Deterministically Executing Concurrent Programs for Testing and Debugging

Steve MacDonald and Jun Chen
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{stevem, j2chen}@uwaterloo.ca

Diego Novillo
Red Hat Inc.
dnovillo@acm.org

Abstract

Non-determinism is a serious problem in the testing and debugging of concurrent programs. Thread interleavings may be different on each run, changing the order of events. Testing a concurrent program requires many of these orders to be run, either to determine that all orders produce correct results or to identify the presence of timing-dependent errors called race conditions. This paper presents preliminary work in deterministically executing a concurrent program using a combination of an intermediate compiler form and aspect-oriented programming. Specifically, we execute the code generated for the intermediate compiler form using aspects to control the value returned for the read of any shared variable. This allows us to deterministically execute specific test cases. This work is preliminary, with many outstanding issues, but we feel the technique shows promise.

Keywords: Concurrent programs, testing, race conditions, aspect-oriented programs, CSSAME compiler form.

1 Introduction

Testing any software is difficult because of the large number of paths through the code. Testing concurrent software is more difficult because of non-determinism. Testing must account for differences in program execution that result from thread interleavings that change each time the program is run. These interleavings can alter the order of events in a program, specifically the order of reads and writes to shared variables, which can lead to *race conditions*. For our purposes, a race condition occurs when a set of operations must execute in a specific order for correctness but the programmer has failed to supply synchronization or mutual exclusion to guarantee that order. This is a more general definition that considers timing-dependent errors and not just missing mutual exclusion.

This paper presents some preliminary work on controlling the execution of a concurrent program using two technologies. The first is the CSSAME form, an intermediate compiler form that analyzes explicitly-parallel shared-memory programs [14, 15, 16]. This compiler form captures definitions and uses of shared variables, noting which writes are visible at a given variable use. The second is aspect-oriented programming [11], specifically AspectJ [10]. We use aspects to control the value of shared variable reads to execute specific test cases. This paper assumes familiarity with aspect-oriented programming.

Using these technologies, we have created a technique for controlling the execution of a concurrent program with three desirable characteristics. First, we can deterministically test a program for all values of a shared variable at a given read, simulating all thread interleavings. Other techniques are dependent on the timing characteristics of the program execution, limiting the values they can consider at a given read. We do not rely on application timing but rather use aspects to control the execution. This allows us to test any possible interleaving and provide better coverage. Second, we do not require an existing execution trace of the program. We are deterministically executing the program, not replaying it. Third, this method works even if shared variable accesses are not protected by locks.

This deterministic execution has several potential uses. First, we can use this technique to construct specific test cases. We can not only produce a specific execution to locate the problem, but run it again to verify it has been corrected. Given the large number of interleavings, it may be best to use this to produce unit tests or an initial sanity test, to verify the basic operation of the system. It could also be used to enumerate over all interleavings for a specific shared variable. Second, this technique could be used to improve the operation of *noisemakers* [7, 8, 18], which run a program many times using heuristics to generate different interleavings. However, these heuristics may not be able to generate all interleavings. Our aspect-oriented approach could increase the test coverage of these tools with fewer runs of the program. Third, this approach could be used for incremental debugging. Users could construct schedules with small differences and compare results. Differences between schedules that produce cor-

rect and incorrect results may help identify the error [5].

2 Related Work

One technique for detecting race conditions is *noise-making*, running a program repeatedly but conditionally executing sleeps or yields to influence the interleavings produced by the scheduler [8, 18]. A recent effort used aspects to insert noise, and employed other heuristics like lowering the arguments to sleep calls [7]. This noise is necessary because many thread schedulers are almost deterministic, generating many executions that are behaviourally equivalent and thus redundant for testing purposes [1]. However, noisemakers depend on proper seeding of delays (how often to delay and for how long) and do not have a fine degree of control over program execution, making it difficult to control testing.

An improvement to noisemaking is *value substitution* [1]. In addition to inserting noise, value substitution tracks shared variable writes. On a read, value substitution randomly chooses a value from an already-executed write that could be visible, simulating different thread schedules. A visibility graph is maintained to ensure a consistent execution. However, the visibility graph is prohibitively large. Further, this technique can only substitute values from writes that have already executed, so the interleavings that can be tested depends on the execution timing of the program. *Fidgeting* was introduced to solve this problem [2]. The choice of value for substitution is delayed as long as possible to increase the set of possible substitutions, but there is still no guarantee of test coverage.

Another common debugging technique is *program replay* provided by tools like DejaVu [4] and others. These tools capture the execution of a program so it can be re-executed. If an erroneous execution is captured, the error can be repeated until the problem is found. However, capturing an error condition can be difficult if it appears infrequently. Further, the code that captures the execution may perturb the program enough to remove the error. Once the problem has been corrected, the execution state likely cannot be used to verify the correction. The complete testing process will have to be repeated.

A variation of replay called *alternative replay* uses the visibility graph from value substitution to produce new thread interleavings [2]. This is similar in style to *reachability testing* [9]. Both replay a program up to a given point in its execution, but rather than continuing with the existing trace, they select a new event to execute next. From that point, the program executes normally. Reachability testing tracks the set of executed schedules to enumerate over all possible interleavings. Both systems rely on deterministic replay (for the events at the start of the program). In contrast, the properties of the CSSAME form permits us to achieve the same effects without imposing this total order, by having a history of each write to a variable. The necessary properties are discussed in Section 4. Further, we can control the execution of the complete program or any portion, not just from the beginning of the execution.

A race condition is sometimes defined as concurrent accesses to shared variables without proper synchronization, a less general definition than the one used in this paper. There are tools that try to verify this property at runtime, like Eraser [17]. Eraser verifies not just that locks are held, but also tries to associate variables with locks to ensure that the correct lock is held. The main drawback is that it can only verify a specific execution of the program.

Another approach to locating errors in concurrent code is model checking, implemented by tools like Java PathFinder (JPF) [3] and many others. Most model checkers do not execute the application, but rather create an internal representation and analyze the representation to verify properties of the original program. JPF is a new Java virtual machine that detects race conditions using the Eraser algorithm [17]. It also implements thread scheduling, which is used to enumerate over different interleavings. This testing uses backtracking to save re-executing a program for each test. Our approach cannot use backtracking, but is considerably simpler as it does not require a new JVM.

3 The CSSAME Compiler Form

CSSAME (Concurrent Static Single Assignment with Mutual Exclusion, pronounced *sesame*) is an intermediate compiler form for analyzing explicitly-parallel shared-memory programs [14, 15, 16]. It extends the Concurrent Static Single Assignment form [12], taking the synchronization structure of the program into account in its analysis. Both are concurrent extensions of the Static Single Assignment (SSA) form for single-threaded programs.

A program in SSA form has two key properties. First, each variable is assigned only once. A single variable in a program is replaced with multiple subscripted versions, one for each assignment. Second, there can only be one *reaching definition* for a given use of a variable. A reaching definition for a variable use is a write to that variable that may be the value that is read (*i.e.*, there is an execution path with no intervening writes between the reaching definition and the use). However, control flow statements can yield multiple reaching definitions for a use. To enforce the second property, SSA inserts *merge operators* into the code, which appear as ϕ functions. The arguments to the ϕ function are the set of reaching definitions for the variable use. The return value of the ϕ function is one of its arguments, determined by the path through the code. An example of the SSA form is shown in Figure 1.

CSSAME extends SSA to include *concurrent reaching definitions* from other threads. A concurrent reaching

```
a = 0                          a1 = 0
if (condition)                 if (condition)
    a = 1                          a2 = 1
print(a)                       a3 = φ(a1, a2)
                               print(a3)
```

(a) Original source code. (b) SSA form.

Figure 1. Example: ϕ functions in SSA.

definition for a variable use is a write to that variable by another thread that may be the value that is read. Like SSA, a use of a variable can only be reached by exactly one definition, so concurrent definitions are merged using π functions inserted into the code. The arguments to a π function are the reaching definition from the local thread together with the set of reaching definitions from other threads. Reaching definitions from within a single thread are still merged with ϕ functions. CSSAME prunes the arguments to a π function based on the synchronization structure of the program, since this limits the set of definitions that could be read by another thread. This pruning is based on two observations about the behaviour of shared variable accesses from within a critical section [16]. First, a definition can only be read by another thread if it escapes the critical section. Second, if a definition and use occur in the same critical section, then concurrent reaching definitions from other threads that are protected by the same lock are not visible. This pruning process reduces the number of dependencies between threads and increases opportunities for optimization.

An example of the CSSAME form is given in Figure 2. The use of a_4 in T_3 can only have one concurrent reaching definition, so a π function is used to merge the writes from T_1 and T_2. Note that a_1 is not part of the merge; the definition does not escape the critical section in T_1 and the code in T_3 runs with the same lock. Thus, it is not possible for T_3 to read a_1. If the lock in either T_1 or T_3 is removed, then a_1 would be added as an argument to the π function.

For unsynchronized accesses to shared variables, the set of concurrent reaching definitions are dictated by the underlying memory model. Different memory models impact the placement and arguments of the π functions [14].

The CSSAME form is generated by the Odyssey compiler [14]. Odyssey supports a superset of C including explicitly-parallel programming constructs. Parallelism is created using cobegin/coend regions, used in this paper, or parallel loops. In the cobegin/coend region, thread bodies are specified using syntax similar to that in Figure 2(a). Each body runs independently. Execution resumes after the coend statement when all threads have completed. Odyssey supports several synchronization constructs. The best supported are locks, which Odyssey analyzes to remove unnecessary terms in π functions. Event variables and barriers are also available but are not analyzed.

Note that Odyssey only analyzes the code that it sees. If

```
a = 0                          a0 = 0
cobegin                        cobegin
    T1: Lock(L)                    T1: Lock(L)
        a = 1                          a1 = 1
        a = 2                          a2 = 2
        Unlock(L)                      Unlock(L)
    T2: Lock(L)                    T2: Lock(L)
        a = 3                          a3 = 3
        Unlock(L)                      Unlock(L)
    T3: Lock(L)                    T3: Lock(L)
        print(a)                       a4 = π(a0, a2, a3)
        Unlock(L)                      print(a4)
coend                                  Unlock(L)
                               coend
```

(a) Original Odyssey code. (b) CSSAME form.

Figure 2. Example: π functions in CSSAME.

concurrent reaching definitions are in code not processed by Odyssey (*i.e.*, library code), then the analysis will be incomplete. We assume no hidden definitions exist.

4 Using Aspects to run CSSAME Code

Deterministically executing a concurrent program requires us to control the order of events in the program. The events we need to control are those that cause interactions between threads, which are access to shared data and synchronization. We consider shared data accesses in this paper. Controlling these accesses requires the ability to order the reads and writes to the data. More specifically, if a shared variable is written by different threads, we need to control which of the written values is returned by a read operation. The most common approach to impose a total order on the events in a program by replaying an existing trace. However, this work is not based on replay.

Our approach is based on executing the CSSAME intermediate form, using aspects to establish an event ordering. However, we do not impose a total order on the events in the system. Instead, we exploit two key properties of the CSSAME form to create the appearance of deterministic execution: π functions that indicate concurrent reaching definitions and single assignment of variables. These properties provide information about thread interactions and allow us to capture the execution history without a total event order.

The π functions in CSSAME provide information about thread interactions by identifying the concurrent reaching definitions at a variable use. For a variable read, we know the writes from other threads that may be returned, presented as terms in the π function before the use. π functions with multiple terms highlight places in the code where the thread schedule may alter the outcome, as the value for a variable use depends on the order in which threads are run. As a result, returning different values for a π function simulates different schedules. One important note is that multiple terms are not necessarily an error. The

result may depend on the thread schedule, but if no concurrent reaching definition causes an error then there is no race condition.

Using the π functions, we can identify two potential sources of concurrency errors. First, a π function may include extra, incorrect terms. In some cases, these terms are false positives; if the compiler cannot recognize the synchronization structure, it assumes unsynchronized accesses and does not prune the terms. Otherwise, the use of the variable may read a value that the programmer does not want, but there is insufficient synchronization to ensure the value is not visible. Given Figure 2, the programmer may wish the program to print out the results from either T_1 or T_2. However, from Figure 2(b), we can see that the initial value of 0 may be read in T_3 if it executes before the other threads. Note that this error can still occur even if the thread bodies are mutually exclusive; barrier or event synchronization is needed to remove this error. This problem can often be identified by inspection. The presence of a_0 in the π function may be enough to spot the potential problem. The problem can also be noted by running the program several times while changing the value returned in the π function.

The second potential problem is that one of the specific values that is written results in an error. The programmer may have written the program so that its correctness does not depend on order, but specific values may cause errors. The programmer may want T_3 to print a value that is greater than 0, so the error is the initial value of a. This problem can be found by enumerating over the terms in the π function.

There are several difficulties in testing concurrent programs that arise here. First, the choice of value for a π function is made non-deterministically at runtime, so enumerating over all terms in a π function is difficult. This problem is exacerbated by the fact that thread scheduler implementations may be deterministic and may not produce all possible interleavings with equal probability. If the error condition occurs on an infrequently-used schedule, it can be difficult to produce the problem. Second, once the problem is uncovered, it may be necessary to repeat it to locate its source. Third, once an error has been detected, it can be difficult to be certain that changes to the program have corrected it without restarting the testing process from the beginning.

Our approach to addressing these difficulties is to deterministically execute a concurrent program by running its CSSAME equivalent, including π and ϕ functions. (For this paper, we will ignore ϕ functions and focus on concurrent reaching definitions.) However, we must still control the execution of the program by ordering the events in a program. We do this by controlling the return values for the π functions, selecting one of the terms before the program runs. For example, in Figure 2(b), the value for a_4 is the result from $\pi(a_0, a_2, a_3)$, which must be one of the three concurrent reaching definitions.

We control the return value for a π function with aspect-oriented programming. We create an aspect that intercepts calls to a π function and overrides the return value to be one of its arguments, chosen by the user. Different test cases can be constructed by a user or by automated testing software. The benefit to using aspects for this task is the same benefit as using aspects in general: separation of concerns. The CSSAME form can be generated once, independently of any test case. Each test case is a separate aspect. Without aspects, we would need to insert complex code into π function bodies that would need to be modified for each test.

However, this technique in itself introduces a race on the selected term in the π function. That is, before the aspect can return a given term, we must first be sure that the variable has been written. This problem is solved using another aspect that intercepts field writes, noting that only fields are shared between Java threads. Here, we exploit the single assignment property of CSSAME; a variable is represented by a set of subscripted versions of that variable, where a version is assigned once. This has two ramifications. First, we have a history of the variable, saved in different subscripted versions. Second, we can treat each version as a latch, knowing that once it is assigned it can be safely used without any further races. This latch is easily implemented by maintaining a boolean value indicating if the value for the variable has been set. The advice for a π function checks this boolean and blocks the calling thread until it is true. The advice for field writes sets the boolean and wakes any waiting threads. This solution allows any thread interleaving to be simulated, independently of the timing of events in the program execution.

It may seem that a simpler approach to this problem is to create aspects to intercept both reads and writes to shared fields and control the reads. However, analysis is still needed to find the set of concurrent reaching definitions for the read. We have chosen the CSSAME form as our analysis; other solutions are possible.

An example aspect, with helper code, is given in Figure 3. Figure 3(a) shows the aspect. The aspect defines four pointcuts: one that captures calls to the π function (line 5), and one that captures writes for variables a_0 (line 10), a_2 (line 12), and a_3 (line 14). The advice following the pointcuts provides code that is inserted into the CSSAME code. The around advice for the first pointcut, at line 21, replaces the body of the π function. This advice returns the value of a_2 for the π function in Figure 2(b). It does so by blocking the calling thread on the latch for a_2 until it opens. This test case selects the thread interleaving that has T_3 execute immediately after T_1; T_2 may have completed or may not have executed. By changing the latch in line 23, we can construct test cases that sim-

```
1  public aspect TestCase1 {
2    // The args() clause lets advice access
3    // argument values.
4    // Pointcut for the π function.
5    pointcut piFunc(int a, int b, int c) :
6    (call * int π(int,int,int) && args(a,b,c));
7
8    // Pointcuts for writes to a0, a2, a3.
9    // a1 is not visible in any π function.
10   pointcut set_a0(int n) :
11     set(protected int Example.a0) && args(n);
12   pointcut set_a2(int n) :
13     set(protected int Example.a2) && args(n);
14   pointcut set_a3(int n) :
15     set(protected int Example.a3) && args(n);
16
17   // Advice that wraps around execution of π
18   // function and replaces method body.
19   // Return the selected argument after it has
20   // been written.
21   int around(int a0, int a2, int a3) :
22       piFunc(int,int,int) && args(a0, a2, a3) {
23     ItemLatch choice = latch_a2;
24     synchronized(choice) {
25       while(!choice.ready) {
26         choice.wait();
27       }
28     }
29     return(choice.value);
30   }
31
32   // Trip the latch when the field is set.
33   after(int n) : set_a0(n) {
34     synchronized(latch_a0) {
35       latch_a0.ready = true;
36       latch_a0.value = n;
37       latch_a0.notifyAll();
38     }
39   }
40   after(int n) : set_a2(n) {
41     // Same as for a0
42   }
43   after(int n) : set_a3(n) {
44     // Same as for a0
45   }
46   ItemLatch latch_a0 = new ItemLatch();
47   ItemLatch latch_a2 = new ItemLatch();
48   ItemLatch latch_a3 = new ItemLatch();
49 }
```

(a) Aspect that selects a_2 for π function.

```
1  public class ItemLatch {
2    public ItemLatch() {
3      ready = false;
4    }
5    public boolean ready;
6    public int value;
7  }
```

(b) ItemLatch helper class.

Figure 3. Aspect and helper code for a test case in Figure 2(b).

ulate all of the potential thread interleavings. Note that this can be done with only three executions; we only need to determine which threads execute before T_3, which does not require us to enumerate over all possible orderings. For example, in this test case, it does not matter if T_2 executes before T_1 or after T_3; it is only important that T_3 runs immediately after T_1.

The advice for a_0, a_2, and a_3 (starting at line 33) runs after writes to the variables. It updates and opens the latch for the variable, waking any threads waiting for the value. The latch code is given in Figure 3(b).

This approach has several benefits. First, it is insensitive to the execution timing of the program. We do not establish a total order on the events in the program, but use the execution history provided by the multiple subscripted versions of shared variables. We can select a value for return in a π function regardless of when that value is written when the program executes. Second, we can construct any valid interleaving for a test case when desired. Specific scheduler implementations may not generate all possible schedules, and may generate others infrequently. Our selection of return values for π functions is not restricted by the scheduler. Third, we separate test cases from the CSSAME form using aspects. Otherwise, the generated CSSAME code would have to include all of the necessary synchronization code. This code would appear in the π function bodies and at every update of a shared variable, making for tangled code.

However, this approach has some problems. First, it is possible to select an invalid schedule. That is, it is possible to construct a test case specifying return values for π functions that cannot result from a valid thread schedule. This case sometimes appears as a program that does not terminate, with threads waiting for latches that will not open because the program did not write the desired variables. In other cases, the program terminates generating illegal output. We need a mechanism for preventing such test cases. Second, SSA (on which CSSAME is based) was created for analyzing procedural programs with scalars. It will need to be augmented to deal with arrays [6] and with the object-oriented constructs in Java. Third, loops pose problems for static single assignment forms. This problem may be addressed using techniques from dynamic single assignment [19]. These issues are discussed in [13].

5 Example

This section describes an example program based on an example found in [1]. The primary objective is to show how we can use our aspect-oriented approach to construct different program executions to expose errors, and to highlight some of the additional characteristics of this approach. Another example can be found in [13].

We do not yet have an implementation of the CSSAME

```
int n = 1
cobegin
    T1: Lock(L)
        n = 0
        Unlock(L)
    T2: sleep(10)
        Lock(L)
        n = 3
        Unlock(L)
        barrier(2)
    T3: barrier(2)
        Lock(L)
        print(21/n)
        Unlock(L)
coend
```

(a) Odyssey code.

```
int n0 = 1
cobegin
    T1: Lock(L)
        n1 = 0
        Unlock(L)
    T2: sleep(10)
        Lock(L)
        n2 = 3
        Unlock(L)
        barrier(2)
    T3: barrier(2)
        Lock(L)
        n3 = π(n1, n2)
        print(21/n3)
        Unlock(L)
coend
```

(b) CSSAME form.

Figure 4. Example based on Fig. 7 from [1].

form for Java code. For now, we are prototyping such a compiler by writing explicitly-parallel code for Odyssey, then manually translating the generated CSSAME code to its Java equivalent and adding the necessary aspects. This translation process is described in [13].

The example, in Figure 4, is based on Figure 7 from [1]. In the original, one thread sets an object representing a network connection to null. A second thread sleeps, executes a long-running method, then initializes the connection object. A third thread waits for the second to complete and then sends data along the connection, assuming a non-null value. We simulate this problem using integer arithmetic. The first thread sets a shared variable to 0. The second sleeps then sets the variable to 1. The third thread uses the variable as a divisor, assuming a non-zero value. To ensure the third thread runs after the second finishes, we use a barrier that waits for two threads to arrive.

For this example, we will assume that all of the concurrent reaching definitions are expected by the programmer, and the error is that some values produce incorrect results.

When a Java version of this program is run 1000 times, any non-zero sleep value in T_2 results in a final value of 7 because T_1 executes to completion first. In fact, the results are correct if the sleep in T_2 is removed as long as T_1 is started first. However, the program runs correctly because the scheduler hides the error. If the sleep in T_2 is removed and it is launched first, the program almost always throws an exception (only 5 of 1000 runs ran correctly).

The π function in T_3 captures the concurrent reaching definitions for n_3, which shows that n_1 from T_1 or n_2 from t_3 will be the divisor. Testing this code only requires us to consider these two possibilities. Although there are a large set of potential thread schedules, these schedules can only impact the results of this program by changing which concurrent reaching definition is used for the divisor. Or rather, all thread schedules result in one of the two possible outcomes, so we only need to test that these two outcomes are correct. We need only run the program with n_3 equal to 3 (from n_2) and n_3 equal to 0 (from n_3).

Using aspects around the π function, we can select either term as the return value to test the two outcomes. With our aspect-oriented approach, we can test this program by running it only twice. In contrast, noisemaking programs are sensitive to the timing execution of the program, making it difficult to properly test programs that mistakenly use sleeps or other delays to remove race conditions. In these programs, the inserted noise must outweigh that already in the program to force different interleavings, which may be difficult since noise is probabilistically executed with random sleep values. If the sleep in T_2 is sufficiently large, it may take many executions to force the incorrect interleaving. The aspect version of noisemaking uses advice to reduce the length of program delays by overriding and reducing the argument to sleep [7], but again this is done probabilistically. The timing relationships in the original source code influence the ability of these tools to properly test this code. Our aspect-oriented approach does not suffer from these timing problems. If we select n_1 as the result for the π function, we need only ensure that the write has completed before n_3 is assigned, which is accomplished using latches. Our approach is more closely related to value substitution [1, 2] except that we are again not influenced by execution timings in the original. Value substitution substitutes values from writes that have already completed in the program. Again, the sleep in this program makes it difficult for value substitution to detect and use the incorrect value from T_1. Our latches remove these timing relationships and allows us to substitute any legally visible value for n_3.

To further show how value substitution depends on the execution timing of the program, consider a version of Figure 4 where the sleep is moved from T_2 to the start of T_1. Now, the problematic write to n_1 is hidden because T_1 executes last. This change has no effect on our aspect-oriented solution because our mechanism involves selecting values for the π functions. Again, the latches for the variables remove execution timing problems. With value substitution, it is unlikely that the write of the zero value will be finished when T_3 accesses it, so the value is unlikely to be substituted. Fidgeting is also unlikely to help. Adding noise to value substitution may increase the chances of testing this scenario, but it again relies on good seeding. Alternative replay is probably the best solution.

An additional problem for noisemakers and value substitution is that the number of executions needed to thoroughly test the program increases with the number of concurrent reaching definitions. Take a π function with w terms (one local reaching definition, $w-1$ concurrent reaching definitions). Further, assume that all w terms are always assigned when the variable is used. Value substitution selects a term with probability $\frac{1}{w}$. This will likely require more than w executions to capture all possible in-

terleavings. Noisemakers, with less control over program execution, will likely require even more executions. Our approach takes exactly w executions.

Our approach also allows executions to be repeatedly executed during debugging. Once the division by zero is uncovered (by selecting n_1 as the return value for the π function), this interleaving can be repeated until the programmer uncovers the problem. In this example, the problem can be identified quickly. In more complex programs, it may take more effort to identify the source of the error.

Equally important is that our approach can help retest the program once a fix is applied to the code. In this example, the error is that n_1 is assigned zero, causing the divide by zero error. If a non-zero value is assigned instead, we need to retest the program to be sure that the results are now correct. Our aspect-oriented approach lets us run the two test interleavings for this retesting, quickly showing the fix removes the error. With noisemakers and value substitution, testing must start from the beginning. However, there is no guarantee that either kind of system will retest the particular error condition. Program replay is not helpful since the captured trace cannot be used to run the new program.

It should be noted that this example requires the CSSAME form to recognize and correctly analyze barrier synchronization. In the program, the barrier ensures that T_3 cannot read the initial value n_0 because it must execute after T_2 and its write. The current version of Odyssey does not properly analyze barrier synchronization, and adds n_0 as another term in the π function. When we implement the CSSAME form for Java, we will correct this limitation.

6 Conclusions

This paper presented preliminary research on a technique to deterministically execute a concurrent program, based on a combination of compiler analysis and aspect-oriented programming. We execute the Java equivalent of the CSSAME form and use aspects to control its execution by overriding the return values of the π functions to return specific values.

The main advantages of this approach are threefold. First, we can deterministically execute the concurrent program, examining all interleavings for a given use of a shared variable. This allows for both testing of code and retesting of code fixes. Most importantly, we can test cases independently of the execution timing of the program. Second, we do not require an execution trace of the program since we are deterministically executing it, not replaying it. Third, this method works even if the shared variables are not protected with synchronization. Although this work is preliminary, we believe it holds promise as the basis for a testing and debugging tool for concurrent programs.

Acknowledgements

This research was supported by the Natural Science and Engineering Research Council of Canada and the University of Waterloo.

References

[1] M. Biberstein, E. Farchi, and S. Ur. Choosing among alternative pasts. In *Proc. 2003 Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2003.

[2] M. Biberstein, E. Farchi, and S. Ur. Fidgeting to the point of no return. In *Proc. 2004 Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2004.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - Second generation of a Java model checker. In *Proc. Workshop on Advances in Verification*, pages 130-135, 2000.

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

[5] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proc. 2002 International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[6] J.-F. Collard. Array SSA for explicitly parallel programs. In *Proc. 5th Intl. Euro-Par Conf., LNCS* vol 1685, pages 383-390, 2005.

[7] S. Copty and S. Ur. Multi-threaded testing with aop is easy, and it finds bugs! In *Proc. 11th International Euro-Par Conf., LNCS* vol 3648, pages 740–749. Springer-Verlag, 2005.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[9] G. Hwang, K. Tai, T. Huang. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4):493-510, 1995.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. 15th European Conf. on Object–Oriented Programming, LNCS* vol 2072, pages 327–353. Springer–Verlag, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conf. on Object-Oriented Programming, LNCS* vol 1241, pages 220–242. Springer-Verlag, 1997.

[12] J. Lee, S. Midkiff, and D. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proc. 10th Workshop on Languages and Compilers for Parallel Computing*, 1997.

[13] S. MacDonald, J. Chen, and D. Novillo. Choosing among alternative futures. In *Proc. Haifa Verification Conference, LNCS* vol 3875, pages 247–264. Spring-Verlag, 2005.

[14] D. Novillo. *Analysis and Optimization of Explicitly Parallel Programs*. PhD thesis, Dept. Comp. Sci., Univ. of Alberta, 2000.

[15] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *Proc. 1998 International Conf. on Parallel Programming*, pages 356–364, 1998.

[16] D. Novillo, R. Unrau, and J. Schaeffer. Optimizing mutual exclusion synchronization in explicitly parallel programs. In *Proc. 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 128–142, 2000.

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

[18] S. Stoller. Testing concurrent Java programs using randomized scheduling. *Electr. Notes Theoretical Computer Science*, 70(4), 2002.

[19] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *Proc. 2003 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 24-33, 2003.