Accelerating Race Condition Detection through Procrastination *

Thomas Arts John Hughes Ulf Norell Nicholas Smallbone Hans Svensson
Chalmers University of Technology
{arts,rjmh,ulfn,nicsma,hanssv}@chalmers.se

Abstract

Race conditions are notoriously frustrating to find, and good tools can help. The main difficulty is *reliably* provoking the race condition. In previous work we presented a *randomising scheduler* for Erlang that helps with this task.

In a language without pervasive shared mutable state, such as Erlang, performing scheduling decisions at random uncovers race conditions surprisingly well. However, it is not always enough. We describe a technique, *procrastination*, that aims to provoke race conditions more often than by random scheduling alone. It works by running the program and looking for pairs of events that might interfere, such as two message sends to the same process. Having found such a pair of events, we re-run the program but try to provoke a race condition by reversing the order of the two events.

We apply our technique to a piece of industrial Erlang code. Compared to random scheduling alone, procrastination allows us to find minimal failing test cases more reliably and more quickly.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Distributed debugging

General Terms Verification

Keywords QuickCheck, Race Conditions, Erlang

1. Introduction

Now that multicore processors are ubiquitous, concurrent programming has become as inescapable as it is difficult. The Erlang programming language [Arm07] was designed to make concurrent programming easy, by making common concurrency mistakes impossible. Erlang processes do not share memory, and thus cannot corrupt each others' data. Erlang data structures are immutable, and thus can be freely copied between process heaps, or between distributed nodes. Erlang processes communicate and synchronize by passing immutable messages from one process to another. These design decisions make *data races*, the scourge of concurrent imperative programming, absolutely impossible.

Nevertheless, Erlang programmers make plenty of concurrency errors. The order of message delivery can vary, leading to scheduling-dependent behaviour. Erlang processes can access global tables managed by the virtual machine—admittedly via an atomic API—but table entries can be used as global variables to recreate the same kind of data races found in other languages. The file store also represents a global state that can lead to race conditions between processes. Erlang's design ameliorates, but does not eliminate race conditions—and thus, tools for race condition testing are still highly relevant.

In previous work, we have implemented a randomizing scheduler for Erlang, and used it to find race conditions in industrial Erlang code [CPS+09]. While data races take only a limited number of forms [VTD06], races in message-passing programs are harder to characterize, and so we detected races in a *black box* manner, as violations in *serializability* of an API under test. We combined our scheduler with a random testing tool, QuickCheck [AHJW06], which generates and simplifies random tests containing sequential and parallel invocations of the API under test. In this way we were able to find *minimal test cases* that exhibit serializability violations, and from them diagnose the underlying race conditions with relative ease.

The search for minimal failing test cases proceeds in two phases: first we search for *any* failing test, then having found one, we search for *simplifications* of that test that also fail. In both phases, we need to determine whether a candidate test case can provoke a race, and in our previous work we did so by running each test many times with different random schedules. If the race we are looking for occurs only rarely, then we may need to run each test case very many times to determine with reasonable accuracy whether or not the race is present. Failure to provoke the race may lead either to failure to find a failing test at all (in the first phase), or failure to simplify the failed test to a minimal example. In our case study, we found we needed to run each smaller test around 10–100 times to obtain good results. This makes testing quite slow.

Finding *data races* in imperative programs by random scheduling is much more difficult, because we must schedule each memory access to shared data, rather than larger scale atomic operations such as message delivery or table access. This motivated Sen to introduce *race directed* random testing [Sen08], in which a test is run once using a random schedule, *possible races* are identified from this first run, and the test is then run again many times using schedules which are specially constructed to provoke each possible race. In Sen's case, a possible race consists of two memory accesses A and B to the same location in different threads, whose execution potentially could be reversed, and the specially constructed schedule tries to delay A until after the execution of B. We call this process of delaying A "*procrastination*". While Sen's paper discusses delaying memory accesses, it is clear that the same idea can be applied to the schedules we construct for Erlang programs—and that, by improving the probability of provoking a race, it could enable us to find minimal failing test cases much more quickly.

* This project was supported by the EU FP7 project ProTest, grant nr. 215868

Erlang'11, September 23, 2011, Tokyo, Japan.

The contributions we present in this paper are

- We transferred to Erlang the idea of using potentially conflicting actions to guide a search for race conditions, which is explained in Section 3.
- Validation of procrastination on an industrial example (Section 4). We show that procrastination is effective and that first-order procrastination seems more useful than higher-order procrastination for this example.
- We extended PULSE to be able to run infeasible schedules (Section 3), in order to re-use as much as possible from a given schedule. This saves a lot of expensive analysis that other approaches need to deal with.
- We experimented with procrastination while shrinking test cases (Section 6) and noticed that in particular the ability to re-use a schedule was very effective (Section 6.3).

In Section 2 we provide background on QuickCheck and PULSE the framework to which we added procrastination. In Section 7 we compare our approach with other approaches to race detection in different contexts.

2. Background

QuickCheck

QuickCheck [CH00] is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in each case, and reports cases for which the property fails. Recent versions also "shrink" failing test cases automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a "minimal"[1] failing case, which often makes the root cause of the problem very easy to find.

Quviq QuickCheck is a commercial version that includes support for model-based testing using a state machine model [Hug07]. It has been used to test a wide variety of industrial software, such as Ericsson's Media Proxy [AHJW06] among others. State machine models define a set of API calls to include in test cases, pre- and post-conditions for each call, and the corresponding state transitions on a model state. QuickCheck then generates well-formed call sequences (satisfying all preconditions, for example), executes them, and checks postconditions with respect to the model state.

Although these state machine models specify the *sequential* behaviour of the API under test, surprisingly, they can also be used to test for race conditions! Assuming that the API calls are intended to behave atomically, then we can generate parallel call sequences and adjudge the test by determining whether there is any *interleaving* of the calls that can explain the actual results observed. Our parallel test cases consist of a random sequential prefix to put the system into a random state, followed by two or more random call sequences executed in parallel. If there is no serialization of the test case that satisfies the postconditions in the model, then the test fails. Details of the method can be found in [CPS$^+$09].

Note that this is a form of *black box* race condition testing: we are only interested in *races that cause a violation of the postconditions*, not in races that lead to non-deterministic, but still *valid* results. Also, note that the approach is only applicable to APIs that are intended to behave atomically. This is of course a limitation, but very many APIs do have this property (or at least, important parts of them do), and the payoff is that very little extra work is required to reuse a sequential state machine model for parallel testing too. (In many cases, it simply requires changing a call of `commands` to a call of `parallel_commands`.)

[1] Minimal in the sense that none of the similar, smaller tests failed.

Simplest is to execute the parallel tests using native Erlang concurrency, relying on the inherent non-determinism of execution on a multicore processor to provoke races. However, because of determinism in the native Erlang scheduler—and perhaps in the processor itself—finding races in this manner can be slow. To speed up their detection, we implemented our own randomizing scheduler, PULSE.

PULSE

PULSE is a *user-level scheduler*, sitting on top of the normal Erlang scheduler [CPS$^+$09]. Its aim is to take control over the sources of non-determinism in Erlang programs introduced by scheduling decisions. This means that we can introduce more randomness in schedules, but also that we can repeat a test using exactly the same schedule by simply recording the scheduling decisions: this makes tests repeatable.

Since PULSE is a user-level scheduler, to give it control over a piece of code the code must be *instrumented* to co-operate with PULSE. Rather than forcing the user to instrument their code themselves, we have a `parse_transform` which does this automatically. We also provide a macro that takes care of running a piece of code under PULSE and collecting the results, so that it takes a reasonably small amount of effort to use PULSE during testing.

The central idea is to provide absolute control over the order of relevant events. Relevant events are interaction of a process with its environment, so called *side-effects*. Code instrumentation replaces each call to side-effect containing functions by a function that gives control to PULSE. Of particular interest in Erlang is the way processes interact by message passing, which is asynchronous. Message channels, containing messages that have been sent but not yet delivered, are thus part of the environment and explicitly modeled as such in PULSE. It makes sense to separate side-effects into two kinds: *outward* side-effects, that influence only the environment (such as sending a message over a channel, which does not block and cannot fail, or printing a message), and *inward* side-effects, that allow the environment to influence the behaviour of the process (such as receiving a message from a channel, or asking for the system time).

PULSE controls its processes by allowing only one of them to run at a time. It employs a cooperative scheduling method: At each decision point, PULSE randomly picks one of its waiting processes to proceed, and wakes it up. The process may now perform a number of outward side-effects, which are all recorded and taken care of by PULSE, until the process wants to perform an inward side-effect. At this point, the process is put back into the set of waiting processes, and a new decision point is reached.

In addition to sending and receiving messages between themselves, the processes under test can also interact with uninstrumented code. PULSE then controls the order in which those interactions take place. We allow the programmer to specify which external functions have side-effects. Each call of a side-effecting function is then instrumented with code that `yield`s before performing the real call, allowing PULSE to run another process at that point.

Side-effecting functions are treated as atomic which is also an important feature that aids in testing systems built of multiple components. Once we establish that a component contains no race conditions we can remove the instrumentation from it and mark its operations as atomic side-effects. We will then be able to test other components that use it and each operation marked as side-effecting will show up as a single event in a trace. Therefore, it is possible to test a component for race conditions independently of the components that it relies on.

```
example() ->
  C = self(),
  B = proxy(100, C),
  A = spawn(fun() ->
              C ! hello,
              B ! world
            end),
  receive Msg1 -> ok end,
  receive Msg2 -> ok end,
  {Msg1, Msg2}.

proxy(0, Pid) ->
  Pid;
proxy(N, Pid) ->
  Proxy = proxy(N-1, Pid),
  proxy(Proxy).

proxy(Pid) ->
  spawn(fun() ->
          receive Msg -> Pid ! Msg end
        end).
```

Figure 1. World-Hello with Proxy

A Problem!

As we showed in [CPS+09], randomized scheduling worked well, almost surprisingly well. However, there are still many situations where a completely random schedule fails to expose an error, and where steering the scheduler can improve the search efficiency. To illustrate the problem, consider the example in Fig. 1. In the example we spawn a process `A`, that first sends the message `hello` directly to a process `C`, and then sends the message `world` to `B`, which forwards it through a chain of 100 proxy processes to `C`. In this (contrived) example there is an obvious race between the two messages, i.e. whether C first receives `hello` or `world`. Still, in practice, regardless of whether the normal Erlang scheduler or PULSE is used, `hello` will never[2] arrive after `world`. If we use PULSE to schedule a run of the example we have to choose, 100 times in a row, to forward the `world` message instead of delivering `hello`. This will happen in 1 out of 2^{100} cases.

3. Procrastination

The main idea behind the procrastination technique is simple: identify potentially conflicting actions, and use the potential conflicts to steer the scheduler. The idea is inspired by race-condition testing of C and Java programs [LCC10, SBN+97], where conflicting memory accesses are recorded and used to steer scheduling.

PULSE records all scheduling decisions taken during program execution. Below is an excerpt from a schedule obtained running the "world hello" example. Here the only side-effect is message delivery: `{A,{deliver,B}}` means that a message from `B` was delivered to `A`. Other side-effects would show up as `yield`s, where `{A,yield,{Fun,Args}}` means that process `A` performed the side-effecting function `Fun` (with arguments `Args`).

```
[{root-proxy99, {deliver, root-example.A }},
 {root,         {deliver, root-example.A }},
 {root-proxy98, {deliver, root-proxy99   }},
 {root-proxy97, {deliver, root-proxy98   }},
 ...
 {root,         {deliver, root-proxy     }}]
```

[2] As in: not in our lifetime!

Given a schedule we can identify potential conflicts. For message delivery there is a potential conflict whenever a process receives messages from two other distinct processes. In the case of user defined side-effects, we rely on the user to define which operations potentially conflict, via a simple call-back function.

As an example consider the fictitious schedule in which process `A` receives a message from both `B` and `C`:

```
[{process_A, {deliver, process_B}},
 {process_B, yield, {do_it,[1,2,3]}},
 ...
 {process_A, {deliver, process_C}},
 ...]
```

Our algorithm detects a potential conflict between the two deliver actions in the schedule. To use this conflict to steer PULSE we create the re-ordered schedule:

```
[{process_B, yield, {do_it,[1,2,3]}},
 ...
 {process_A, {deliver, process_C}},
 {process_A, {deliver, process_B}},
 ...]
```

By feeding the re-ordered schedule to PULSE we delay (procrastinate!) the delivery from `process_B` until after the delivery from `process_C`. However, there is a caveat, namely that the re-ordered schedule might not be feasible. For instance, it might be that the message from `process_C` is a response to a request that `process_A` makes after receiving the message from `process_B`. In this case, when running the re-ordered schedule, we will get to the point where we're supposed to deliver the message from `process_C` but there will be no such message waiting to be delivered.

There are two main solutions to the problem with infeasible schedules: (1) make the analysis more exact to avoid creating infeasible schedules, or (2) allow the scheduler to follow infeasible schedules in some way. We opted for the second solution, and adapted PULSE accordingly. The new version of PULSE tries to follow the given schedule, but whenever there is a scheduling decision that is infeasible (such as delivering a message that has not in fact been sent—such decisions are easy to detect at runtime), it is discarded and PULSE tries the next action in the schedule. With this relaxed strategy PULSE might run out of schedule to follow—if this happens then it reverts to its original, purely random, strategy. Interestingly, in most of the related work, the opposite approach is taken, and various techniques such as "happens before"-relations are used, e.g. [LCC10]. There is good reason for this extra analysis, since an infeasible schedule in a normal scheduler (for example the Java VM) might result in a dead-lock of the whole system. Here, since we have full control of the scheduler, we are in a better position and can take the simpler path without expensive analysis.

It is easy to detect at runtime when a scheduling decision is infeasible: we simply look at the set of actions that we would've chosen from had we been following a random schedule, and check that the action we are about to take is in that set. This also makes it impossible for procrastination to behave *incorrectly*—i.e. to fail to respect the semantics of Erlang—since the behaviours that we provoke are ones that our purely random scheduler could also have provoked, given enough luck.

3.1 Procrastinating World Hello

What happens now if we apply procrastination to the "world hello" example? We have a schedule from PULSE as shown before where there is actually only one potential conflict, namely between the delivery from `root-example.A` and the delivery from `root-proxy` to `root`. (Where `root` is `C` in the example, `root-example.A` is `A`

and `root-proxy` is the last process in the proxy chain.) Our reordered schedule would then be:

```
[{root-proxy99, {deliver, root-example.A }},
 {root-proxy98, {deliver, root-proxy99   }},
 ...
 {root,         {deliver, root-proxy     }},
 {root,         {deliver, root-example.A }}]
```

If we supply this schedule to PULSE the result of running the program is what we tried to achieve, namely `{world,hello}`. (This process is normally automatic, we show the concrete steps as an explanation.)

```
27> pulse:run(fun() -> example() end,
              [{schedule,ReOrderedSchedule}]).
...
{world,hello}
```

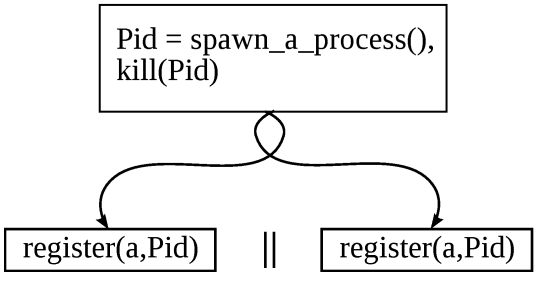
4. The ProcReg Example

Procrastination works well in the simple "world hello" example—but how does it perform in a more realistic setting? To investigate this question, we applied it to the industrial case study used in our previous work [CPS+09]—the proc_reg *process registry*.

In Erlang, every process has a unique, dynamically assigned process identifier (*pid*); the only way to communicate with a process is via its pid. To enable processes to find each other's pids, Erlang provides a *process registry*—a kind of local name server—which associates names with pids. The registry provides an API with operations to register a process with a name, look up the pid associated with a name, and unregister a name. It is heavily used to provide access to system services: a newly started Erlang node already contains thirteen registered processes.

However, the built-in registry does impose sometimes-unwanted restrictions. Names are restricted to be Erlang *atoms*, rather than more general terms. No process may simultaneously be registered with two different names. To lift these restrictions, Wiger developed an extended process registry in Erlang—much easier to modify than the one built into the virtual machine. Wiger's code has been in use in Ericsson products for several years [Wig07].

Our case study is a prototype version of Wiger's registry, with a known race condition. This prototype consists of a registry server that responds to client requests and stores registration data in a so-called *ets table*—hash tables built in to the virtual machine. The use of a single server process serializes operations on the ets table. But, as an optimization, the prototype performs some ets table operations on the client side—thus introducing the possibility of a race condition. And indeed, a race condition occurs. One case that can provoke it is illustrated by this diagram:



Here a process is created and killed, resulting in a dead process, registered with the name `a`, then registered twice in parallel under the same name. The intention is that the registry contain only *live* processes, and that requests to register a dead pid are ignored. For consistency with the built-in registry, registering a dead pid should return `true`. But in this example, one of the parallel calls of `register` occasionally raises an exception instead.

Our QuickCheck specification of the registry models the state as a set of alive pids, a set of dead pids, and a set of name/pid pairs. The specification also defines transition functions on this state for each operation, and checks postconditions against this model state. In this example, the model tells us that `Pid` is a dead process, and therefore the postcondition for `register` requires the result to be `true`. We adjudge parallel test cases like this one by determining whether there is any serialization of the test case in which all the postconditions hold. In this example there are only two serializations, and in both of them, all the calls of `register` should return `true`. When an exception is raised instead, then our QuickCheck property fails.

In fact, this prototype version was abandoned in 2005, after QuickCheck revealed the *existence* of a race condition, but we were unable to diagnose it. At that time we could only generate very large failing test cases. It was not until 2009 that we were able to *shrink* the failing test cases to the example above, and with its help, find and correct the error in the code. But now we know what the bug is; in this paper our focus is on the *performance* of our testing.

The QuickCheck properties that we use to test the process registry generate sequences of operations to spawn, kill, register, or unregister a process, or look up a name, all with equal probability. We generate parallel tests by splitting the tail end of the sequence into two parallel sequences (respecting preconditions). We then *run* these parallel tests in a variety of different ways.

4.1 Measurements

The simplest approach is just to run the tests using the built-in Erlang scheduler on a multicore computer (we used an eight core i7 machine with 16GB RAM for all our tests). That is, we relied on the non-determinism inherent in parallel execution to provoke race conditions. When we did so, we found that slightly less than 0.1% of tests failed, which enabled us to find 59 failing cases in 30 minutes of testing (or a mean-time-to-failure of 30.5s). Note that since the known race condition depends on trying to register a dead process twice in parallel, then many generated test cases cannot fail at all. For this particular bug, around 8% of the generated test cases can fail. On the other hand, even a test case that potentially provokes the race will pass in many cases. We found that the failing tests we found in this way, relying solely on parallel execution, failed in approximately one of every 30 runs.

When we ran the same tests using PULSE, our randomizing scheduler, then we saw 741 failures in 30 minutes—more than twelve times as many (mean-time-to-failure 2.43s, c.f. Figure 2). This is despite the overheads that PULSE imposes—each test ran 1.5 times slower under PULSE than with the native scheduler. This demonstrates the benefits of randomising the schedule very clearly.

5. Procrastinating ProcReg

We need to refine the ideas of Section 3 a little in order to be able to cope with real-world Erlang code such as ProcReg.

Dealing with side effects Unlike the "world hello" example, the ProcReg example contains side effects other than message passing, namely reads and writes to the Erlang mutable term storage, *ets*.

As described in Section 2, PULSE can deal with arbitrary side-effecting operations: we may register the ets functions as having a side effect. Thereafter, whenever a process wants to read or write

to an ets table, it will first ask PULSE for permission; at this point PULSE might delay the execution of the process arbitrarily.[3]

These side effects are recorded as decisions in the schedule that PULSE produces. Thus, if our program manipulates ets tables then we will get entries such as

```
{process_A, yield, {ets, insert, [...]}}
{process_B, yield, {ets, lookup, [...]}}
```

in our schedule. Each entry represents the point when a process was *given permission* to execute a particular side effect. We can apply procrastination to side effecting functions in *exactly the same way* that we do with message deliveries: given a schedule, we identify pairs of function calls that might conflict (such as an `ets:insert` and an `ets:lookup`), and attempt to delay the execution of the first function call until after the execution of the second.

Not all pairs of function calls can conflict, and it is not useful to try to swap the order of two function calls that do not conflict. We allow the user to specify which function calls can conflict: this reduces the total number of procrastinations we try by ruling out some procrastinations that can never be fruitful. PULSE will assume that any two side-effecting functions can conflict if not specified otherwise: this is safe, but by giving more fine-grained conflict information the user can reduce the number of fruitless procrastinations that are tried. In the case of ets, we say that two operations conflict if they operate on the same ets table and at least one of them is a write.

Implementing procrastination Our implementation of procrastination consists of four steps: (1) run the test case with a random schedule using PULSE, (2) for each scheduling decision identify later potentially conflicting decisions, (3) for each scheduling decision try, by running a modified schedule with PULSE, to move it past the *last* potentially conflicting decision, using the techniques of section 3 to detect infeasible schedules, and record which potentially conflicting decisions we managed to move it past (we call these pairs of potentially conflicting schedule decisions *feasible procrastinations*), and (4) for each such feasible procrastination, construct and execute the procrastinated schedule.

Controlling the amount of procrastination There may be many possible ways to procrastinate a given schedule. Thus, for a given test case QuickCheck might spend a considerable amount of time repeating the test case, trying various procrastinations for that test case. However, if the test case is one that cannot provoke a bug then this will get us nowhere! So there is a trade-off between exploring many procrastinations of a single test case and trying many test cases. Therefore we allow the user to choose how many of the feasible procrastinations to try for each schedule.

Higher-order procrastination In our basic approach we find two conflicting scheduling decisions and try to reverse their order. However, it might in general be necessary to procrastinate several decisions at once—especially if the schedule is large. We call this *higher-order* procrastination. Since procrastinating one decision may make other procrastinations possible or impossible, it does not really make sense to apply several simultaneous procrastinations to one schedule. Instead, we first apply one procrastination and re-run the test case to get a *new* schedule. If that procrastination succeeded we can then apply further levels of procrastination to the *new* schedule.

Since the number of procrastinated schedules is exponential in the number of levels of procrastination we try, this technique is not always useful: we allow the user to say if we should apply it or not and how many levels of procrastination to try.

[3] This treatment assumes that side effects are atomic, because we never attempt to execute two side effecting operations in parallel—a perfectly reasonable assumption in our case, but one which does not make sense for traditional shared state concurrency.

Scheduler	**Level**	**Limit**	$Time\ (s)$
VM			30.5
PULSE			2.43
PULSE	1	$\langle 10 \rangle$	1.32
+ procrastination	1	$\langle \infty \rangle$	3.01
	2	$\langle 5, 1 \rangle$	3.90
	2	$\langle 10, 3 \rangle$	6.47
	2	$\langle 10, 10 \rangle$	8.91
	2	$\langle 20, 20 \rangle$	16.1

Figure 2. Mean time to failure

5.1 Measurements

We re-ran the experiment of section 4.1—testing ProcReg and recording how many failing test cases QuickCheck was able to find in 30 minutes—but using procrastination. The results are shown in Figure 2.

The "Level" column indicates how many levels of procrastination we tried (see paragraph "Higher-order procrastination") and the "Limit" column shows how many procrastinated schedules we tried at each level (see paragraph "Controlling the amount of procrastination"), ∞ meaning that we try all possible procrastinations. "Time" is, as before, the mean time to failure.

We can see that procrastination almost doubled the number of failures we were able to provoke in 30 minutes, compared to using random scheduling alone—a reasonable success. However, higher-order procrastination actually made things *worse* than not procrastinating at all. This is because we repeat each test case over and over again even if that test case simply cannot fail—too much procrastination is a bad thing. According to the table above, for ProcReg the happy medium is one level of procrastination with ten decisions from each schedule chosen for procrastination.

Measuring failure reproduction The results above may be deceptive because they take no notice of *which* race conditions each testing method is able to provoke. If a method only works well with simple test cases, it will still get a low mean time to failure if it runs each test quickly—even if it is completely unable to provoke race conditions in more complicated cases.

Therefore, we also tried using each testing method to reproduce bugs that were found by the *other* testing methods. We collected a variety of bug-provoking test cases that were found by each testing method, and for each test case, we repeatedly ran each method on that test case until it reproduced the bug, recording how many attempts we needed to find the bug in each case and how long it took.

The results are shown in figure 3. The "Origin" column shows which method we used to originally find the bug and "Scheduler" shows which method we were using to try to reproduce the bug. We list the average amount of time taken to make each test case fail and the average number of times we had to repeat each test case before it failed.

Using the virtual machine's scheduler performs very poorly, as we expect. Even on test cases that were originally found without the help of PULSE, we have a very low probability—about $\frac{1}{100}$ on average—of reproducing the bug when we run the test case. For test cases that were found by PULSE, we need to repeat each test case thousands of times, and because of that, using the virtual machine increases the time taken to reproduce the bugs by a factor of 100.

PULSE is easily able to reproduce bugs that were found using the VM's scheduler, even without procrastination—and here

Origin	Scheduler	Level	Limit	Time (s)	Avg #tests
VM	VM			0.82	101.3
	PULSE			0.14	2.0
	PULSE	1	$\langle 10 \rangle$	0.28	1.0
	+ procrastination				
PULSE	VM			20.16	3584.1
	PULSE			0.24	13.6
	PULSE	1	$\langle 10 \rangle$	0.36	1.16
	+ procrastination				
PULSE	VM			35.03	7126.6
+procrastination	PULSE			0.83	70.9
	PULSE	1	$\langle 10 \rangle$	0.32	1.54
	+ procrastination				

Figure 3. Failure detection—grouped by origin

procrastination slows things down by a factor of two. This is because we can only find very small counterexamples using the VM's scheduler and for those random scheduling is enough.

For bugs that were found by PULSE without the help of procrastination, it might *seem* from figure 3 as though using procrastination greatly reduces the number of tests we need to run. However, using procrastination, each "test" executes the program several times—once for each procrastinated schedule—so the number of tests is a deceptive measure. Looking at the time taken shows that in reality PULSE without procrastination wins here, too.

However, on bugs that we originally found with the help of procrastination, PULSE with procrastination is the surefire winner. We see that for those test cases, using procrastination is more than twice as fast as not using it, and more than 100 times faster than using the VM's scheduler. Since each test using procrastination executes the test case at most 11 times (we are using a limit of 10 procrastinations), we can see that using procrastination reduced the number of times we had to execute the program by a factor of at least 3.

Thus, procrastination really is better at provoking complex race conditions than random scheduling alone. There is one test case in particular for which PULSE without procrastination has great trouble. It is short—7 commands long, and the test case itself is completely sequential—but PULSE without procrastination takes on the order of 1000 attempts to find it on average. Using procrastination—even first-order procrastination with a limit of 10 procrastinated schedules—we find the bug first time, every time. We conjecture that in more complex systems—where there are more scheduling decisions to be made than in our ProcReg example—there are many such bugs that require extremely good luck to find using random testing but where adding a small amount of procrastination can reliably provoke the error. In particular, with a purely random scheduler, once a particular action is *possible* then it is likely to be chosen within a few steps, since at each step there is a reasonable probability that the scheduler will choose that action. Thus an action will not be delayed for very long in a purely random scheduler; this is what happens in the "world hello" example of section 2, where to provoke an error we need to delay a message delivery for 100 steps and have only a one in 2^{100} chance of doing that. With procrastination, we can delay some decisions for an arbitrary amount of time.

6. Shrinking Counterexamples

In order to understand an error it is important that the failing test case is reasonably small. In the case of the process registry example it was not until we were able to shrink the failing test cases to the one shown in Section 4 that the error could be successfully diagnosed. To achieve this QuickCheck, having found one failing test case, then shrinks it by trying many smaller, similar tests. If any of these fail, the original failing test is replaced by the smaller one and the shrinking process continues. When no smaller tests fail, then a "minimal" failing case has been found.

Since the outcome of a test is non-deterministic, it might be that a test succeeds even though the test case is one that sometimes provokes a bug in our program, if when we run the test we are unlucky with the scheduling decisions we choose. If this happens for all of the candidate shrunk test cases, the shrinking process stops with a result that is not necessarily minimal. Thus it is particularly important during shrinking that test cases that contain the race condition reliably fail when they are executed.

6.1 Improving shrinking by repeating tests

One method for improving the reliability of shrinking is to run each test N times during the shrinking process. If the test fails in any one of these runs it is considered to be a failing case. This increases the likelihood of successfully identifying test cases that contain the race condition at the expense of making shrinking more time consuming. This is the method we used originally to find the minimal failing case for the process registry.

To measure the performance of shrinking we ran the 45 random test cases we collected in which the race condition is present, varying the scheduler (the Erlang virtual machine scheduler or PULSE) and the value of N. For each execution we measured the length of the shrunk test case, the time taken by the shrinking process, and whether or not the shrunk test case was minimal[4]. We measured this 10 times to reduce the impact of coincidentally getting the best performance. Figure 4 shows the results for each scheduler and value of N.

The first observation we can make is that when running with PULSE, increasing N improves the quality of the shrinking as we expect, but when running with the Erlang scheduler it has hardly any effect. This is likely due to the Erlang scheduler being quite deterministic and not gratuitously delaying actions, so if a test case does not reveal a race condition the first run it is unlikely to do so in later runs.

Another interesting observation is that going from $N = 1$ to $N = 10$ does not affect the shrinking time significantly. This can be explained by the fact that when shrinking, QuickCheck first tries to take big shrinking steps, throwing away much of the original test case, and only if that fails tries smaller steps like taking out a single command. By repeating each test case during shrinking, QuickCheck is more likely to provoke an error with the

[4] We deem a test case minimal if we could not provoke a bug in any of its shrinkings within 50 tests using any of the methods at our disposal.

Scheduler	N	Time (s)	Avg Len	Max Len	Minimal
VM	1	1.74	11.4	45	0.0 %
	10	2.68	10.4	45	0.5 %
	25	4.19	9.7	47	2.2 %
	100	29.04	11.5	48	1.0 %
PULSE	1	2.42	12.0	38	0.7 %
	10	3.20	7.1	26	17 %
	25	5.14	5.9	28	43 %
	50	8.22	5.2	27	67 %
	100	14.28	4.6	9	90 %

Figure 4. Shrinking performance when repeating each test N times.

large shrinking steps, which reduces the number of steps required to reach a minimal test case.

6.2 Shrinking with procrastination

Procrastination works in a similar way to the repeated test strategy described in the previous section in that we are executing each test case several times looking for a failure, but instead of blindly rerunning the test hoping for a random schedule that reveals a race condition, we choose schedules that are likely to do so. Thus we would expect shrinking with procrastination to outperform repeated testing with random schedules. This is indeed the case as shown by the shrinking results for procrastination in Figure 5.

We can see that using procrastination we find minimal test cases around 60% of the time for all of the parameter values that we tried, but the time spent shrinking varies a lot. In this example, it seems that sticking to first order procrastination and only trying a small number of conflicting actions is most efficient. In fact this is faster than any of the previously attempted shrinking strategies, and reaching the same quality of shrinking using repeated tests would take somewhere around 2 to 3 times longer.

Which parameters to give when procrastinating depends very much on the nature of the race condition: If the race condition is present in many test cases but requires very specific scheduling to be revealed we need thorough procrastination. If, on the other hand, few test cases contain the race condition—which is to say that many of the candidate shrunk test cases will not be able to provoke the bug—then thorough procrastination will waste a lot of time on test cases that cannot fail. In our example only around 8% of the test cases contain the race condition and finding it when it is present seems to be relatively easy, so we can get away with less thorough procrastination.

It is interesting to note that higher order procrastination does improve the quality of shrinking, in particular in the worst case, but it is not obviously better than simply repeating each test 100 times.

6.3 Reusing old schedules when shrinking

When implementing procrastination we had to modify PULSE to enable it to follow schedules that contain infeasible decisions. This led us to another idea for improving shrinking: what if, instead of running the smaller test cases on completely random schedules, we reused the schedule that made the bigger test case fail. That is, we try as far as possible to repeat the scheduling decisions that we know led to a failure for the bigger test case. The rationale for this is that if the smaller test case still contains the race condition, then the scheduling decisions that revealed the race condition for the bigger test case are likely to be valid also for the smaller one.

Naming of processes An issue that showed up when we started to reuse schedules during shrinking, that is not present for procrastination was that we had to be more careful about how processes are named. If a schedule says to deliver a message from process `A` to process `B` it is important that we use the names `A` and `B` for these processes when reusing that schedule.

The names are chosen automatically by PULSE. With our default naming scheme, the name chosen for a process depends on the names of the processes that have already been spawned, since each process needs a unique name. For instance, in the "world hello" example the first process to be spawned in the proxy chain is named `root-proxy`, the second `root-proxy1`, then `root-proxy2` and so on. This means that if a shrinking step removes the spawning of a process that could affect the names of processes that are spawned later in the test, and the schedule from the original test case would not make much sense for the shrunk test case, because the shrunk test case and the original test case would name their processes differently. To solve this problem we record in the schedule when a process is spawned and what name it's given. When following an existing schedule PULSE then reads the process name from the schedule instead of generating a fresh one. This is not a perfect solution, but it is a significant improvement over the previous situation.

Results In the process registry case reusing the schedule during shrinking turns out to work amazingly well (see Figure 6). Shrinking is more than 3 times faster than using procrastination and the shrunk test cases are as small as what we obtained using procrastination. The number of minimal test cases, however, is surprisingly low. The reason for this is that schedule reuse works poorly when shrinking moves commands from one process to another, for instance moving a command from one of the parallel sequences to the initial sequential part. This is because actions performed by the moved command will take place in a different process, so the corresponding scheduling decisions will be invalid. On the other hand it works very well for shrinking steps where the structure of the test case stays the same and we just remove unnecessary commands.

Note that in this case we did not repeat any tests during shrinking. When reusing the schedule this makes less sense, since each repeated test would start from the same schedule. It could still be of some benefit since PULSE reverts to random scheduling when there are no more feasible scheduling decisions.

Combining schedule reuse with procrastination improves both the quality and speed of shrinking, the proportion of minimal test cases goes from 60% to 80% compared to procrastination without schedule reuse and the worst case is significantly improved.

It is still more than twice as fast to just use schedule reuse when shrinking without compromising the quality of shrinking significantly. The reason why procrastination is so much slower is that it is spending quite a lot of time procrastinating shrunk test cases that cannot fail. One possible improvement would be to interleave the procrastinations for all shrinkings, that is, first try each smaller test case once, and if none of them fails try one procrastination for each case, and so on. This should result in similar performance to schedule reuse with no procrastination, but

Level	Limit	Time (s)	Avg Len	Max Len	Minimal
1	$\langle 10 \rangle$	2.25	5.6	27	60 %
1	$\langle \infty \rangle$	3.80	5.6	25	56 %
2	$\langle 5, 1 \rangle$	7.01	5.3	15	66 %
2	$\langle 10, 3 \rangle$	8.93	5.2	14	65 %
2	$\langle 10, 10 \rangle$	10.54	5.2	14	66 %
2	$\langle 20, 20 \rangle$	19.36	5.3	13	63 %

Figure 5. Shrinking performance with procrastination.

Scheduler	Level	Limit	Time (s)	Avg Len	Max Len	Minimal
PULSE			0.74	5.3	14	33 %
Procrastination	1	$\langle \infty \rangle$	3.31	4.6	9	80 %
	1	$\langle 10 \rangle$	1.87	4.6	9	78 %
	2	$\langle 5, 1 \rangle$	5.93	4.6	9	86 %
	2	$\langle 10, 3 \rangle$	10.08	4.6	10	84 %
	2	$\langle 10, 10 \rangle$	12.03	4.6	10	82 %
	2	$\langle 20, 20 \rangle$	21.74	4.6	9	83 %

Figure 6. Shrinking performance with schedule reuse.

with procrastination's quality. We leave the implementation of this strategy to future work.

7. Related Work

Much of the work regarding race condition detection has been focused around imperative (object oriented) programming languages, such as C, C++, and Java. In these languages a data race, in its simplest form, is when a shared piece of data is accessed and updated in a malign pattern. Several techniques have been proposed for detection of such data races, like [OC03] and [SBN+97]. A problem, however, is that a large portion of the potential data races found are benign. As a result, research effort has focused on atomicity and/or serializability violations [FF04, WS06]. By defining *units of work* to be atomic/serializable, it is possible to more accurately detect true concurrency bugs.

A problem with the previously mentioned techniques is that they do not take into account, potential correlations between the shared variables. Thus, it is possible to miss high-level data races and also to report false warnings. Vaziri et al. [VTD06] address this problem with a more involved correctness criterion, namely *atomic-set serializability*. All of the race conditions above, simple data races, high-level data races, atomicity, and serializability, can be characterized by atomic-set serializability. The idea is to choose (parts of) the (object) state as atomic sets and methods/functions as units of work, and identify problematic interleavings of units of work in relation to the atomic sets.

Atomic-set serializability has proved to be accurate in discovering true concurrency bugs [KRDV09], and the tool ASSETFUZZER [LCC10] is based on this technique.

Techniques where the scheduling is randomized has successfully been tried for concurrent, multi-threaded, programs [Sto02, CPS+09]. Although effective, the simple random technique is depending on the fact that harmful schedules are not too hard to find. To improve the situation, active randomized scheduling is used. The idea was introduced by RACEFUZZER [Sen08], and further improved by ASSETFUZZER [LCC10]. The technique is similar to our procrastination: the program is run, the obtained schedule is analyzed for potential conflicts, and finally the program is re-run with a schedule that is (more) likely to trigger a conflict. A problem with this technique is *thrashing* [JPSN09], where tests are failing because the calculated, potentially problematic, schedule is not feasible and the program dead-locks. Luckily, with our relaxed schedules, this is not a problem for us.

In contrast to using a race condition detection criterion, and detecting violations in terms of crashes or memory access patterns, we use properties to decide whether a test has passed or not. When `parallel_commands` is used with QuickCheck, PULSE, and procrastination, the end result is close functionality-wise to ASSETFUZZER.

Another often used approach is to systematically try all possible schedulings, either in the form of ordinary model checking or a repeated-test strategy. CHESS [MQB+08] uses the latter, it systematically generates all (non-equivalent) interleavings for a given test scenario. By using some model checking techniques the number of explored interleavings can be kept at a reasonable level. For Erlang programs, McErlang is a traditional model checker that can find concurrency bugs [FS07]. Although, in theory, the idea to explore all possible execution paths is tempting, there are usually problem for these methods to scale to larger programs. CHESS reports on some success [MQB+08] on this issue, while Kidd et al. [KRDV07] concludes that their approach is not scalable even for medium-sized programs.

8. Conclusions

Testing concurrent programs is difficult, even when the program is written in Erlang. We have shown how we can introduce procrastination into the earlier developed user-level scheduler PULSE. With procrastination, potential race conditions are detected in a given schedule and a new schedule is computed to provoke such potential races. We changed PULSE in such a way that it can run infeasible schedules, i.e., it runs a schedule as far as possible and continues with random scheduling when the given schedule cannot be followed any longer. This allowed us to use computationally inexpensive heuristics to compute new schedules from a schedule with a potential race condition. Nevertheless, our procrastination is very effective in provoking race conditions as our empirical data shows.

We use procrastination in combination with QuickCheck. Test cases are automatically generated and by executing PULSE with procrastination, we are able to find race conditions effectively. A beneficial side-effect of the changes made to PULSE, is that in order to facilitate the usage of infeasible schedules, we are also able to re-use a schedule for a *different* test case. This can be exploited

while shrinking, where we can now re-use the failing schedule when we test a shrunk, very similar test case. For the running example of this paper, this turned out to be amazingly effective. In fact, rather unexpectedly, just re-using of the schedule and no procrastination during shrinking gave the quickest shrinking, albeit not the smallest counterexamples. In our further research we will examine additional examples in order to develop a good strategy in mixing procrastination and schedule reuse during shrinking in order to quickly find the smallest counterexamples.

References

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2006.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proc. of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[CPS+09] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. In *ICFP*, pages 149–160, 2009.

[FF04] C. Flanagan and S.N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 269, April 2004.

[FS07] Lars-Åke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, 2007.

[Hug07] John Hughes. QuickCheck Testing for Fun and Profit. In *9th Int. Symp. on Practical Aspects of Declarative Languages*. Springer, 2007.

[JPSN09] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 110–120, New York, NY, USA, 2009. ACM.

[KRDV07] Nicholas Kidd, Thomas Reps, Julian Dolby, and Ana Vaziri. Static detection of atomic-set-serializability violations. In *Technical Report 1623*, University of Wisconsin-Madison, 2007.

[KRDV09] Nicholas Kidd, Thomas Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. In Neil Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 198–213. Springer Berlin / Heidelberg, 2009.

[LCC10] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, NY, USA, 2010. ACM.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[Sto02] Scott D. Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142 – 157, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[Wig07] Ulf T. Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.