Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs*

Venkatesh Prasad Ranganath and John Hatcliff

Department of Computing and Information Sciences, Kansas State University,
234 Nichols Hall, Manhattan KS, 66506, USA
{rvprasad,hatcliff}@cis.ksu.edu

Abstract. In this paper, we show how previous work on escape analysis can be adapted and extended to yield a static analysis that is efficient yet effective for reducing the number of *interference dependence* edges considered while slicing concurrent Java programs. The key idea is to statically detect situations where run-time heap objects are reachable from a single thread and use it to prune spurious interference dependence edges. We also show how this analysis can be extended to reduce the number of *ready dependence* edges – dependences that capture indefinite delay due to Java synchronization constructs.
The analysis we describe has been implemented in the Bandera[9] slicer which is being applied to reduce the size of software models for model-checking. Using this implementation, we give experimental results that demonstrate the effectiveness of our approach. We believe leveraging escape information in the manner we describe is a crucial element in scaling slicing techniques to larger concurrent object-oriented programs.

1 Introduction

Program slicing has proven to be effective for program debugging, understanding, and specialization. Despite the widespread use of concurrent object-oriented languages such as Java and C#, the amount of work on relevant techniques for slicing programs in these languages is much less than the amount of work devoted to slicing sequential programs written in languages such as C. Most of the algorithms [32] proposed for slicing concurrent programs extend the structures and algorithms proposed for slicing sequential programs. Results from such extensions are often overly pessimistic as they adapt a rather simplified approach to deal with the additional dependencies ([13]) that arise in concurrent programs.

Interference dependence [18] is one such additional dependence that is similar to the notion of *data dependence* that exists in conventional slicing for sequential programs. In a sequential program, a statement s_2 is said to be *data dependent* on another statement s_1 if a definition to (i.e., an assignment to) variable x at s_1

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), and by Rockwell-Collins and by Intel Corporation (Grant 11462).

reaches a use of x in s_2 along a particular program control-flow path. In concurrent program slicing, an interference dependence may be explained intuitively as a data dependence between s_1 and s_2 where these statements occur in different threads, i.e., thread interleaving allows a definition of x at s_1 to be used at s_2. Just as with data dependences, calculation of interference dependences becomes complicated when reaching definitions associated with heap-allocated objects are considered. In particular, precise alias information is required to avoid retreating back to safe but very conservative definitions such as "any definition of a field f of any object o_1 of type T can reach any use of field f of any object o_2 of the same type T".

Efficient calculation of precise interference information for concurrent OO programs is challenging because it is difficult to statically reason about all the possible run-time thread interleavings. Existing approaches for Java slicing tend to fall into two categories: approaches that use exponential symbolic execution to approximate interleavings [5, 18, 24] and approaches that avoid the high cost of the symbolic execution technique by making very conservative assumptions [13, 32].

The techniques that we propose in this paper are based on two well-studied forms of static analysis for concurrent Java – escape analysis and race detection analysis – that can be leveraged to improve the precision of interference dependence, hence, the resulting program slice. Our approach is much more precise than the conservative approaches mentioned above and much less expensive than the symbolic execution-based techniques.

Until now, as far as we know, there has been no work reporting the use of these analyses to improve the precision of dependence information required to slice concurrent OO programs. In fact, the recent work by Krinke ([19]) that uses symbolic execution to prune interference edges does not consider dynamically created objects or pointers, and so our approach can also be viewed as complementary to such techniques.

The contributions of this paper are as follows.

- We adapt and extend the escape analysis by Ruf [28] (which was targeted to removal of unnecessary synchronization) to yield more precise information about sharing and aliasing of objects across threads by changing the points at which flow data is merged and by adding additional symbolic entities to track specific def/use and wait/notify relationships.
- We describe how to apply the results of escape analysis to improve the precision of calculation of interference and ready dependences in the context of program slicing.
- We report experimental results from applying our techniques to Java Grande benchmarks that demonstrate their effectiveness in removing spurious dependence edges.

The following section defines the notions of interference and ready dependence used in slicing concurrent programs, and it presents an example to illustrate the reductions that can be achieved by using escape analysis. Section 3 presents our modified version of Ruf's escape analysis. Section 4 explains how

the information from our analysis can be used to drive transformations such as program slicing. Section 5 presents the results of our experiments. Section 6 describes related work, and Section 7 concludes and describes possible extensions to our approach.

2 Background

2.1 Data flow across threads in Java

Interference dependences can only arise when an update is made to a data item that is "shared" between two or more threads. Thus, we begin by considering how an object can be shared between threads using the simple program of Example 1 (ignore the `mainAlt` method for now). Java[12] supports threads via `Thread` objects.[1] The example program has three threads: the main thread which runs the method `main` in class `Home`, a thread associated with an instance of `Man`, and a thread associated with an instance of `Wife`. In Java, a thread becomes "alive" when `java.lang.Thread.start()` is dispatched on the associated thread object (e.g., the calls to `start` at line 30 and line 31). This dispatch results in the invocation of `run()` on the receiver of the `start()` method (e.g., the call to `start()` at line 31 causes the newly activated thread to invoke the `run` method at line 16).

Let us now consider all possible ways in which a heap object o created by a thread t can be communicated to a newly created thread t'. Java disallows any parameters for the `run()` or `start()` methods of `java.lang.Thread`. Thus, the only way to provide data to the run method of t' is to have it be reachable[2] from the `this` variable of the `Thread` instance for t' or to have it be reachable from static fields (in essence, global variables). Note also that there are only two ways to assign data/values to instance fields of an object. One is via direct assignments to reachable fields. The other is via invocation of an instance method in which the fields reachable via the receiver will be assigned data reachable from the arguments to the method. For instance, in the example program, the `main` thread communicates the savings account object to the `run()` method of the `Wife` thread by invoking the constructor of `Wife` with the account as a parameter; the constructor then assigns the account object to the `savings` field reachable from `this`. The main properties of this example that we will use to illustrate object escape properties are: (a) an escaping savings account object ends up being shared between `Man` and `Wife` instances and (b) a new `Acct` object is created and used only in the `run()` method for `Wife`.

[1] The term *thread* indicates an executing entity and the term *thread object* indicates an instance of `java.lang.Thread` or any of it's subclasses.

[2] From here on, "reachable" will signify reachability based on following references held in accessible fields.

```
1  class Acct {
2    protected int balance;
3    Acct() {
4      balance = 100;
5    }
6  } // End of class Acct
7
8  class Man extends Thread {
9    protected Acct savings;
10   protected Acct checking;
11   Man(Acct act) {
12     this.savings = act;
13     checking = new Acct();
14     int i = checking.balance;
15   }
16   public void run() {
17     savings.balance += 20;
18     synchronized(savings) {
19       savings.notify();
20     }
21     checking.balance += 10;
22   }
23 } // End of class Man
24
25 public class Home {
26   public static void main(String[] s) {
27     Acct savings = new Acct();
28     Wife wife = new Wife(savings);
29     Man man = new Man(savings);
30     wife.start();
31     man.start();
32   }
33
34 public static void mainAlt(String[] s) {
35     Acct savings = new Acct();
36     Thread wife, man;
37     for(int i = 2; i <= 3; i++)
38       if (i % 2 == 0)
39         wife = new Wife(savings);
40       if (i % 3 == 0)
41         man = new Man(savings);
42     wife.start();
43     man.start();
44   }
45 } // End of class Home
46
47 class Wife extends Thread {
48   protected Acct savings;
49   protected Acct checking;
50   Wife(Acct act) {
51     this.savings = act;
52     checking = new Acct();
53   }
54   public void run() {
55     Acct newAcct = new Acct();
56     synchronized(savings) {
57       savings.wait();
58     }
59     savings.balance -= 20;
60     synchronized(checking) {
61       checking.wait();
62     }
63     checking.balance -= 10;
64     newAcct.balance += 10;
65   }
66 } // End of class Wife
```

Fig. 1: A simple Java program illustrating object sharing between threads.

2.2 Interference Dependence

Definition 1. *Let P be a program, o be an object with field f, and t_1 and t_2 be threads such that $t_1 \neq t_2$. If there exists an execution trace of P such that f is written at trace state s_m by a statement at program point m executed by t_1 and read at state s_n by a statement at program point n executed by t_2 (with s_n occurring after s_m) and no write to $o.f$ occurs between s_m and s_n, then n is interference dependent on m.*

Given the above definition of interference dependence in terms of traces of concurrent programs, it is obvious that precise and efficient calculation of interference edges is infeasible. Therefore, one must generally fall back to safe approximations that are cheaper to compute such as the following:

> *If a field f of object o is being written and read at program points m and n, respectively, and m and n occur in different threads t_m and t_n, then n is interference dependent on m*

By referring to the example of Figure 1, we now explain (1) how interference edges are detected in previous presentations of slicing [13, 32], (2) how our modification of Ruf's escape analysis can be used to safely reduce the number of edges, and (3) how simple extensions to track aliasing gives even further reduction.

Type-based approach: A simple approach is to classify field read/writes as interfering if m represents a read of an expression such as `a1.f`, n is a write to `a2.f`, and `a1.f` and `a2.f` have identical signature (i.e., both instances of `f` represent the same field of the same class – note that field resolution in Java is based on the static type of the primary[3] and `not` it's dynamic type) then m is interference dependent on n. While this approximation of interference dependence is easy to compute, it includes inference edges even for objects that are not shared. Hence, it leads to spurious interference dependence edges such as ones between line 21 and both line 59 and line 64.

Leveraging escape analysis: The definition of interference dependence implies that objects that induce interference dependences must be accessible from more than one thread at run-time. Escape analyses (e.g., Ruf's) can determine when objects are accessed by at most one thread, and these objects cannot participate in an interference dependence. In Figure 1, typical escape analyses will mark `newAcct` in `Wife.run()` as holding only thread-local objects. This allows the elimination of a spurious edge between line 21 and line 64 generated in the naive method. However, Ruf's escape analysis would mark `checking` and `savings` in `Wife.run()` as escaping since they are both accessed in the constructor for `Wife` which is executed by the `main` thread – *not* the `wife` thread. This means that the edge between line 21 and line 59 cannot be removed.

Leveraging alias information: Alias information can be used to prune edges relating `checking.balance` and `saving.balance`. For example, the knowledge that the primaries in line 21 and line 59 are not aliases can be used to remove the interference edges between these lines. One of our several modifications that we give to Ruf's analysis is an extension that uses what we call *share entities* to detect such information (see Section 4.1).

2.3 Ready Dependence

In simple words, a statement m is ready-dependent on a statement n if the execution of m can be infinitely delayed due to the fact that n fails to complete its execution and n is a synchronization related construct. This notion of dependence is relevant when using slicing to generate reduced models for checking temporal properties (specifically, liveness properties) [13]. We shall use `notify()` to denote both `notify()` and `notifyAll()` methods in `java.lang.Object` and `wait()` to denote all overloaded versions of `wait()` in `java.lang.Object`.

Definition 2. *If m and n are program points in a given Java program then m is ready dependent on n if any one of the following is true.*

1. *m and n occur in the same thread and m is reachable from n and n is an* ***entermonitor****.*
2. *m and n occur in the same thread and m is reachable from n and n is an invocation of* ***wait()****.*

[3] In `a.f` and `a[b]`, a is referred to as the primary.

3. *m and n occur in different threads and m is an* ***entermonitor*** *statement on object o and n is an* ***exitmonitor*** *statement on object o.*
4. *m and n occur in different threads and n is an invocation of* ***notify()*** *on object o and m is an invocation of* ***wait*** *on object o.*

Condition 1 and 2 are uninteresting for us as they can be dealt with a simple sequential control flow graph. Condition 3 and 4 provide interesting cases as they span across threads. From here on, by 'ready dependence' we mean 'ready dependence resulting from conditions 3 and 4 only'. The techniques we consider for pruning interference dependences can also be used to prune ready dependence (e.g., if an object o is not shared then it cannot generate a ready dependence via conditions 3 or 4).

In Figure 1, a naive type-based ready dependence calculation as implemented previously in Bandera[9] will report ready dependence edges between line 19 and both line 57 and line 61. However, the latter edge is a spurious one which cannot be eliminated by escape information. However, it can eliminated using alias information computed using what we call *ready entities* as described in Section 3.

3 Details of the Analysis

Although there are numerous escape analyses, we choose to work with Ruf's escape analysis because it is relatively easy to understand, straightforward to implement, scales well, and its particular strategy for identifying thread-local data yields more precise results than several other approaches (see [28] for details).

Our goal is to create an analysis that calculates information that will enable the following questions to be answered.

- Given a field/array read expression m and a field/array write expression n, is m interference dependent on n?
- Given a `entermonitor` statement m and a `exitmonitor` statement n, is m ready dependent on n under condition 3 in the definition of ready dependence?
- Given a `wait()` call-site m and a `notify()` call site n, is m ready dependent on n under condition 4 in the definition of ready dependence?

The analysis proceeds in three phases. The first phase collecting information about the system to be used during phase two and three. In phase two, intra-procedural analysis happens along with bottom-up(in the call graph) interprocedural analysis in an interleaved fashion. In phase three, the information from the callers is propagated to the callees. After presenting the details of the analysis, we summarize how it differs from Ruf's original version.

3.1 Alias sets and contexts

Each program variable v is associated with an *alias set* – an abstract object that summarizes properties of concrete objects that may be referenced by v at

run-time. An alias set for v is either $\perp$ (indicating that only null reference values are assigned to v at run-time, or that v is of non-reference type) or a tuple of properties as follows

aliasSet ::= $\perp$ | $\langle$*global, fieldMap, accessed, escapes, waits, notifies, readyEntities*$\rangle$.

The elements of the tuples are described below.

global is a boolean that when *true* indicates that the abstract object is reachable via static fields.

fieldMap associates each field f of the abstract object with an alias set abstracting the concrete objects that may be referenced by f at run-time, i.e., it maps fully quantified field names to alias sets. `$ELT` is a special field used to represent all cells in a dimension of an array.

accessed is a boolean that when *true* indicates that the abstract object was read or written.

escapes is a boolean that when *true* indicates that the abstract object is accessed by more than one thread.

waits is a boolean that when *true* indicates that the abstract object received a `wait()`.

notifies is a boolean value that when *true* indicates that the abstract object received a `notify()` or `notifyAll()`.

readyEntities is a set of entities drawn from a domain E. It forms a finer partition than the alias sets. This entity set indicates the participation of the objects represented by the alias set in wait/notify relationship (leading to a ready dependence). Specifically, if the intersection of the ready entities for some alias sets a_1 bound to variable v_1 and a_2 bound to v_2 is non-empty, then invocations of `wait`/`notify` on v_1 and v_2 can lead to a ready dependence.

As the analysis manipulates alias sets, certain invariants representing consistency properties must be enforced. If an alias set has *global=true*, then all alias sets reachable via *fieldMap* should have *global=true* (indicating that if an object o is reachable from more than one thread, then the objects referenced by o's fields are as well). If $global = true$, then $escapes = true$. Finally, if $readyEntities \neq \emptyset$ then $escapes = true$.

Upon creation of an alias set, all boolean elements of the alias set are set to *false* except for *accessed* which is set to *true* [4] *readyEntities* is set to $\emptyset$ and the *fieldMap* is empty. Alias sets of fields are created on demand. Alias sets are operated on by *clone*, *unify*, and *unifyAtStart* operations.

The *clone* operation creates a new alias set isomorphic to the existing one. As a space optimization, when cloning alias sets marked as global, a reference to the existing alias set is returned. In addition, the elements of *readyEntities* are always shared when cloning.

[4] The mere existence of the alias set can be substituted for the truth value of *accessed*. Hence, this field is redundant, however, it is used as it makes explanation of the analysis more intuitive.

Domain

$v \in V$ set of variables
$f \in F$ set of fields
$m, p \in M$ set of methods
$a, r, e \in A$ set of alias sets
$mc, sc \in C$ set of alias contexts
$s \in CS$ set of call sites

Mappings

$AS : V \rightarrow A$ alias set lookup
$MC : M \rightarrow C$ method context lookup
$CALLEES : M \times V \rightarrow \wp(M)$ callee lookup
$SCC : M \times \wp(M)$ SCC lookup
$MULTI\_EXEC : CS \rightarrow \{true, false\}$ multiply executed call-site lookup

Rules

statement	action
$v_1 = (t)v_2$	$unify(AS(v_1), AS(v_2))$
$v_1 = v_2.f$	$unify(AS(v_1), AS(v_2).fieldMap(f))$
$v_2.f = v_1$	
$v_1[] = v_2$	$unify(AS(v_1).fieldMap(\$ELT), AS(v_2))$
$v = v_1[]$	
return v	$unify(AS(v), AS(r))$
throw t	$unify(AS(t), AS(e))$
$v_r = v_t.m(a_1, \ldots a_n)$	*let* $sc = \langle AS(v_t), \langle AS(a_1), \ldots AS(a_n)\rangle, AS(v_r), e\rangle$ $\forall p \in CALLEES(m, v_t).$ *let* $mc = MC(p)$ *and unifier* = *unify* *if* m = *java.lang.Thread.start then* *unifier* = *unifyStart* *if* $SCC(m) = SCC(p)$ *then* *unifier(sc, mc)* *else* *unifier(sc, newInstance(mc))* *if* $MULTI\_EXEC(v_t.m(a_1, \ldots a_n))$ *and* m = *java.lang.Thread.start then* *unifier(sc, sc)*

Fig. 2: Domains, mappings and rules used in intra-procedural analysis. In the rules, m represents the method in which the rules are being applied and e is the alias set corresponding to the exceptions thrown in m.

The *unify* operation merges the information represented by elements of the tuple (except for *escapes* and *readyEntities* as explained below). The unification of the boolean and map values is defined as the join under the boolean lattice (with *true* as the top element) and function lattice, respectively. The domain of the resulting *fieldMap*s is the union of the domains of the maps being unified. The alias sets of the field names occurring in the domains of both the maps are (recursively) unified, as well.

The *escapes* and *readyEntities* elements represent information that spans threads, and thus these elements should only be merged at thread start sites. The *unifyAtStart* operation implements this distinct form of merging. Intuitively, an object should be considered escaping when it is accessed in more than one thread. To capture this, the unification of the *escapes* element is defined as the join under the boolean lattice when either one of the arguments is *true* (i.e., if objects are

arg1.accessed	**arg2.accessed**	**result.escapes**
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Table 1: Rules to unify *escapes* under *unifyAtStart* operation when **arg1.escapes** = **arg2.escapes** = *false*.

already marked escaping in at least one of the alias sets being merged, then they should continue to be considered as escaping). However, when the *escapes* field in both the alias sets being unified is *false*, the value of *escapes* in the merged alias set is given by Table 1 (intuitively, the arguments to *unifyAtStart* are alias sets coming from two different threads, and the resulting merged alias set should only be marked as escaping (i.e., shared) if *accessed* is true from both threads).

The result of unifying *readyEntities* is determined by the *waits* and *notifies* elements of the alias sets being unified. The *readyEntities* of the alias sets being unified is modified only when *waits* element of one alias set is *true* and the *notifies* element of the other alias set is *true*. In this case, the *readyEntities* set is modified by injecting a fresh entity into the *readyEntities* of the alias set occurring in the site context (introduced below). In all other cases, *readyEntities* set is unmodified in the result of unification.

An *alias context* is an aggregate data structure used to represent the flow of information into and out of a method. An alias context is a tuple of alias sets corresponding to `this`, method arguments $(a_1, a_2, a_3, \ldots)$, return value(r), and the exceptions thrown by the method (e).

$$aliasContext ::= \langle this, \langle a_1, a_2, a_3, \ldots \rangle, r, e \rangle$$

Like alias sets, alias contexts support *clone* and *unify* operations. *clone* creates a new alias context isomorphic to the existing one. The *unify* operation is a point-wise extension of alias set unification to alias contexts tuples. For the sake of simplicity, we use the the term *method context* to indicate the alias context of a method at its entry point and the term *site context* to indicate the alias context at a call-site.

Both alias set and alias context are implemented as union-find data structures [29].

3.2 Phase One

The analysis begins by forming abstract representatives for groups of one or more concrete threads. The abstract representative of a group of threads is a call graph rooted at the start site of each thread. Thus, there is an abstract thread for each start site, and this abstract thread represents all concrete threads started at that site. The complete collection of such call graphs for a program is called a *threaded call graph*.

The threaded call-graph is constructed as follows. First, a value-flow-based framework is used to construct a traditional call graph [27]. Strongly-connected components (SCC) of the call graph are calculated. This is followed by identifying call-sites which invoke `java.lang.Thread.start()`. If any of these call-sites can be executed more than once during the life-time of the system, (i.e., if they occur in a method called more than once, if the method occurs in a SCC of size greater than one, or the call-site is enclosed in a loop), then the associated call-graph is marked as possibly representing more than one concrete thread. The intuition is that if an object is thread-local to an abstract thread representing multiple concrete threads then one must assume that it can be shared between multiple concrete threads created at the same start site – the analysis cannot distinguish between threads started at the same `start`-call site.

3.3 Phase Two

This phase starts off by creating alias sets with *global=true* for static fields. It then performs an interprocedural analysis by processing each method in each SCC in a bottom-up fashion on the call graph. The processing of the methods includes the creating of the method context and performing flow-insensitive intra-procedural analysis on the method as described below. The effect of this phase is to accumulate information about object accesses in the call-graph of a particular thread t, bubble this information up, and expose it at the start call site for a particular thread.

Intra-procedural analysis Figure 2 presents the rules for processing the statements of a method during intra-procedural analysis. These rules ensure that the aliasing of data inside the method is represented appropriately in method contexts and global alias sets. In a method, all statements not containing calls to `start()` are processed before processing statements containing calls to `start()` to capture the effect of `post-start()` site data accesses at `start()` sites. During processing, if an alias set for a variable does not exist, a new instance is created.

The interesting rule is the one for method invocation. At a call-site, if the caller and the callee exist in the different strongly connected components in the call graph then the method context of the callee is cloned and the clone is unified with the site context to achieve context sensitivity. When both caller and callee occur in the same SCC, the method context of the callee is unified with the site context to achieve an effect similar to simple fixed-point iteration.

The rule for method invocation varies the alias set unifier depending on the method being invoked. For method calls other than `start()`, the same thread executes the caller and the callee (thus *unify* is used as the unifier operation). For `start()`, different threads execute the caller and the callee, and thus *escapes* in the alias sets needs to be updated (thus *unifyAtStart* is used as the unifier operation). As *readyEntities* depend on *waits* and *notifies* of alias sets representing values in different threads, they are also updated via *unifyAtStart*. If the call-site invokes `start()` and is determined to be executed multiple times,

the site context is unified with itself via *unifyStart* to account for the effect of multiple executions as described in Section 3.7.

3.4 Phase Three

In this phase, the call hierarchy is traversed in a top-down fashion. This ensures that any changes to the information accumulated by the parent thread are exposed to children threads.

Call-sites are the only points of processing in this phase. At each call-site encountered in the top-down traversal of the call graph, each alias set in each method context is visited recursively. On visiting an alias set of a method context, the *escapes* element of the alias set is joined with the *escapes* element of the alias set's counterpart in the site context. As for *readyEntities*, the values are not joined (which would destroy context sensitivity), but rather values from the site context alias sets are injected into the *readyEntities* of method context alias sets.

3.5 A comparison to Ruf's analysis

We now summarize how our escape analysis differs from the original version given by Ruf.

- Ruf's analysis is geared towards the removal of unnecessary synchronization operations. Hence, Ruf's alias sets have elements *synchronized* and *syncThreads* to capture information indicating whether or not objects represented by the alias set are used in synchronization and the number of threads that synchronize on the objects, respectively. However, the alias sets in our analysis have other fields tailored to calculate escape information and dependence information.
- Unlike Ruf's analysis, we have used two versions of unification (*unify* and *unifyAtStart*), and the form that is used at `start()` call-sites alters the escape information while the other form does not.
- The unification rules in Ruf's analysis mark an alias set as synchronized only at synchronization expressions. However, in our analysis, if a variable occurs in an access expression then the associated alias set is marked as accessed and the decision to mark the alias set as escaping is deferred to our new *unifyAtStart* unification operation that is used at `start()` call-sites.

3.6 Example

When the analysis is applied to the program of Figure 1, Phase 1 is uninteresting since there aren't any `start()` call sites that will be executed multiple times in this simple program.

Phase 2 starts by processing `Acct.Acct()`, `Wife.Wife(Acct)`, `Man.Man(Acct)`, `Wife.run()`, `Man.run()`, and `Home.main()`. On processing `Acct.Acct()`, a method

context $\langle \alpha_0, \langle\rangle, \alpha_0, \bot \rangle$[5] is created where and $\alpha_0 = \langle \mathit{false}, \langle\rangle, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$ where α_0 is a newly created alias set[6]. Note that the *accessed* element of α_o is set to *false* as a result of WHB optimization.

The method context at the start of the processing of `Wife.Wife(Acct)` will be $\langle \alpha_1, \langle \alpha_2 \rangle, \alpha_1, \bot \rangle$ in which α_1 and α_2 are newly created alias sets. After processing the method, $\alpha_1 = \langle \mathit{false}, \{\mathit{savings} \rightarrow \alpha_2, \mathit{checking} \rightarrow \alpha_3\}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$ where α_3 is a newly created alias set and $\alpha_2.\mathit{accessed} = \mathit{false}$. A similar method context $\langle \alpha_4, \langle \alpha_5 \rangle, \alpha_4, \bot \rangle$ occurs for `Man.Man(Acct)` where $\alpha_4 = \langle \mathit{false}, \{\mathit{savings} \rightarrow \alpha_5, \mathit{checking} \rightarrow \alpha_6\}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$.

Upon processing `Wife.run()`, it's method context will be $\langle \alpha_7, \langle\rangle, \bot, \bot \rangle$ where $\alpha_7 = \langle \mathit{false}, \{\mathit{savings} \rightarrow \alpha_8, \mathit{checking} \rightarrow \alpha_9\}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$, $\alpha_{8/9} = \langle \mathit{false}, \langle\rangle, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \emptyset \rangle$. The local variable *newAcct* in `Wife.run()` is associated with α_{10}, however, this alias set is not visible from the method context. We omit the details of these alias sets from here on due to space constraints. A similar method context occurs for `Man.run()` except that the *notifies* element in the alias sets of `savings` and `checking` fields is set to *true* and *false*, respectively.

The processing of the line 27 creates a new alias set, α_{10}, associated with the local variable `savings`. The processing of line 28 associates `wife` with $\alpha_{11} = \langle \mathit{false}, \{\mathit{savings} \rightarrow \alpha_{10}, \mathit{checkings} \rightarrow \alpha_{12}\}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$. The processing of line 30 will unify $\langle \alpha_{11}, \langle\rangle, \bot, \bot \rangle$ with an isomorphic copy of the method context of `Wife.run()`. This will result in *accessed* and *wait* in α_{10} being set to *true* due to unification with an isomorphic copy of α_8. By the unification rules in Table 1, *escapes* element of α_{10} is set to *true*.

The processing of line 31 will unify α_{10} with an isomorphic copy in which *accessed* is set to *true*, hence, based on the unification rules at `start()` call-sites $\alpha_{10}.\mathit{escapes}$ is set to *true*. Also, as *notifies* will be *true* on the isomorphic copy, by the unification rule for *readyEntities*, a new object is inserted into $\alpha_{10}.\mathit{readyEntities}$ indicating that `wait()` and `notify()` on `saving` should be considered for ready dependence.

3.7 Allocation sites executed multiple times

Call-sites and allocation sites that are executed multiple times pose a major obstacle for static analyses such as escape analysis.

For example, if `Home.mainAlt()` was considered as the entry point to the system, then the abstract thread corresponding to the thread allocation site in the loop at line 37 in Figure 1 represents more than one run-time thread object. In such situations and also when `start()` call-sites are executed multiple times, Ruf's analysis will provide pessimistic information by marking all reachable alias set from such site contexts as shared.

However, upon unrolling the loop once and analyzing using the rules given earlier, each alias set reachable from the site context will be unified with itself.

[5] The alias set, α_0, indicates that the constructor returns a value to signify the creation of the object and the invocation of the constructor on it.

[6] We use the term *newly created alias set* to indicate an alias set unaltered after creation.

Hence, in our analysis, we use this observation in Phase 3 to unify the site context with itself via *unifyStart* before propagating the information across a `start()` call-site that is executed multiple times (note that due to the definition of unification on escape and ready information, unifying an alias set with itself does not necessarily correspond to the identity function). This captures the effect of the loop on the flow of information. This is not done at other call-sites as they all occur in the same thread, hence, the optimization cannot affect escape information.

3.8 Complexity

The algorithm visits each node in the call graph twice and at each node it visits the statements of the methods once. The worst case processing time for unification is dictated by the recursive nature of the data structures used in the program, in particular when propagating information across method boundaries. Hence, the worst case time complexity for the analysis should be in the order of $O(N + S)$ where N is the number of nodes in the call graph and S is the total number of statements in the system if time complexity of object construction is considered as a constant.

New instances of alias sets are created at method call sites. Hence, a data structure with m fields being passed across n call sites at each level of a call chain of length p can lead to alias sets whose accumulated size is in the order of $O(m * n^p)$. However, in reality, large data structures are recursive in nature rather than a single big chunk. Hence, reaching worst case space complexity should be a rarity.

4 Transformations

We now describe how the information computed by our escape analysis can be used in slicing and other applications.

Our slicer implementation proceeds in two phases: an initial phase collects various dependence information, and the second phase follows the dependence edges to calculate the program slice. As we have indicated, the results of our analysis can be used to prune interference dependence edges between non-escaping primaries. Also, ready dependence edges can be pruned when the *readyEntities* of the alias sets corresponding to the receivers of `wait()` and `notify()` are disjoint.

An interesting by-product of this analysis is detection of buggy `wait()`s and unnecessary `notify()` calls in the system. From the semantics of `wait()` and `notify()` it can be inferred that the receiver object should be shared across threads for these calls to be safe and useful, respectively. Hence, if *waits* and/or *notifies* element of an alias set are set to *true* and *escapes* is set to *false*, then the corresponding invocations can be concluded as buggy and unnecessary. In the former case, the developer can use this information to make the program

safe by avoiding infinite waits. In the latter case, the corresponding invocations of `notify()` can be removed to improve run-time performance.

As with other escape analyses, we can use the result of our analysis to remove unnecessary synchronization and stack allocate objects. Both optimizations can be triggered when the *escapes* element of the alias set associated with the participating variable is set to *false*.

4.1 Increased precision using *share entities*

Detection of interference based on escape information is still pessimistic as it may be the case that two field access expression may have the same signature but may correspond to different run-time primaries. This can be remedied to some degree by extending the alias set with the following three elements.

read is a boolean that when *true* indicates that the value was read,
written is a boolean that when *true* indicates that the value was written, and
shareEntities is a set of object entities that represents objects which participate in the interference dependence. It is similar to *readyEntities* in meaning.From the nature of interference dependence, $shareEntities \neq \emptyset \Rightarrow escapes = true$.

Similar to the unification rule of *readyEntities*, the *shareEntities* of the alias sets being unified is modified only when *read* element of one alias set is *true* and the *written* element of the other alias set is *true*. In this case, the *shareEntities* set is modified by injecting a fresh entity into the *shareEntities* of the alias set occurring in the site context. In all other cases, *shareEntities* set is unmodified in the result of unification.

With this information, access expressions can be considered for interference dependence only when the *shareEntities* of their corresponding alias sets are not disjoint. This information can also be used to improve the precision of ready dependence based on condition three of Definition 2.

5 Experimental Results

We have implemented all variants of the analysis described in the previous sections in the the Indus Slicer component of the Bandera tool set [16]. Soot [30] was used as the intermediate representation. Unlike Ruf's analysis, we use *local variable splitting* transformation of Soot instead of SSA representation to ensure that variables are defined only once.

We summarize the results of applying the analysis to the Java Grande[17] benchmark suite. Java Grande provides three classes of applications: low-level operations, kernels, and large scale applications. In our experiments, we considered each method with the signature `public static void main(String[])` as representing a possible entry point into the system.

Table 2 presents the results of applying the previously described escape analysis followed by the generation of interference dependence edges. We not give timing data in the table, *since all the runs completed in less than 3 seconds*

Program Names	Reachable methods	Interference edges based on Type	Escape info	Entity info
JGFBarrierBench	117	117	29	29
JGFForkJoinBench	121	1929	155	16
JGFSyncBench	87	122	36	36
JGFCryptBenchSizeA	110	1228	180	58
JGFLUFactBenchSizeA	139	3624	181	24
JGFSORBenchSizeA	143	2318	162	23
JGFSeriesBenchSizeA	136	2138	157	8
JGFSparseMatmultBenchSizeA	144	3833	183	14
JGFMolDynBenchSizeA	146	8026	373	209
JGFMonteCarloBenchSizeA	394	5245	309	11
JGFRayTracerBenchSizeA	157	1330	171	166

Table 2: Results of analysis and interference dependence generation applied to Java Grande benchmarks.

on 2.5GHz machine with 2GBs of RAM using Sun JDK1.4.2 with a maximum heap-size of 1GB. Note that this timing data does not include the cost of parsing, basic control-flow analysis and call-graph construction; the call-graph and threaded-graph construction complete in less than 100 milli-seconds for each example.

The "Type" column in the table contains the baseline data for our experiments – it shows the number of interference edges created based on the simple field signature and value type-compatibility approach described in Section 2.2. The "Escape Info" column shows the number of interference edges detected using escape analysis *without share entities* optimization. These results indicate an order of magnitude reduction of the simple previous approach. The "Entity Info" column shows the number of interference edges detected using escape analysis along *with share entities* optimization of Section 4.1. These results indicate that the simple extension of using entities to form relationships between read/writes can dramatically improve the precision of the analysis with marginal increase in time and space. We believe that this optimization can be incorporated in other variants of escape analysis to obtain more precise results. Given the low cost (time) of the analysis, the results provide convincing evidence that escape information should be leveraged in slicing concurrent object-oriented programs.

6 Related Work

Numerous escape analyses have been proposed to calculate if an object escapes (i.e., can be accessed outside of) a particular method m and/or thread t. Ruf's equivalence-class-based analysis [28] for calculating if an object is only accessed in a single thread and Choi et.al.'s fixed-point-based analysis [6] to calculate if an object escapes a method or a thread are two well-known escape analyses. Aldrich et.al. [1], Blanchet [2], and Bogda and Hölzle [3] also propose similar analyses

to improve runtime performance of Java programs by removing unnecessary synchronization and to enable stack allocation of objects.

Similarly, numerous data-race detection algorithms have been proposed. Choi et.al. propose an on-the-fly technique[7] to detect data-race conditions (i.e., situations where accesses to shared objects are not protected by locks). This requires instrumentation and various optimizations such as static data-race detection analysis as a pre-phase to the dynamic analysis [8]. Flanagan and Freund proposed a annotated type-based technique [10] along with an annotation inference mechanism which is similar to escape analysis [11] for the purpose of detecting race-condition by ensuring objects annotated/marked as thread-local are indeed thread-local (i.e., through the entire execution, they are only reachable from a particular thread t). A similar sort of analysis presented by Boyapati and Rinard [4] proposes extensions to Concurrent Java [10] and uses the auxiliary information to deduce escape information.

There is a wide body of literature on slicing sequential programs, beginning with Weiser's original paper on slicing [31]. Horwitz et.al[15] proposed a interprocedural program slicing algorithm which has been extended by others to handle various features such as exceptions and unconditional jumps. There has been effort ([20, 14, 22, 21]) in addressing issues involved in slicing program written in object oriented languages with features such as pointers and/or references.

In the realm of slicing concurrent programs, Ramalingam[26] proved the calculation of context-sensitive synchronization-sensitive program slices is undecidable. Later on, Müller-Olm[23] provided tighter lower bounds for slicing synchronization-free programs in the absence of procedures and loops.

Krinke[18] considered intra-procedural slicing for a simple **while**-language with **co-begin/co-end** statements and proposed a form of symbolic execution to prune interference dependences starting from the observation that considering interference dependences to be transitive is overly conservative. Nanda[24] proposed algorithms that he described as context sensitive for slicing concurrent programs. Recently, Krinke[19] proposed a context-sensitive algorithm for slicing concurrent programs. However, these algorithms rely on symbolic execution and/or *may-happen-in-parallel(MHP)* [1, 25] information to prune interference dependences. However, in the absence of synchronization operations, the use of MHP algorithms will not provide large reductions in the dependences as it is harder to detect instruction execution ordering. Also, these techniques does not address pruning of interference dependences arising from language features such as dynamically created objects and threads. As the justification for pruning interference dependences are independent (thread locality of objects vs instruction execution ordering), orthogonal pruning techniques such as ours and those mentioned above can be combined to obtain further reductions.

7 Conclusion

We have demonstrated how an existing escape analysis can be adapted and extended to produce information that is useful for reducing the number of edges

in dependence graphs used in slicing concurrent Java programs. In addition, the simple addition of what we called entity information to the analysis yields dramatic improvement in precision with very little cost. We believe that leveraging escape information in this way is absolutely necessary for scaling slicing of concurrent object-oriented programs to larger code bases.

References

1. J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of Static Analysis Symposium (SAS'99)*, pages 19–38, 1999.
2. B. Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.
3. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999.
4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, FL, USA, Oct 2001.
5. Z. Chen and X. Baowen. Slicing concurrent java programs. *SIGPLAN Notices*, 36(4):41–47, April 2001.
6. J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for object oriented languages. application to Java. In *Proceedings of Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, USA, Oct 1999. ACM.
7. J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, June 2002.
8. J. D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000.
10. C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.
11. C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, second edition, 2000.
13. J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Lecture Notes in Computer Science*, Sept 1999. Proceedings on the 1999 International Symposium on Static Analysis (SAS'99).

14. S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 28–40. ACM, 1989.
15. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI'88)*, volume 23, pages 35–46, 1988.
16. Indus, a toolkit to customize and adapt java programs. This software is available at http://indus.projects.cis.ksu.edu.
17. Java grande forum benchmark suite - thread version 1.0. This software is available at http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/. Java Grande Benchmarking Project at Edinburgh Parallel Computing Centre.
18. J. Krinke. Static slicing of threaded programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).
19. J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of ESEC/SIGSOFT FSE'03*, 2003.
20. L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of International Conference on Software Engineering (ICSE'96)*, pages 495–505, 1996.
21. D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of International Conference on Software Maintenance (ICSM'98)*, pages 358–367, 1998.
22. P. Livadas and A. Rosenstein. Slicing in the presence of pointer variables, 1994.
23. M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing (STOC'01)*, pages 647 – 656, Hersonissos, Greece, 2001. ACM, ACM Press.
24. M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 180–190, 2000.
25. G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent java programs. Technical Report UM-CS-1998-044, University of Massachusetts, Amherst, October 1998.
26. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416 – 430, March 2000.
27. V. P. Ranganath. Object-flow analysis for optimizing finite-state models of java software. Master's thesis, Kansas State University, 2002.
28. E. Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 203–213, June 2000.
29. A. V.Aho, R. Sethi, and J. D.Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley International, 1986.
30. R. Vallée-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, School of Computer Science, McGill University, Montreal, Canada., Oct 2000.
31. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
32. J. Zhao. Slicing concurrent Java programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 126–133, May 1999.