then } \top \text{ else } \bot \\ \forall E \in (\mathit{Edges} - \{E_0\}) : X^*[E] &= \\ & \bigsqcup_{E' \in \mathit{epred}(E)} F^*_{\mathit{npred}(E)}(X^*[E'], E) \end{aligned} \tag{4}$$

The final summarized solution $S_{\mathcal{A}}$ is defined for an edge $E \in \mathit{Edges}$ as

$$S_{\mathcal{A}}[E] = \bigsqcup_{\gamma \in \Gamma} S^*_{\mathcal{A}}[E](\gamma)$$

8.2 Integrating Call-String Analysis into Our Framework

Intuitively, any abstract state that is reachable at a release node via any call-string should be joined with all abstract states at the corresponding acquire node, as the release and the acquire nodes may belong to different dynamic threads at runtime and there is no relation among the call-strings of different threads. Therefore we modify the call-string lattice by adding a second component and use it to propagate the joined abstract value along an msw edge. If the abstract state corresponding to some call-string is $\bot$ at the acquire node, it implies that the call-string is not reachable at that program node. Hence we join the propagated value only with the call-strings that are mapped to non-bottom values.

Given a concurrent program P and a call-string based analysis $\mathcal{A}^*$, we first construct the *sync-ICFG* $\mathbb{C} = (\mathit{Nodes}, \mathit{Edges}, E_0, E_\sharp)$ in the same way as the context-insensitive case. We define a new analysis framework $\mathbb{A} = (\mathbb{L}, \mathbb{F})$ as follows. We define $\mathbb{L} = (\mathbb{D}, \preceq)$ where $\mathbb{D} = \{(\xi, d) \in \mathcal{D}^* \times \mathcal{D} \mid \forall \gamma \in \Gamma : \xi(\gamma) \neq \bot \Rightarrow d \preceq \xi(\gamma)\}$ and $(\xi_1, d_1) \preceq (\xi_2, d_2)$ iff $\xi_1 \preceq \xi_2$ and $d_1 \preceq d_2$. For $\psi_1, \psi_2 \in \mathbb{D}$ where $\psi_1 = (\xi_1, d_1)$ and $\psi_2 = (\xi_2, d_2)$, $\psi_1 \sqcup \psi_2 = \langle (\xi_1 \sqcup \xi_2) \nabla (d_1 \sqcup d_2), (d_1 \sqcup d_2) \rangle$. The operator $\nabla : \mathcal{D}^* \times \mathcal{D} \rightarrow \mathcal{D}^*$ is defined as follows: For $\xi \in \mathcal{D}^*$ and $d \in \mathcal{D}$,

$$(\xi \nabla d)(\gamma) = \begin{cases} \xi(\gamma) \sqcup d & \text{if } \xi(\gamma) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Given $\xi \in \mathcal{D}^*$, we define *reduce* : $\mathcal{D}^* \rightarrow \mathcal{D}$ as $\mathit{reduce}(\xi) = \bigsqcup_{\gamma \in \Gamma} \xi(\gamma)$. We define the flow function $\hat{F} \in \mathbb{F}$ for $N \in \mathit{Nodes}$, $\psi \in \mathbb{D}$ and $E \in \mathit{esucc}(N)$ below. Let $\psi = (\xi, d)$ and $d' = \mathit{reduce}(\xi)$. Then

$$\hat{F}_N(\psi, E) = \begin{cases} \langle \lambda\, \gamma.\ \text{if } \gamma = \epsilon \text{ then } d' \text{ else } \bot, d' \rangle & \text{if } N \text{ is a } \texttt{spawn} \text{ node and } E \\ & \text{is an msw edge} \\ \langle \lambda\, \gamma. \bot, d' \rangle & \text{if } N \text{ is not a } \texttt{spawn} \text{ node but } E \\ & \text{is an msw edge} \\ \langle F^*_N(\xi), \bot \rangle & \text{otherwise} \end{cases}$$

As in the context-insensitive case, the call-string solution $\mathbb{S}_\mathcal{A}$ is characterized by the least solution of the following set of equations:

$$\begin{array}{r} \mathbb{X}[E_0^M] = \langle \lambda\, \gamma \cdot \text{ if } \gamma = \epsilon \text{ then } \top \text{else } \bot, \bot \rangle \\ \forall E \in (\mathit{Edges} - \{E_0\}) : \mathbb{X}[E] = \\ \bigsqcup_{E' \in \mathit{epred}(E)} \hat{F}_{\mathit{npred}(E)}(\mathbb{X}[E'], E) \end{array} \tag{5}$$

The final *summarized* solution $S_\mathcal{A}$ at a program edge E is given by

$$S_\mathcal{A}[E] = \mathit{reduce}(\xi_E)$$

where $\mathbb{S}_\mathcal{A}[E] = (\xi_E, d_E)$.

The solution described in previous section may not be computable even if $\mathcal{L}$ is of finite height, because in presence of recursive procedure calls, the call-strings may grow unboundedly. In practice, we use an approximate but sound call-string approach where we represent a call-string by a finite length suffix, as described in [26].

The soundness of our context-sensitive analysis can be proved in a similar way to the context-insensitive analysis.

9 Implementation

We have implemented our technique into a framework STAND (STatic ANanlysis for Datarace-free programs) that automatically converts dataflow analyses for sequential Java programs to analyses for concurrent program. We use Soot [27] as the frontend and SPARK [18] for the alias analysis. We instantiated STAND for a simple null-dereference analysis and used it to prove safety of dereferences in three large Java programs, `jdbm` (a transactional persistence engine), `jdbf` (an object-relational mapping system) and `jtds` (a JDBC driver). As mentioned in [23], developers of these programs fixed the dataraces detected by Chord [23] and hence, they are likely to be datarace-free. All our experiments are carried out on an Intel Xeon machine with 2.27 GHz clock and 2 GB RAM.

We report the percentage of dereferences proven to be safe for our benchmark programs in column *% safe* of Table 1. We observe that on an average, STAND is able to prove over 80% of the dereferences safe. We compare our precision with an unsound sequential analysis that is obtained by removing the msw edges (except for edges from `spawn` to `start`) from a sync-CFG and running the same underlying sequential analysis on the modified graph. Note that this analysis is unsound as it does not account for the interference from other threads. The column *% seq-safe* denotes the percentage of dereferences shown to be safe by this unsound, sequential analysis. We observe that the difference between *% safe* and *% seq-safe* is small. Hence it can be concluded that the loss of precision in STAND can largely be attributed to the underlying sequential analysis. Finally, we report the total analysis time in two parts: *SPARK time* denotes the time taken by the SPARK alias analysis and *STAND time* denotes the time taken by our analysis excluding alias analysis. Note that the analysis time of STAND after alias analysis is fairly small for these benchmark programs.

We also compare our approach with Radar [4] by implementing null-pointer analysis for concurrent C programs with the pthread library. This implementation uses the LLVM compiler infrastructure [19]. We executed

Table 1: Results using STAND

Benchmark	LOC (w/o lib)	% safe	% seq-safe	STAND time(s)	SPARK time(s)
jdbm	19077	79.5	81.0	2.518	35
jdbf	15923	81.9	82.8	2.883	120
jtds	66318	80.3	84.3	1.709	51

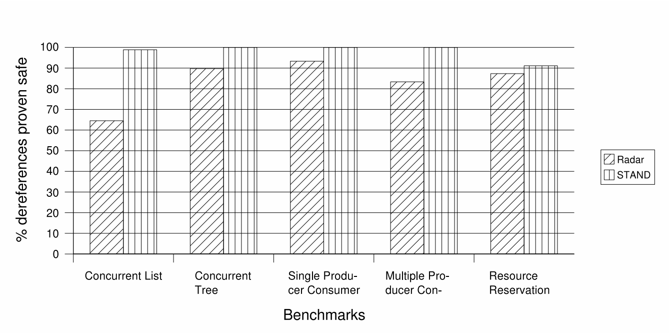


Figure 4: Precision comparison between Radar and Stand

Radar and STAND on the five concurrent programs (average size greater than 1000 LOC) implementing some classic concurrent algorithm and data-structures. The precision results, measured as the percentage of dereferences proven to be safe, are shown in Figure 4. We observe that STAND consistently does better than Radar. We manually checked that the reason behind this difference is that Radar conservatively kills a dataflow fact whenever there is a race possibly affecting that fact whereas STAND propagates the exact facts from one thread to another. The analysis time of STAND for this set of programs is only 0.8 seconds on average.

Acknowledgments.

We thank Ankur Sinha for helping with the experiments.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU , University of Copenhagen, 1994.

[2] H.-J. Boehm. Reordering Constraints for Pthread-Style Locks. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, New York, NY, USA, 2007. ACM.

[3] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, New York, NY, USA, 2008. ACM.

[4] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow Analysis for Concurrent Programs Using Datarace Detection. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–326, New York, NY, USA, 2008. ACM.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[6] M. B. Dwyer and L. A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, New York, NY, USA, 1994. ACM.

[7] A. Farzan and Z. Kincaid. Compositional Bitvector Analysis for Concurrent Programs with Nested Locks. In R. Cousot and M. Martel, editors, *SAS 2011*, volume 6337 of *LNCS*, pages 253–270. Springer Berlin / Heidelberg, 2011.

[8] C. Flanagan, S. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. In D. Le Mtayer, editor, *ESOP 2002*, volume 2305 of *LNCS*, pages 285–301. Springer Berlin / Heidelberg, 2002.

[9] C. Flanagan and S. Qadeer. Thread-Modular Model Checking. In T. Ball and S. Rajamani, editors, *SPIN 2003*, volume 2648 of *LNCS*, pages 624–624. Springer Berlin / Heidelberg, 2003.

[10] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-Modular Shape Analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–277, New York, NY, USA, 2007. ACM.

[11] D. Grunwald and H. Srinivasan. Data Flow Equations for Explicitly Parallel Programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, New York, NY, USA, 1993. ACM.

[12] IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 2004.

[13] JSR-133 Expert Group. JSR-133: Java Memory Model and Thread Specification. `http://www.cs.umd.edu~pugh/java/memoryModel/jsr133.pdf`, August 2004.

[14] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7:305–317, 1977.

[15] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[16] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 18(3):268–299, 1996.

[17] J. Lee, D. A. Padua, and S. P. Midkiff. Basic Compiler Algorithms for Parallel Programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 1999. ACM.

[18] O. Lhoták. Spark: A Flexible Points-to Analysis Framework for Java. Master's thesis, McGill University, December 2002.

[19] LLVM Project. The LLVM Compiler Infrastructure. `http://llvm.org/`.

[20] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-Modular Counterexample-Guided Abstraction Refinement. In R. Cousot and M. Martel, editors, *SAS 2010*, volume 6337 of *LNCS*, pages 356–372, 2010.

[21] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[22] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[23] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[24] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 77–90, New York, NY, USA, 1999. ACM.

[25] J. Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[26] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[27] R. Valle-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, McGill University, July 2000.

[28] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.