

Towards a Better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs

Jun Chen

David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
j2chen@uwaterloo.ca

Steve MacDonald

David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
stevem@uwaterloo.ca

ABSTRACT

Testing concurrent programs remains a difficult task due to the non-deterministic nature of concurrent executions. Many approaches have been proposed to combine static and dynamic analysis to reduce the complexity of uncovering potential concurrency bugs. However, the existing collaboration schemes only provide a limited mechanism for exchanging relevant information between the two analyses. For example, alias information only flows from the static analysis module to the dynamic analysis module at the beginning of the dynamic analysis. Therefore, we cannot fully exploit the advantages of each type of analysis. Motivated by this observation, in this paper we present a new testing technique which enables a tighter collaboration between static analysis and dynamic analysis. In this collaboration scheme, static analysis and dynamic analysis interact iteratively throughout the whole testing process. Static analysis uses coarse-grained analysis to guide the dynamic analysis to concentrate on the relevant search space, while dynamic analysis collects concrete runtime information during the guided exploration. The runtime information provided by the dynamic analysis helps the static analysis to refine its coarse-grained analysis and provides better guidance on dynamic analysis. Currently, our implementation consists of a static analysis module based on Soot and a dynamic analysis module based on JPF (Java PathFinder).

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.2.5 [Testing and Debugging]: Debugging Aids

General Terms

Languages, Verification, Reliability

1. INTRODUCTION

Concurrent programs are much harder to debug than sequential programs mainly due to the non-deterministic

nature of their execution. Individually, static analysis and dynamic analysis could be used to uncover hidden concurrency bugs. For example, static analysis could perform escape analysis to determine shared instance fields in a program, then type analysis can determine the consistency of lock protection using either derived or annotated locksets. Static analysis can uncover bugs without executing the program and avoids the non-determinism problem altogether. Dynamic analysis captures bugs by running a program directly. To uncover concurrency bugs, dynamic analysis tries to trigger different interleavings in test runs. This could be achieved by randomly seeding sleep statements [8] or systematically exploring all interleavings using an explicit-state model checker [21].

However, both static analysis and dynamic analysis have their limitations when they work alone. Static analysis tends to report a high number of false positives due to the coarse-grained nature of the analysis. The main cause of this high false positive rate is that static analysis has an overall knowledge of the program under analysis but has imprecise information with respect to a particular control flow or interleaving. It may determine two accesses *may-alias* when considering the overall program. However, it generally lacks the ability to determine in which control flows those two accesses will actually occur. Therefore, static analysis will treat those accesses as always *may-alias*, even if no feasible control flow at runtime will fulfill this alias relationship.

Dynamic analysis suffers from the coverage issue because the triggering of interleavings for all possible distinct partial orderings requires excessive computational resources. A random triggering can never guarantee that all partial orderings are tested. A systematic exploration using an explicit state model checker can lead to the program state explosion problem. The main cause of this problem is that dynamic analysis has precise local (a particular control flow or interleaving) knowledge of the program, but little to no overall knowledge of the program. In other words, dynamic analysis can report precise program facts about an executed program interleaving, such as the aliasing relationships between two accesses, but it generally lacks the ability to apply the collected information to further exploration. To apply the collected information globally, dynamic analysis needs to know two things: what other feasible partial orderings might also exist in the program, and how those partial orders could be triggered based on what has already been tested.

Due to their respective advantages and limitations, many existing techniques have tried to combine static analysis and dynamic analysis to leverage the advantages of both. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-052-4/08/07 ...\$5.00.

most common collaboration scheme is to have static analysis identify a set of relevant artefacts that will help guide dynamic analysis. For example, the static analysis provides a set of instructions that access variables that are shared by multiple threads using alias and MHP (May Happen in Parallel) analysis. Those instructions serve as markers that could trigger an interleaving for an unexplored partial ordering if permuted during runtime. In general, static analysis provides some guidance to dynamic analysis on what should be checked while dynamic analysis ensures a low false positive rate by running the relevant interleavings. It is important to note that in this scheme, the information flow is uni-directional, from static analysis to dynamic analysis. Moreover, this collaboration only happens once, at the beginning of static analysis.

We believe this uni-directional and one-time collaboration has not fully exploited all possible collaboration opportunities between static and dynamic analysis. Static analysis can tell dynamic analysis more than which instructions should be considered for permutation. Static analysis can also tell the dynamic analysis how to efficiently test these permutations at runtime, taking advantage of what has already been tested. Moreover, we can leverage the precise interleaving-specific information collected by the dynamic analysis to improve the quality of the static analysis, by taking the flow of control in a program into account. In both cases, the static analysis has to be performed more than once in the testing process. More importantly, the information flow between static analysis and dynamic analysis is now bi-directional.

In this paper, we present the implementation of this proposed collaboration scheme, based on our previously-developed testing technique for concurrent programs called value-schedule-based testing [6] [7]. In value-schedule-based testing, we use static analysis to construct fulfilling interleavings for value schedules [3] of a concurrent program, then check the feasibility and correctness of a value schedule by executing its fulfilling interleavings in an explicit state model checker. In previous work, we used static analysis to guide the model checker to carry out possible relevant permutations [21]. This process is easily extended to incorporate our proposed collaboration scheme. For example, the alias and control flow information collected from the controlled execution in the model checker can be used to refine the original program facts computed by static analysis. Then, the refined program facts could be used to help static analysis guide the model checker to perform further exploration. More importantly, by adding our new collaboration to a tool that already included some collaboration between static and dynamic analysis, we could concretely demonstrate how various collaborations could be integrated. Hopefully, our work will be able to shed light onto other potential collaborations between static and dynamic analysis.

This paper is organized as follows. In Section 2, we motivate our new collaboration scheme with a concrete example. Then, an overview of our new collaboration scheme is introduced in the context of our value-schedule-based testing technique. We present and discuss the implementation of our new collaboration scheme in Section 3. In Section 4, we evaluate the benefits of our proposed collaboration between static and dynamic analysis with a case study. Related work is discussed in Section 5. Finally, the conclusion is given in Section 6.

2. OVERVIEW

In this section, we start by showing a concrete example where the testing process could benefit from improved collaboration between static and dynamic analysis. Then, we briefly present the concept of value-schedule-based testing for concurrent programs using the sample code in Figure 1. Moreover, we will describe some of the basic components of value-schedule-based testing as shown in Figure 2.

2.1 Motivation

We motivate our paper by showing examples in which increased collaboration between static and the dynamic analysis could improve the testing experience.

Currently, dynamic analysis uses the analysis output from MHP and alias analysis to determine whether an instruction should be permuted with other instructions at runtime. An instruction accessing a variable will be permuted if it forms a relevant MHP pair with another access that uses the same variable. However, such a MHP pair may not be relevant at runtime. First, the MHP relationship between two instructions is coarsely determined by static analysis using an imprecise control flow graph. One of two instructions in a MHP pair could be unreachable at runtime. Second, the evaluation to determine if two instructions access the same variable is based on coarse-grained may-alias analysis. This analysis may not be able to determine if the variable accessed by two MHP instructions must-alias each other or must-not-alias each other, instead indicating the two variables may-alias one another. For testing purposes, we must assume such variable are aliases so dynamic analysis will still perform the permutation. If these variables are not aliases at runtime, the permutation is unnecessary and results in the execution of extraneous tests. A closer examination reveals that both the concrete control flow graph and alias relationships could be captured and verified by the dynamic analysis tool at runtime. The difficulty lies in how to instruct the dynamic analysis to check the relevancy of each MHP relationship in time and prune out the unnecessary permutations as early as possible. We believe we can use static analysis to generate appropriate exploration paths which can then be used to guide the dynamic analysis to check such information.

For example, from the program in Figure 1, static analysis will report the read of *ta1.val* at line 27 and write of *ta1.val* at line 43 form a relevant MHP pair, and the read of *val* following the method invocation at line 30 and the write at line 45 as another relevant MHP pair. Existing techniques such as [21] [12] will permute them to trigger different partial orderings. Once a permutation is done, both diverging interleavings are executed to completion. However, neither will produce different partial orderings, as we will demonstrate.

If we can schedule dynamic analysis to follow an instruction path where both accesses are about to be executed by their respective threads, we can easily see that the read of *ta1.val* at line 27 and the write of *ta1.val* at line 43 are actually accessing non-aliased objects by comparing the memory locations of two accesses before running the instructions. Line 26 redefines *ta1* to a new object of *TypeA*. Moreover, in the case of the read of *ta1.val* following the invocation at line 30 and the write at line 45, we cannot construct an execution path in which these two instructions are both run by their respective threads because no feasible path exists.

This is because line 29 overwrites *ta1* in thread *T1* to an object of *TypeB*, which does not access the instance variable *val*. Since no such path exists, there are no relevant permutations between those two instructions that could trigger different partial orderings. Thus, the problem is transformed into finding a way to determine such infeasibility. For our sample pair, this infeasibility could be checked by determining the runtime type of *ta1* just before the method call at line 30 is executed and performing a coarse-grained reachability analysis based on this type. This analysis will determine that *TypeB.op1()* is run and that this method does not modify the instance variable *val*. Thus, permuting the instructions at line 30 and 45 is unnecessary.

To ensure no spurious permutation is executed at runtime, we have to repeatedly switch between the static analysis tool and the dynamic analysis tool. The static analysis decides what to check in the dynamic analysis (relevant MHP pairs) and how to check it (instruction paths). Then, the dynamic analysis performs the checking using the instruction paths provided by static analysis and reports back the relevant program state information, such as concrete alias information and call graph edges, to validate a permutation. Once a permutation is determined to be necessary, the static analysis extends its analysis to check the next possible permutation. Otherwise, the static analysis will instruct the dynamic analysis to abandon further execution along the current path. As permutations are incrementally checked, the different partial orderings of concurrent accesses in the programs are dynamically derived and tested as well. During the testing process, the information transfer between two analyses is bi-directional and happens repeatedly. This paper will concentrate on discussing the type of data that is obtained and exchanged between static and dynamic analysis tools. Before we go into the details of this topic, we briefly introduce a technique that was previously developed to generate a fulfilling interleaving for a value schedule. This technique forms the basis for the implementation of our proposed collaboration scheme.

2.2 Value-Schedule-Based Testing

A value schedule of a concurrent program consists of a sequence of pairs of conflicting concurrent accesses to a shared variable, and each pair of conflicting concurrent accesses is annotated with the partial ordering between two elements [3]. For example, two MHP read/write accesses to shared variables or two MHP entries to the same monitor will form a pair in a value schedule. Then, the partial ordering of a pair of MHP read/write accesses determines whether the read access reads in the value assigned by the write, and that of a pair of MHP monitor entries determines the order of monitor entry. Thus, a distinct value schedule of a program corresponds to a distinct partial ordering of all accesses to shared variables in a concurrent program. Thread interleavings that fulfill the same partial ordering of a program will be considered to be equivalent because they will always bring the system into the same state.

The goal behind value-schedule-based testing is to execute exactly one fulfilling interleaving for each distinct value schedule in the concurrent program. Then, the fulfilling interleaving is executed in a controlled manner using a dynamic analysis tool, such as an explicit state model checker, to determine whether the program works properly under such a value schedule. The main advantage of explicitly con-

```

1  public class Sample {
2      public static void main(String[] args){
3          TypeA ta = new TypeA();
4          TypeB tb = new TypeB();
5
6          T1 t1 = new T1(ta, tb);
7          T2 t2 = new T2(ta, tb);
8
9          try {
10             t1.start(); t2.start();
11             t1.join(); t2.join();
12             System.out.println(ta.val+"_"+tb.val);
13         } catch (InterruptedException e) {
14             e.printStackTrace();
15         }
16     }
17 }
18 class T1 extends Thread{
19     public T1(TypeA o, TypeA o2){
20         this.ta1 = o;
21         this.ta2 = o2;
22     }
23     public void run(){
24         if (this.ta1.val < 3){
25             int l = this.ta1.val + 1;
26             this.ta1 = new TypeA();//Alias change
27             l += this.ta1.val;
28         }else{
29             this.ta1 = tb1;//Type change
30             this.ta1.op1();
31         }
32     }
33     TypeA ta1;
34     TypeA tb1;
35 }
36 class T2 extends Thread{
37     public T2(TypeA o, TypeA o2){
38         this.ta1 = o;
39         this.ta2 = o2;
40     }
41     public void run(){
42         this.ta1.val = 1;
43         this.ta1.val = 3;
44         this.ta1.val = 4;
45     }
46 }
47     TypeA ta1;
48     TypeA tb1;
49 }
50 }
51 class TypeA{
52     public void op1(){
53         int local = val + 1;
54     }
55     public int val = 10;
56 }
57 class TypeB extends TypeA{
58     public void op1(){
59         int local = 3;
60     }
61 }

```

Figure 1: A Sample Program

Value Schedule	Interleaving	Feasible
$[r(ta1.val, t1.24), w(ta1.val, t2.43)]$ $[r(ta1.val, t1.25), w(ta1.val, t2.43)]$	$w(ta1.val, t2.43)$ $r(ta1.val, t1.24)$ $r(ta1.val, t1.25)$	TRUE
$[r(ta1.val, t1.24), w(ta1.val, t2.44)]$ $[r(ta1.val, t1.25), w(ta1.val, t2.45)]$	$w(ta1.val, t2.43)$ $w(ta1.val, t2.44)$ $r(ta1.val, t1.24)$ $w(ta1.val, t1.45)$ $r(ta1.val, t1.25)$	FALSE

Table 1: Some Sample Value Schedules

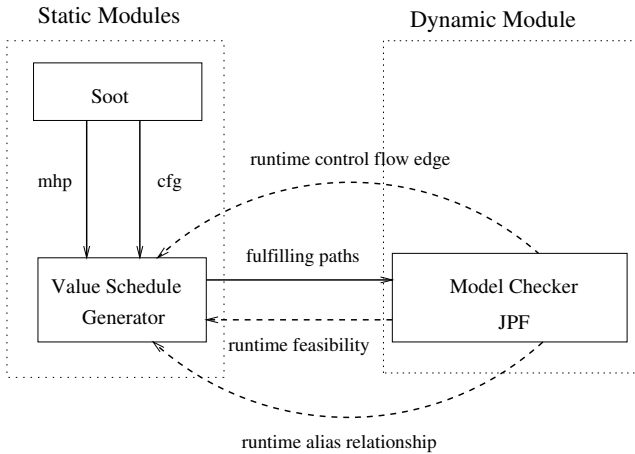


Figure 2: An Overview of Collaboration between Static and Dynamic Analysis

trolling the dynamic analysis tool to fulfill a distinct concurrent access pair is to avoid exploring more than one thread interleaving for each partial ordering. As a result, the total computational resources required for testing could be greatly reduced.

Some sample value schedules and their fulfilling interleavings for the program in Figure 1 are given in Table 1. A concurrent access pair is specified in the form $[r(v, t.linenum), w(v, t.linenum)]$, where r and w stand for read and write access respectively, v stands for the shared variable accessed, t stands for the thread this access belongs to, and $linenum$ stands for the line in the source code where this access is specified. The value schedule in the first row specifies that $r(ta1.val, t1.24)$ reads in value written by $w(ta1.val, t2.43)$, then the next read of $ta1.val$ in thread $t1$ at line 25 reads the same value. Its fulfilling interleaving is given in the second column of the first row. A controlled execution of the program using that interleaving will show this interleaving is feasible at runtime. Thus, the value schedule represented by this interleaving is feasible as well. However, the value schedule in the second row is infeasible since its fulfilling interleaving is infeasible because the branch statement at line 24 will skip the read access at line 25. The value schedules are derived and their fulfilling interleavings computed one concurrent access pair at a time [7]. So, once the partial value schedule in row 1 of Table 1 is successfully tested, static analysis resumes to discover the next concurrent access pair that could be reached from the current program state and generate its fulfilling interleavings accordingly.

2.3 Overview of Collaborations

Currently, our new extended value-schedule-based testing technique is implemented as shown in Figure 2. It consists of two main modules: the static analysis module and the dynamic analysis module. The static analysis module consists of two sub-modules: *Soot* and *Value Schedule Generator*. The *Soot* module is based on the Soot program analysis package [20]. It can produce useful information about a program, such as MHP relationships among instructions, interprocedural call graphs, and intraprocedural CFGs. Although the information is not precise, it is used by the *Value*

Schedule Generator module to statically uncover potential value schedules and compute their fulfilling interleavings. The computed value schedules are checked for feasibility and correctness by executing their fulfilling interleavings in the dynamic analysis module. The dynamic module is a customized version of an explicit state model checker called JPF. The dynamic information obtained from the controlled execution will help the *Value Schedule Generator* with further value schedule discovery.

The arrow-headed lines between modules indicate the exchange of information proposed by our new collaboration scheme in the context of the value-schedule-based testing technique. The solid lines indicates information flow from the static analysis module to the dynamic analysis module. The dashed lines indicate the information flow in the other direction. In the following section, we discuss each information flow in detail.

3. COLLABORATIONS BETWEEN STATIC AND DYNAMIC ANALYSIS

In this section, we present the different collaborations between static analysis and dynamic analysis shown in Figure 2 in the context of value-schedule-based testing. We will use the sample program in Figure 1 to illustrate these exchanges of information. Although the implementation of some of the collaborations discussed in this section have been presented in [6] [7], in this paper we concentrate on discussing the rationale behind the information exchange. For example, we want to discuss why the introduction of one type of information makes the presence of other types of information necessary, and how each type of information improves the testing process.

3.1 Static Analysis: MHP and CFG

The testing process starts with a coarse-grained static analysis. The output of the static analysis is a set of fulfilling interleavings of value schedules that will serve as an initial set of guides to the dynamic analysis.

The analyses provided by *Soot* gives two main sets of information. The first set of information includes some general program facts such as alias information (via data flow analysis), and intraprocedural and interprocedural control flow graphs (via control flow analysis). The second set of information concerns concurrent aspects of the program semantics, such as MHP instructions. The MHP instructions are computed by combining CFG information and alias information contained in the first set of analysis output. For example, two accesses to an aliased object from the *run()* methods of two threads could be considered to form a MHP pair by the *Soot* analysis.

The CFG and MHP information collected from the *Soot* package are used to compute value schedules and fulfilling interleavings. This process starts by randomly selecting a thread and traversing its CFG. Every CFG node encountered during the traversal will be added to a temporary path. When a node participating in a pre-determined MHP pair is encountered, the exploration is diverted to explore the thread that contains the partner CFG node that might form a MHP relationship with the current node. After the target node is found, the exploration path leading to the target partner node will be added to the path already computed on the thread that triggered the divergence. Then,

the exploration is resumed from the triggering node. In this way, we could compute an execution path that leads the program to test a partial order in which the partner node happens first. An example of value schedule generation and fulfillment is shown in Table 1. The details of value schedule generation are given in [6] and [7].

Compared to many existing approaches [16] [2] [11] which only use MHP instructions as markers for possible permutations in the dynamic analysis, we used static analysis to generate much more detailed information. The output from our initial static analysis not only indicates where the permutations might need to occur but also the possible interleavings that the dynamic analysis should follow to fulfill the partial ordering promised by the permutation. With the close guidance of static analysis, we try to guide dynamic analysis to systematically discover and test all partial orderings in the program, and test every distinct partial ordering only once. However, the introduction of static analysis brings along both its advantages and its disadvantages. First of all, due to the imprecise CFG, fulfilling interleavings may be generated to fulfill a partial ordering that is not feasible. Secondly, static analysis always generates correct but conservative facts about the program, so the guidance generated from those imprecise facts is not precise either. The solution to those issues lies in how to use dynamic runtime information to correct the imprecision of the initial static analysis. We illustrate our solution in the following sections.

3.2 Dynamic Analysis: Feasibility

For each value schedule derived by the static analysis, dynamic analysis will have to check its feasibility by running a fulfilling interleaving using a dynamic analysis tool, JPF. As the fulfilling interleavings are executed by our modified version of JPF, JPF will report various types of dynamic runtime information back to the static analysis. In this section, we introduce the first type of information returned by JPF: the feasibility information of each value schedule. We first show how this information is collected. Then, we show how the feasibility information helps to remedy the imprecision of coarse-grained static analysis.

JPF executes the interleaving by exploring the program state space according to the instruction sequence specified by the interleaving produced by static analysis. Every instruction will be considered as a transition. After an instruction is executed in JPF, the instructions referenced by the program counters of active threads will form the *next-to-run-set* for JPF. If the next instruction statically specified by the interleaving computed by static analysis is not in the *next-to-run-set*, then the interleaving is determined to be infeasible. This can happen because the control flow of the program does not take the branch expected by static analysis. On the other hand, an interleaving is feasible if its every instruction is executed in order by JPF. It is important to note that a partial ordering becomes infeasible if all interleavings generated to fulfill it fail to execute in JPF. The details of computing feasibility of a value schedule are given in [6] and [7].

The feasibility status collected by JPF on value schedules will inform JPF whether the interleavings triggered by a permutation will actually lead to a new partial ordering. Once an interleaving is infeasible, any further exploration on that interleaving can be abandoned. On the other hand, approaches that only use MHP instructions as markers for

permutation cannot know this because they have no mechanism to evaluate the success of a permutation. In that case once a permutation is carried out, both interleavings have to be explored, even though one of them might not trigger any new partial ordering.

As for our example in Figure 1, we use the value schedule in the second entry in Table 1 to illustrate the usefulness of feasibility information. Both our approach and the marker-based approach will permute the write to *ta1.val* at line 45 with the read of *ta1.val* at line 25. From the feasibility information, we know such a permutation will not lead to a new partial ordering if the read at line 24 has read in the value given by the write at line 44 because that assignment forces the *true* branch to be skipped and renders the interleaving infeasible. Thus, we could instruct JPF not to execute any additional transitions once the path to *r(ta1.val, t1.25)* is shown to be infeasible. On the other hand, the marker-based approach will try to trigger and run both interleavings to completion whenever the write at line 45 is encountered regardless of the context. Therefore, the runtime feasibility information helps the dynamic analysis to prune out irrelevant explorations. The feasibility information is shown by the *feasibility* arrow in Figure 2. In Section 3.4, we will discuss how we could help JPF detect the infeasibility of a value schedule or partial ordering as early as possible.

3.3 Dynamic Analysis: Alias Information

Once a feasible fulfilling interleaving for a permutation is computed and tested, we must determine if the partial ordering we just fulfilled really happened on a pair of concurrent accesses to the same object. As discussed in Section 3.1, the MHP instructions that are marked for permutations are computed based on *may-alias* or *must-alias* relationships. Therefore, it is possible that a partial ordering fulfilled at runtime actually happened for accesses to two non-aliased objects. For example, static analysis will conclude that the write of *ta1.val* at line 45 and read of *t1.val* at line 27 is a concurrent access pair. Therefore, the access at line 45 will be marked to permute with the access at line 27 when encountered during the CFG traversal. However, these two accesses actually operate on different objects at runtime due to the redefinition of *ta1* at line 26. Nevertheless the *Value Schedule Generator* will compute feasible interleavings for permutations between those two accesses. However, these permutations will not lead to any distinct value schedules. To cope with this kind of imprecision in the static analysis, we now introduce the second type of dynamic information returned by JPF: the dynamic alias relationship on the statically determined MHP members. This information is represented by the *runtime alias information* edge in Figure 2.

Although the precise alias relationship between two instructions is prohibitively expensive to compute statically, this same relationship could be easily obtained by querying the dynamic analysis tool, such as JPF, to obtain the memory address referenced by the two instructions at runtime. If the memory addresses referenced by two instructions are different, then they are non-aliased accesses and irrelevant to the partial ordering. To retrieve this information, we have to access the instructions in question from the JVM provided by JPF just before they are run. Because JPF is an explicit state model checker, it creates a snapshot of the JVM state after the execution of an instruction. Such a snapshot is called a *kernelstate* by JPF. A *kernelstate* stores

relevant information such as the heap structure of the program and stack information of each thread. The thread stack is where we can retrieve the program counters, which can be dereferenced to obtain the next instruction to execute. The last issue how to produce an interleaving when the two instructions in question are the next to run in their respective thread. Recall from Section 3.1 that a fulfilling interleaving for a permutation is comprised of the instructions leading to, but excluding, the permutation triggering node in one thread and the instructions leading to the permutation partner node in another thread. It is important that the permutation partner node is always the last instruction in a fulfilling interleaving. So, once the fulfilling interleaving is successfully executed up to the last instruction, the instruction that triggered the permutation and its permutation partner are both the next-to-run instructions of their respective threads. Using the program API provided by the JPF, we can compare the memory references of those two instructions.

In the case of our sample program in Figure 1, the spurious concurrent access pair $[r(ta1.val, t1.27), w(ta1.val, t2.45)]$ could be pruned out by executing all but the last instruction in the fulfilling interleaving: $w(ta1.val, t2.43) \rightarrow r(ta1.val, t1.24) \rightarrow w(ta1.val, t2.44) \rightarrow r(ta1.val, t1.25) \rightarrow r(ta1.val, t1.26)$. After the memory references of $r(ta1.val, t1.27)$ and $w(ta1.val, t2.45)$ are checked, we know they are not aliased at runtime. Then, this interleaving will not be further extended by the *Value Schedule Generator*.

A closer examination of the alias checking technique will certainly raise the question why we check the alias relationships for $ta1.val$ instead of $ta1$ directly. In that case, we should make the $w(ta1, t1.26)$ and $r(ta1, t2.45)$ into next-to-run instructions in their respective threads and compare them. Ideally, if they do not match, all subsequent operations on fields of those two objects will not form any concurrent access pair. However, this requires the static analysis to be precise. For example, we have to know which accesses to $ta1.val$ are bound to the object allocated at line 26. Unfortunately, precise static analysis for concurrent programs is an undecidable question [17]. Therefore, the alias checking on accesses to fields of shared variables is a practical and safe solution to remedy the imprecision of the static analysis.

3.4 Dynamic Analysis: Runtime CFG

As discussed in Section 3.1, the fulfilling interleaving of a partial ordering is computed by traversing the CFG of the program following both intraprocedural and interprocedural edges. However, the CFG generated by the coarse-grained static analysis is not precise with respect to a particular runtime interleaving. For example, the branch node at line 24 of our sample program has two successors. Thus, for a CFG traversal to gather interleavings, we might need to traverse both branches. However, because every feasible runtime interleaving leading to this branch node will take only one of the two branches, the work done by traversing one of two branches following this branch node is always superfluous.

The imprecision in the CFG caused by dynamic method calls on polymorphic objects also affects the efficiency of interleaving generation. A coarse-grained static analysis will assign more than one possible type to an object reference in the code. Consequently, a method call to an overridden method on that object will produce multiple edges in the interprocedural call graph, each bound to a possible ob-

Value Schedule	Interleaving
$[r(ta1.val, t1.24), w(ta1.val, t2.44)]$ $[r(ta1.val, t1.53), w(ta1.val, t2.45)]$	$w(ta1.val, t2.43)$ $w(ta1.val, t2.44)$ $r(ta1.val, t1.24)$ $w(ta1, t1.29)$ $r(ta1.val, t1.53)$ $w(ta1.val, t1.45)$

Table 2: Value Schedule using Runtime CFG

ject type. Therefore, during interleaving generation, the traversal will follow every possible out-edge from a dynamic method call site. Because only one edge is actually taken for any feasible interleaving at runtime, all but one traversal done in the interleaving generation stage are superfluous.

Although fulfilling interleavings generated from those superfluous traversals could be filtered out as infeasible interleavings as described in Section 3.2, the computational resources spent on superfluous traversals could be saved if the runtime decision on the control flow branch could be provided to the *Value Schedule Generator* before the traversal continues from a control flow branch point. As discussed in Section 3.3, the next-to-run instructions from each thread maintained by JPF can again be used to extract useful runtime information. In the case of a control flow branch caused by a conditional statement such as an *if* statement, the interleaving leading to that branch, including the conditional statement, will be executed. After the conditional is executed, the next-to-execute instruction referenced by the program counter of that thread will indicate which branch has been taken. Then, the *Value Schedule Generator* will follow that successor in further fulfilling interleaving computations. In the case that a branch is caused by a dynamic method call, then only the interleaving leading to the method call is executed by JPF. At this point, the object referenced and its runtime type are known and can be extracted from the program state saved by JPF. When the runtime type of the callee is determined, the concrete method can also be determined. With this updated information, the *Value Schedule Generator* will only traverse the out-edge corresponding to the runtime method. The runtime CFG information is shown as the *runtime control flow* edge in Figure 2.

With the concrete runtime information, when the fulfilling interleaving is computed for the value schedule in the second entry of Table 1, the branch statement at line 24 will trigger a branch test by executing the interleaving leading to it. The runtime results collected will instruct the *Value Schedule Generator* to take the *else* branch of the conditional statement when looking for the permutation partner $r(t1.a, t1.25)$, and skip the *true* branch entirely. A very coarse-grained reachability analysis will indicate that the desired permutation partner cannot be found in the *else* branch, and declares the value schedule infeasible.

Suppose we modify the value schedule listed in the first entry of Table 1 to that shown in Table 2 to include the partial ordering between the read of val that happens at the call to $ta1.op1()$ at line 30 and the write of $ta1.val$ that happens at line 45. The execution of the interleaving, $w(ta1.val, t2.43) \rightarrow w(ta1.val, t2.44) \rightarrow r(ta1.val, t1.24) \rightarrow w(ta1, t1.29)$ shows the runtime type of $ta1$ at line 30 is actually *TypeB*, while the runtime type of the write at line 45 is *TypeA*. Therefore, further exploration will not reach the permutation partner, the read of val in *TypeA*'s $op1()$ method, and the exploration can be halted. It is important to note that

the alias relationship between $w(ta1, t1.30)$ and $r(ta1, t2.45)$ could also prune this interleavings. However, due to the imprecision of the static analysis, the alias relationship cannot be properly resolved until an interleaving that leads to the execution of both instructions is constructed and run. With concrete runtime CFG information, we can detect infeasibility more quickly and avoid exploring interleavings that cannot produce the desired concurrent access pair.

3.5 Discussion

To summarize, static analysis performs the first cut in the overall search space for a program, then the dynamic analysis refines the remaining search space with additional help from static analysis. The initial program facts computed by static analysis provide the first set of guidance to the dynamic analysis tool. However, the guidance also introduces the imprecision of static analysis into the dynamic analysis tool. To remove or reduce the imprecision, the runtime information specific to a particular interleaving will have to be provided to the static analysis to refine its initial computations. The feasibility information tackles the imprecision in the control flow analysis, while runtime CFG edges further improve the efficiency of feasibility analysis. The runtime alias checking reduces the imprecision in the data flow analysis. In the end, all information exchanges and refinements strive for the same goal: discovering and testing a partial order as fast as possible.

A fair question to ask is whether there are other ways in which static analyses could combine with dynamic analyses. For example, we would like to know what other types of information should be exchanged and in what manner. More importantly, once the information is collected, how can each type of analysis make use of it? For instance, at a permutation point in interleaving generation, is it useful to invoke static analysis to compute a refined CFG for both interleavings based on the dynamic program facts collected so far? Moreover, the concrete information collected by dynamic analysis is only conservatively valid for the thread interleaving under testing. It would be interesting to know whether some of this information could be applied to other interleavings using static analysis. In the future, we would like to explore these questions in detail.

4. A CASE STUDY

In this section, the effectiveness of our collaboration scheme is demonstrated by running the sample program in Figure 1 using out-of-the-box JPF (JPF version 3.1.2) and our implementation of our new collaboration scheme. We would have liked to compare the performance of our technique against other testing techniques such as [2] [16] which also combined static and dynamic analysis for testing. However, implementations of those techniques are not available. The number of interleavings, program states and transitions taken by the two techniques to cover all distinct partial orderings will be compared.

The sample program has two threads. Each thread has two shared variables $ta1$ and $tb1$. $ta1$ and $tb1$ are instances of type $TypeA$ and $TypeB$ respectively. Moreover, $TypeB$ extends $TypeA$. Thread $T1$ performs different operations based on the value of $ta1$, while thread $T2$ modifies the value of $ta1$ three times. The operations in thread $T1$ are divided into two branches by an *if* statement. In the *true* branch, the $ta1$ of thread $T1$ is redefined to a new object of

	Our Approach	Out-of-the-box-JPF
Interleavings	6	10
Program States	49	210
Transitions	7839	12977

Table 3: Experimental Results

type $TypeA$, while in the *false* branch, the $ta1$ of thread $T1$ is set to the value of $tb1$ in $T1$.

This program will produce 6 distinct partial orderings or value schedules. In the case where $r(ta1.val, t1.24)$ reads in the value of $ta1.val$ written by $w(ta1.val, t2.43)$, the *true* branch will be taken. The $r(ta1.val, t1.25)$ will form three distinct partial orderings with the three writes in thread $T2$. The $r(ta1.val, t1.27)$ operation will not spawn any extra partial orderings because it operates on a new object, as we discussed in Section 3.3. In the case that $r(ta1.val, t1.24)$ reads in the default value of $ta1.val$, the value given by $w(ta1.val, t2.44)$ or the value given by $w(ta1.val, t2.45)$, then the *false* branch will be taken. Because $ta1$ in the *false* branch is redefined to another type, only one partial ordering is generated for each of those three cases. In total, there are 6 distinct partial orderings or value schedules that could be generated.

As shown in Table 3, our technique covers all partial ordering or value schedules with much fewer interleavings, program states and transitions than out-of-the-box JPF. The out-of-the-box JPF permutes on every access to an instance variable that is reachable by more than one thread at runtime. Thus, JPF will permute on an access whose concurrent access partners are no longer reachable in the successive exploration. For example, for the interleaving listed in Table 2, JPF will permute before executing $w(ta1.val, t2.45)$ and execute both interleavings to completion even though they do not produce different value schedules.

5. RELATED WORK

There are two main approaches for uncovering concurrency bugs: static analysis, and dynamic analysis. Static analysis performs different analysis methods iteratively to uncover potential bugs [15] [14] [1] [10]. Static analysis is insensitive to different program inputs. However, because many program analysis problems for concurrent semantics are either NP-Complete [19] or undecidable [17], static analysis must make numerous assumptions during program analysis. Depending on these assumptions, static analysis methods could produce both false positives and false negatives in the bug report.

Dynamic analysis uncovers concurrency bugs by running the program using different thread interleavings. In [18] [22], programs have been devised to run many times to uncover deadlocks. In [8] [3] [4], pre-installed sleep statements are invoked randomly at runtime to trigger different thread interleavings. This approach may trigger and test a partial ordering more than once. Moreover, there is no guarantee that a feasible partial ordering will be tested before the computational resources run out. An approach proposed by [3] [4] is to change the values returned by a concurrent read dynamically as the program executes. It is an attempt to test different partial orderings on-the-fly, but it can only account for events that have already occurred and not those that will happen later. However, this approach may run into circular dependency issues. Dynamic analysis could also be

implemented using an explicit state model checker such as JPF [21] and Verisoft [12] to systematically explore all interleavings of a concurrent program. However, explicit state model checkers can suffer from the state-explosion problem without aggressive partial order reduction. Dynamic analysis may contain false negatives because computational resources could be used up before all relevant interleavings are triggered.

Attempts have been made to combine static and dynamic analysis for testing concurrent programs. Bruno et al. [9] use call graphs collected from execution runs to refine the statically computed escape properties of variables. Choi et al. [16] uses MHP and alias analysis to identify a set of instructions relevant for partial orderings, and instrument them to capture data races at runtime. R. Agarwal et al. [2] uses static type checking to determine unsafe code in programs, then focus on-the-fly checking on the unsafe code for possible race conditions and atomicity violations. MacDonald et al. [13] try to combine static analysis and aspects to control the execution of program according to potential value schedules. However, this approach does not handle control flow well, and can produce an infeasible execution of a concurrent program. Godefroid [12] uses alias analysis to identify the dependency between enabled transitions to perform persistent-set-based partial order reduction. Brat et al. [5] use alias analysis to identify non-aliased accesses to avoid permutations at runtime. This non-aliased set is refined with the alias information collected from every exploration of an interleaving in the model checker. To the best of our knowledge, this work is the only one that supports a bi-directional exchange of information. However, it only exchanges one type of information, alias information, while our approach exchanges a wider range of information. In particular, without dynamic feasibility information, Brat's approach may test a partial ordering more than once.

6. CONCLUSION

In this paper, we presented a new collaboration scheme between static and dynamic analysis for testing concurrent programs. The static analysis gives guidance to the dynamic analysis to avoid testing a distinct partial ordering more than once. This guidance is corrected during testing using runtime information collected by the dynamic analysis tool. Collected runtime information includes alias relationships and CFG information. In turn, the refined guidance provides better further guidance to the dynamic analysis. The collaboration is bi-directional and repeated multiple times during the testing process. We have integrated our new collaboration scheme into our value-schedule-based testing technique. Moreover, some preliminary experiments show it delivers an improvement over out-of-the-box JPF.

7. REFERENCES

- [1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.
- [2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005.
- [3] M. Biberstein, E. Farchi, and S. Ur. Choosing among alternative pasts. In *Proc. 17th International Symposium on Parallel and Distributed Processing*, page 289.1, 2003.
- [4] M. Biberstein, E. Farchi, and S. Ur. Fidgeting to the point of no return. In *Proc. 18th International Parallel and Distributed Processing Symposium*, page 266b, 2004.
- [5] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, pages 262–269, 2001.
- [6] J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 313–322, 2007.
- [7] J. Chen and S. MacDonald. Incremental testing of concurrent programs using value schedules. Submitted to *23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.
- [8] S. Copty and S. Ur. Multi-threaded testing with AOP is easy, and it finds bugs!. In *Proc. 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 740–749. Springer-Verlag, 2005.
- [9] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proc. 2007 International Symposium on Software Testing and Analysis*, pages 118–128, 2007.
- [10] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.
- [11] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.
- [12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.
- [13] S. MacDonald, J. Chen, and D. Novillo. Choosing among alternative futures. In *Haiifa Verification Conference*, volume 3875 of *Lecture Notes in Computer Science*, pages 247–264. Springer-Verlag, 2005.
- [14] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2007.
- [15] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 308–319, 2006.
- [16] R. O’Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.
- [17] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable.

ACM Transactions on Programming Languages and Systems, 22(2):416–430, 2000.

- [18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.
- [19] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, 1983.
- [20] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java bytecode optimization framework. In *Proc. 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, 1999.
- [21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
- [22] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.