# Healing Data Races On-The-Fly

Bohuslav Křena
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
krena@fit.vutbr.cz

Zdeněk Letko
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
xletko00@stud.fit.vutbr.cz

Rachel Tzoref
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
rachelt@il.ibm.com

Shmuel Ur
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
ur@il.ibm.com

Tomáš Vojnar
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
vojnar@fit.vutbr.cz

## ABSTRACT

Testing of concurrent software is extremely difficult. Despite all the progress in the testing and verification technology, concurrent bugs, the most common of which are deadlocks and races, make it to the field. This paper describes a set of techniques, implemented in a tool called ConTest, allowing concurrent programs to self-heal at run-time.

Concurrent bugs have the very desirable property for healing that some of the interleaving produce correct results while in others bugs manifest. Healing concurrency problems is about limiting, or changing the probability of interleaving, such that bugs will be seen less. When healing concurrent programs, if a deadlock does not result from limiting the interleaving, we are sure that the result of the healed program could have been in the original program and therefore no new functional bug has been introduced.

In this initial work which deals with different types of data races, we suggest three types of healing mechanisms: (1) changing the probability of interleaving by introducing sleep or yield statements or by changing thread priorities, (2) removing interleaving using synchronisation commands like locking and unlocking certain mutexes or waits and notifies, and (3) removing the result of "bad interleaving" by replacing the value of variables by the one that "should" have been taken. We also classify races according to the relevant healing strategies to apply.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Algorithms, Verification

## Keywords

Concurrency, Testing, Self-healing

## 1. INTRODUCTION

The increasing popularity of concurrent programming—for the Internet as well as on the server side—has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as unintentional data races or deadlocks are difficult and expensive to uncover and analyse, and such faults often escape to the field. Having new dual core or hyper-threaded processors (which is the case of all general purpose processors currently in production) in personal computers makes the testing of multi-threaded programs even more important. The programs that used to work well on single-threaded and single-CPU-core processors are now exhibiting problems. As a result, large companies such as IBM, Intel, and Microsoft have now dedicated teams that work on tools and methodologies for the multi-threaded domain.

Work on race detection, the most common way to find concurrency bugs, [33, 35, 23, 28, 17, 2, 3, 6] has been going on for a long time. Race detection tools suffer from the dual problem of having too many false alarms as well as not identifying some of the races. Other available techniques (such as static analysis or model checking) do not solve the problem in a satisfiable way either.

Even with the best testing and/or verification techniques available, problems still escape to the field. Moreover, even if the problem is known, there are situations in which it is not easy to fix it. Examples include situations in which the software is embedded in hardware which has no remote connection or when replacing is very expensive. In such situations, it would be very desirable if the software could fix its concurrency problems itself on-the-fly. This is a strong motivation for developing *software self-healing techniques*, which is an idea that we pursue in this paper.

In general, a self-healing approach may consist of the following steps:

1. **problem detection**—before any self-healing action can be performed, it is necessary to detect that something is wrong with the system,

2. **problem localisation**—when an incorrect behaviour of the monitored system is witnessed, one has to find

the root cause of the problem,

3. **problem healing**—applying a fix to the problem found in the localisation stage,

4. **healing assurance**—by an application of the self-healing action, the system and its behaviour are modified in the hope that the problem will be resolved while no new problem will be introduced to the system. However, it is desirable to check/prove whether this goal was achieved or not.

We apply the above self-healing methodology to fixing *data races in concurrent programs* in the following way:

**Problem detection**. Currently, we are able to monitor execution of concurrent programs written in Java by an instrumentation of the Java byte-code. The crucial difficulty within debugging of concurrent programs—which one must address for a successful development of concurrent software—is a huge number of possible execution interleavings which results in a very low probability of finding a bug during testing. The instrumentation of Java byte-code has therefore an additional aim—not only to monitor the program execution, but also to increase the probability of a concurrent bug exhibition by adding some noise. For the instrumentation and noise injection, we use the tool ConTest that is described in Sec. 6.1 in more detail. For the actual detection of data races on top of ConTest, we then use a modification of the Eraser algorithm [35][1].

**Problem localisation**. Locating the source of the detected problem is often a hard task even for humans. The approach we concentrate here on consists in developing an oracle specialised on a particular class of concurrent bugs, more specifically data races. We, in particular, consider oracles based on looking for pre-specified data race bug patterns in the code with the aid of information collected by the data race detector that is used to detect problems. Another possible approach—which we do not discuss in detail here—is using a large number of tests with different instrumentation points and statistical evaluation [37]. Both of these approaches can be successfully combined with formal methods (like model checking or static analysis) in order to reduce the number of false alarms. Formal methods can be in principle applied without any support, but some of them suffer from false alarms too, or—on the other hand—incur a state explosion problem, and hence do not scale well.

**Problem healing**. Our healing approach depends on the category of races found. There are two major kinds of races: *Atomicity races*—races that are caused by violation of wrong assumptions that some blocks of code will be executed atomically. These races are characterised by the fact that if the block of code is executed without an context switch the race does not occur. We explain how to heal such races on-the-fly. Our healing techniques include reducing the likelihood of a race by changing the probability of the "bad" timing scenario, or removing the "bad" interleavings by introducing an appropriate synchronisation.

*Inherent races*—races not related to atomicity. The result of a computation in this type of race depends only on the order of events between different threads, and not on the fine-grained interleavings. The lost notify bug pattern [16] is one example of such a problem. Another example is simply executing `A=3` in one thread and `A=4` in another. Our healing approach for inherent races is more complicated. First, when we detect a race, we query an oracle about the state of the program ("good" or "bad"). We need the oracle in order to learn which of the orders is better. Then we can try to force the right order. This can be done by changing the probability of the "bad" timing scenario, by overriding results, or by forcing the appropriate order.

**Healing assurance**. One cannot hope that self-healing will work in general under any circumstances. One can find healing techniques (such as introducing new locks) that will always fix the given problem, but—on the other hand—they may cause another problem, such as a deadlock. Conversely, there are techniques that are not risky (such as adding calls to `sleep()`), but that do not guarantee they will solve the given bug. Therefore, once we are about to use a certain healing strategy for a certain specific case, it is desirable to check whether the chosen strategy will be effective and safe (i.e. whether it will solve the encountered problem without introducing any further problems). To tackle this problem, we suggest to use a *suitable formal verification technique* (like model checking or static analysis) albeit in a *limited fashion* allowing one to deal with real-life software systems.

**Plan of the rest of the paper.** In Sec. 2, we define the basic terms used and describe the Eraser algorithm that we apply for detecting data races. In Sec. 3, we describe healing of atomicity problems. Healing of inherent races is then covered in Sec. 4. We briefly discuss healing assurance in Sec. 5. We discuss our preliminary implementation of some of the presented ideas and give the first experimental results in Sec. 6. Finally, we briefly comment on the related work in Sec. 7 and we conclude in Sec. 8.

## 2. BACKGROUND

A *data race* in a concurrent program occurs when two threads access a shared memory location, the accesses are unordered by any (explicit or implicit) synchronisation, and at least one of these accesses is a write access. Data races are usually considered to be bugs as they can lead to an unpredictable behaviour of the program.

Since deciding whether a program contains a data race is computationally hard, most research on data race detection focuses on the so-called *apparent data races* [30]. These are races in which two threads access a shared memory location, at least one of theses accesses is a write access, and no explicit synchronisation mechanism (such as a mutex or a barrier) prevents the threads from a concurrent access. Apparent data races are approximations of data races, i.e. not every apparent race is a true data race. However, they are much easier to detect.

The main approaches to detection of apparent data races include static and dynamic analysis. The *static analysis* approach [4, 15, 17, 25] is based on a compile-time analysis of the code. This approach is able to analyse the whole code at once, but it often suffers from many false alarms. The *dynamic analysis* approach tries to detect a race in a specific execution of the program. In this approach, information is collected during the execution and analysed either on-the-fly [10, 13, 14, 23, 31, 34, 35, 38, 32] or when the execution

[1]Our choice of Eraser for the current version of our healing framework was motivated by the possibility of its easy implementation. In the future, we plan to experiment with other race detection algorithms such as [19].

terminates [1, 28, 29]. Dynamic analysis suffers from less false alarms than the static approach since it reports apparent races that actually occurred during the execution. However, it is less complete in the sense that it concentrates on specific executions of the program. Other approaches to detection of apparent data races include, e.g. *model checking* [12], which can in theory detect all data races without producing any false alarms (at least when the system can be viewed as finite-state). However, model checking is extremely costly, and despite the recent advances in software model checking [5, 9, 22], it is still not applicable beyond relatively small pieces of the most critical code.

One of the most popular on-the-fly race detection algorithms is the *Eraser algorithm* [35] which was originally proposed to find data races in C programs. Since our race detector implementation relies on Eraser, we describe it in more detail in Subsection 2.1.

A stronger requirement than a lack of races is *atomicity*. A block of code is *atomic* [20] if for every interleaved execution of the program in which the block is executed, there is an equivalent run of the program where the block is executed sequentially (without interleaving with other threads). It is argued that many faults in multi-threaded code (e.g. many occurrences of data races) are the result of non-atomic blocks [18, 21, 20, 39, 40]. Similarly to race detection, atomicity checking includes static analysis methods [21, 20, 24, 40] and dynamic analysis methods [18, 39], which check atomicity at run-time.

## 2.1 The Eraser Algorithm

The *Eraser algorithm* [35] is based on the consideration that every shared variable should be protected by a lock. Since Eraser has no way of knowing which locks are intended to protect which variables, it must deduce the protection relation from the execution history. For each shared variable $v$, Eraser maintains the set $C(v)$ of candidate locks for $v$. This set contains those locks that have protected $v$ for the computation so far. That is, a lock $l$ is in $C(v)$ if, in the computation up to that point, every thread that has accessed $v$ was holding $l$ at the moment of the access. When a new variable is initialised, its candidate set $C(v)$ contains all possible locks. When the variable is accessed, Eraser updates $C(v)$ by the intersection of $C(v)$ and the set of locks held by the current thread. If some lock $l$ consistently protects $v$, it will remain in $C(v)$ till the end of the execution run.

In order to reduce false alarms, Eraser takes into account that the following situations will not cause any problem despite they can be determined as data races by the definition given above:

- A shared variable can be initialised without holding a lock if it becomes really shared only after its initialisation.
- A shared variable is written during the initialisation only and it is read-only after.
- Read-write locks allow multiple readers to simultaneously access a shared variable but allow an access of a single writer only.

Eraser reflects these situations in the following way. As long as a variable has been accessed by a single thread only, reads and writes have no effect on the candidate set $C(v)$. Since simultaneous reads of a shared variable are not races, Eraser reports races only after an initialised variable has become write-shared by more than one thread. These assumptions lead to introducing internal states *Virgin*, *Exclusive*, *Shared*, and *Shared-Modified* for each shared variable with the following meanings (and transitions depicted in Fig. 1):

- *Virgin*—the variable has not been initialised yet.
- *Exclusive*—the variable is accessed only by the thread which initialised it.
- *Shared*—the variable is read by multiple threads.
- *Shared-Modified*—the variable is read and written by multiple threads.
- *Race*—a data race on this variable has been detected (due to no or a wrong lock has been used when accessing the variable).

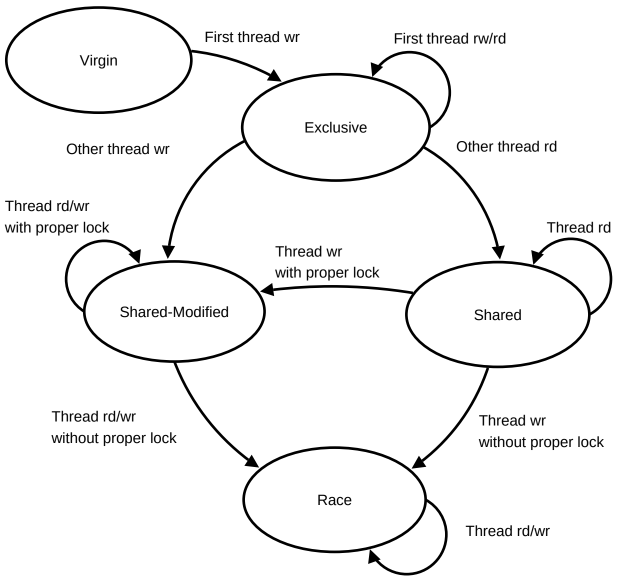

**Figure 1: Possible states of a shared variable**

The basic Eraser algorithm has been modified in various ways in the literature. In our approach, in order to further reduce false alarms, we implemented an extension of Eraser by the *ownership model* [18, 38, 11]. In this approach, building of the set $C(v)$ of candidate locks is delayed to reflect the following typical initialisation pattern used in Java programs: A variable $v$ is created by a thread $t_1$, and then another thread $t_2$ (which is, e.g. started only after the creation of $v$ is finished) accesses $v$ and writes an initialisation value into it. To reflect this pattern in Eraser, the *Exclusive* state is split into two states—$t_1$ accesses $v$ in *Exclusive1*, then once $t_2$ starts working with $v$, the state is changed to *Exclusive2*. Subsequently, if another thread $t_3$ different from $t_2$ (but possibly equal to $t_1$) accesses $v$, we set $C(v)$ to the locks held by $t_3$, go to either *Shared* or *Shared-Modified* and further behave as in the basic Eraser.

Clearly, the ownership model introduces a risk of missing some races because some unprotected access can be hidden by the transition from *Exclusive1* to *Exclusive2* or from *Exclusive2* to *Shared* [38]. However, practical experience shows that this modification brings more positive impacts (due to reducing false alarms) than negative ones. In our implementation, the use of this feature is optional, but in the experiments that we describe, it was successfully applied.

## 3. HEALING ATOMICITY VIOLATIONS

One of the common kinds of bugs causing apparent data races is a wrong atomicity assumption taken by a programmer. In other words, the programmer forgets that there can be a thread switch in a certain place in the code, which may lead to severe consequences. In this section, we discuss how to heal this kind of races. The solution is to reduce, or altogether remove, the likelihood that the scheduler will do a thread switch within the area in which it is better not to have a thread switch. Healing of the more complex inherent races is described in Sec. 4.

In order to be able to heal apparent data races stemming from an atomicity violation, we identify *bug patterns* causing such a violation. In this work, we, in particular, consider three concrete, very frequent bug patterns described bellow. A deeper study of other possible bug patterns is a part of our future work.

### 3.1 Atomicity Violation Bug Patterns

A common bug pattern causing an atomicity violation in Java is a pattern that we call the *load-store bug pattern*. It is an assignment statement that is translated into the byte-code as a sequence of instructions consisting of one or more load instructions on a shared variable followed by one store instruction on the same variable. An elementary example of this pattern is the statement `x++`. The corresponding byte-code is shown in Fig. 2. At first, the current value of the shared variable is loaded into the local memory of the thread by the instruction at line 2, then the local copy is incremented by instructions at lines 5 and 6, and finally, the result from the local memory is stored back to the variable at line 7.

```
2:   getfield  #2
5:   iconst_1
6:   iadd
7:   putfield  #2
```

**Figure 2: Byte-code of the `x++` statement**

A load-store bug pattern is usually caused by forgetting that the source code is translated into the byte-code with a different level of operations granularity.

Another atomicity violation bug pattern we consider is the *test-and-use bug pattern* which can be understood as a special case of the so-called *two-stage access bug pattern* [16]. The test-and-use bug pattern is a conditional statement where the condition is checked at the beginning of the statement and then the result is used inside the statement without making sure that the condition still holds. An example of a code fragment containing the test-and-use bug pattern is in Fig. 3.

The byte-code representation of the conditional statement is in Fig. 4. The value of the shared variable `p` is loaded to the local memory at lines 0 and 1. Then, the condition is tested at line 4. If `p` is not null, then the value of `p` is loaded once again (by the instructions at lines 8 and 9) followed by loading the value of `p.next` (by lines 7 and 12) and storing it back to `p` at line 15.

```
if (p != null) {
    p = p.next;
}
```

**Figure 3: Example of the test-and-use bug pattern**

```
0:   aload_0
1:   getfield  #2
4:   ifnull  18
7:   aload_0
8:   aload_0
9:   getfield  #2
12:  getfield  #3
15:  putfield  #2
18:  ...
```

**Figure 4: Byte-code of the example from Fig. 3**

A similar bug pattern called the *repeated test-and-use bug pattern* arises in loops. An example of this bug pattern is demonstrated by the `while` loop in Fig. 5.

```
while (p != null) {
    p = p.next;
}
```

**Figure 5: A `while` loop showing an example of the repeated test-and-use bug pattern**

At the byte-code level, there is no instruction for `while` loops, which are implemented by instructions for tests and jumps as can be seen in Fig. 6. The byte-code instructions are similar to the byte-code of the conditional statement. The only difference is the `goto` instruction at line 18 which closes the loop.

For each atomicity violation bug pattern, one has to propose an appropriate healing pattern describing how to apply particular healing actions when the given bug pattern is detected. We consider two classes of possible healing actions, namely *influencing the scheduler* and *adding synchronisation.*

### 3.2 Influencing the Scheduler

The first class of healing actions that we consider exploits the nature of concurrency bugs—namely, the fact that they often occur only for a small subset of all possible program execution schedules. In order to exploit this fact, we try to *influence the scheduling* of threads such that a detected concurrency bug will not show up any more (or at least show with a smaller probability). In Java, this can be achieved via *forcing a context switch* by statements like `yield()` or `sleep(0)` or via *temporary changes of the threads priorities.*

In particular, the load-store bug pattern can be healed by forcing a context switch just before the appropriate critical assignment statement. The goal is to execute the statement at the beginning of a new scheduling time slice that is given to the thread that forced the context switch after it is again

```
0:   aload_0
1:   getfield  #2
4:   ifnull  21
7:   aload_0
8:   aload_0
9:   getfield  #2
12:  getfield  #3
15:  putfield  #2
18:  goto    0
21:  ...
```

**Figure 6: Byte-code of the `while` loop from Fig. 5**

scheduled to run. This increases the chance of the critical statement to be finished without an interruption. For example, in order to increase the chances of the `x++` statement considered above to execute atomically, `yield()` can be called just before the instruction `getfield #2` from line 2 in Fig. 2.

Another possibility of how to use a forced context switch for healing the load-store bug pattern is to maintain an indication whether some thread has started but not yet executed a critical assignment statement and if so, force a context switch before any attempt to access the variable from another thread. This should give the processor back to the thread that was interrupted in the middle of a critical assignment and allow it to safely finish the statement. The approach can also be combined with the former one.

Adjusting of the threads priorities can be used to heal the load-store bug patterns by increasing the priority of the thread that is about to execute a critical assignment statement. The priority is then restored to the original value after the critical statement is executed. The increased priority should cause the processor to be granted to the given thread and allow it to finish the critical statement atomically while other threads are forced to wait.

The healing solutions proposed above for the load-store bug pattern can also be used for healing the test-and-use bug pattern without any modification. However, when dealing with the repeated test-and-use bug pattern, we should be more careful. When using the healing solution based on forcing a context switch, e.g. via `yield()`, we must ensure that the context switch is forced after each iteration of the loop. Further, increasing of the thread priority for the whole execution of the loop is not generally applicable due to it can cause serious problems like starvation. The thread priority can thus be increased only for one iteration of the loop, then it must be decreased (possibly with a forced context switch to allow other threads to run) and increased again for next iteration of the loop.

Let us stress that the healing actions we just described do not guarantee that a bug is always healed but they decrease the probability that the bug will actually show up. These healing actions can also introduce a significant overhead in some cases, for instance, when healing the repeated test-and-use bug pattern. On the other hand, these healing methods are safe because if they are applied carefully (e.g. we have to be prepared for handling the `InterruptedException` that the `sleep()` method in Java may throw), they do not introduce any new bug. Thus, we can apply them without any healing assurance.

### 3.3 Adding Synchronisation Actions

We now examine another approach to healing atomicity violation bugs that is based on *adding new explicit synchronisation actions.* In particular, the synchronisation actions we add are based on a suitable use of mutexes (locks).

Occurrences of the load-store as well as test-and-use bug patterns can be healed by introducing a new lock `l` that protects the shared variable `v` which is a subject of the detected data race. It is necessary to guarantee that the lock `l` will be held by every thread when it accesses `v`. In addition, the lock `l` must be held during the entire execution of the critical statement forming the detected bug. This means to put `l.lock()` before this statement and `l.unlock()` after it (or after the last access to `v` within the statement). Introducing a new lock in the described way leads to a considerable overhead. Sometimes, we are able to guess—using information given to us by the data race detector used—which out of the locks already in use (let us mark it by `lv`) is used in the program for guarding the shared variable `v`. In such a case, it is not necessary to introduce a new lock `l`—instead, we can heal the bug by introducing `lv.lock()` and `lv.unlock()` to the same places as above. This solution does not influence the application efficiency so much, however, unlike the former solution, it is not guaranteed that it always succeeds.

Healing of the repeated test-and-use bug pattern using additional locks faces the same problems as healing of this pattern by influencing the scheduling. We introduce a new lock `l` or guess an already used lock `lv` as within healing the test-and-use bug pattern above. We call `l.lock()` or `lv.lock()` at the beginning of each loop (e.g. just before executing the instruction at line 1 in Fig. 6) and `l.unlock()` or `lv.unlock()` after the last access to the shared variable `v` in the loop body (e.g. just after the execution of the instruction at line 15 of the same byte-code fragment).

The described healing by introducing synchronisation on a new lock is able to completely remove the detected bug, but—on the other hand—it can introduce new (and even more dangerous) bugs. In particular, we can cause a deadlock this way. Thus, an application of such a healing action requires healing assurance that is discussed in Sec. 5 (or a use of locks with a time-out).

**Remark.** Performing automatic healing actions does not make sense *during the software development phase.* Instead, a bug should be reported to a developer together with a suggestion how to correct it. This can significantly reduce the time spent on program debugging and thus reduce the development costs.

## 4. HEALING INHERENT RACES

In Sec. 3, we discussed healing of races that occur due to a violation of an atomicity assumption. However, there are other causes for data races that do not relate to atomicity violation, and hence different healing methods should be considered for them. Consider, for example, the simple program in Fig. 7 where two threads access a shared variable `done`. `Thread1` sets it to `false`, and `Thread2` sets it to `true`. The execution starts with `Thread1`. There is an implicit assumption that the execution of `Thread2` ends after the execution of `Thread1`, hence there is no explicit mechanism to guarantee the intended order of events, i.e. that when the program ends, the value of `done` is `true`.

```
    Thread1                  Thread2

1) done=false   1) for (int i=1;i<100;i++) {
                2)     print(i);
                3) }
                4) done=true
```

**Figure 7: An inherent race**

In this example, adding a lock that should be obtained before the accesses to `done` will not heal the race. The reason is that executing atomically any segment of code in either thread does not guarantee a certain order of accesses to `done`. Namely, the race does not occur due to atomicity violation but rather to a missing logic that would guarantee a certain order of events. Note also that there is a notion of an intended order of accesses here—first, `Thread1`, and then `Thread2`, i.e. the possible orders of accesses to the shared variable can be classified as "good" and "bad".

We define a data race as an *inherent race*[2] if the following conditions hold:

- Executing any segment of code in each thread atomically does not determine an order of accesses to the shared variable.
- The different orders in which the shared variable is accessed can be classified as "good" and "bad" according to the expected behaviour of the program.

It stems from the definition of an inherent data race that in order to detect such a race, one should have a notion of what are "good" and "bad" (buggy) runs of the program. This can be done by a user indication provided through an oracle, which receives different details about the run, such as its output and run-time, and labels the run accordingly. It also stems from the definition that races that involve an atomicity violation are not inherent races. However, even if the race is due to an atomicity violation, it may be difficult to classify it correctly. In our approach, we let such cases to be treated as though they were inherent.

The healing approach for inherent races is more complicated than for races involving an atomicity violation. For the latter case (described in Sec. 3), the healing attempts to cause the atomic execution of a code segment. For inherent races, it is first required to learn what are the "good" and "bad" orders of execution of the statements that access the shared variable. Only once such information is collected, healing can take place. Thus, the healing of inherent races requires a preliminary stage in which the program is run multiple times, in each run the order of execution with respect to the race is recorded and classified as either "good" or "bad" according to the oracle that observes the outcome of the program.

[2]Note that inherent races could even be defined independently of the notion of data races that we gave in Sec. 2 since they can appear even if all accesses to the involved shared variable are guarded by locks. We do not consider such a situation here as we suppose a data race detector to be used for detecting bugs, which does not allow us to detect the more general kind of inherent races. However, if one uses another way of detecting bugs, the healing techniques we describe could be applied even in the more general case.

Once this data is collected, we would like to enforce "good" orders in the following runs of the program, or at least prevent "bad" orders. We suggest two different approaches. The first approach is to change the scheduling of the program (in a similar way as within the self-healing of atomicity violation described in Sec. 3.2). This can be done by adding `wait()` and `notify()` statements before the accesses to the shared variable. The `wait()` statement should have a time-out in order to avoid introducing deadlocks into the program due to the lost notify bug pattern. Another option is to change the timing of the program: we keep track of the order of execution with respect to the race. If the next access to the variable will result in a "bad" order, we try to prevent the next access, for example by changing threads priorities, or inserting calls to `sleep()` or `yield()`. Similarly, if the next access to the variable will result in a "good" order, we try to prevent a context switch, for example by changing threads priorities. The advantage of changing timing is that it is safe as was already discussed in Sec. 3.2. The drawback is that it does not guarantee the healing of the race, but rather only increases the probability for a "good" order.

The second approach for healing inherent races concentrates on multiple write accesses to a shared variable. This approach does not prevent "bad" orders from occurring, but rather once a run is identified as "bad", tries to override the race. For example, in the program in Fig. 7, assume we learned that the "good" order is first setting `done` to `false` and then to `true`, and the other order is "bad". Now assume that `done` is first set to `true` by `Thread2`. This will classify the run as `bad`. Once `done` is set to `false` by `Thread1`, the healing will override its value with the value written by `Thread2`. Note that this healing approach will not result in a "good" order, but rather is an attempt to revert a "bad" order into a "good" one. This approach is of course not safe. For example, if the value of the shared variable is read by some thread before it is overridden, this can result in an inconsistent state. To check that such a situation cannot happen (or at least cannot easily happen), one should use some of the healing assurance methods mentioned in Sec. 5.

## 5. HEALING ASSURANCE

When we apply advanced healing actions (like adding a new lock within healing an atomicity bug or when overriding values within inherent race healing), we can potentially introduce new and even more dangerous bugs (e.g. a deadlock). Therefore, as a part of our future work, to ensure that a new bug cannot be introduced by a self-healing actions, we intend to explore the use of suitable *formal verification* techniques (like *model checking* or *static analysis*). Moreover, formal verification can also be used for ensuring that a selected healing action really fixes the detected bug. Otherwise, a different healing action should be taken into account or no self-healing action should be performed at all (due to an application of a healing action usually entails some performance drop).

We should take into account that model checking of a complete state space is not possible when dealing with large real-life software systems. However, we can apply it at least in a limited fashion. For example, on the modified software system that we try to heal, we may run *bounded model checking* in a limited neighbourhood of the state in which a problem was located—we first backtrack a few steps and then do a systematic or heuristic state space search—and check how the system behaves. The number of forward/backward steps in the state space search may be adjusted according to the amount of available computing resources.

In many cases, even a simple *static analysis* (possibly run on the byte-code level of the Java programs that we consider) may help a lot too. For instance, when we introduce a new lock and unlock calls on some mutex in order to get rid of some data race, we might subsequently check whether in the scope of the introduced lock/unlock pair there may be (perhaps indirectly) required some further locks. If this is not the case, there is no danger of the healing introducing a deadlock. In practice, the race conditions may often occur in very simple statements (such as `x++`) where static checking that no locks will be acquired is easy.

## 6. PRELIMINARY RESULTS AND EXPERIMENTS

Here, we briefly describe prototype tools that we have developed and summarise our first results that we have already achieved in the data race self-healing.

### 6.1 ConTest

ConTest is an automated tool for testing concurrent software. It is designed to discover bugs in concurrency by a repeated execution of an instrumented code. The objective of ConTest is to make multithreaded bugs materialise while having a minimal impact on the user and software performance. ConTest provides us with three essential services: instrumentation, heuristic noise injection, and a listeners architecture.

The ConTest *instrumentation* works on the Java byte-code level. An instrumentator adds calls of ConTest's methods at different locations. ConTest is called during an execution of the instrumented software, e.g. when a shared variable is accessed, a thread is started or stopped, or some synchronisation events occur, etc.

*Noise injection* is a technique that forces different legal interleavings for each execution of a test in order to check that the test continues to perform correctly. In a sense, it simulates the behaviour of other possible schedulers. When a call of ConTest from the execution of instrumented software is received, the noise heuristic decides, randomly or based on a specific bug-finding technique, if some kind of delay is needed. This technique increases the probability of finding a bug.

Finally, ConTest contains a *unified listeners architecture* that provides an easy to use interface to the execution trace of the instrumented software. Tools of third parties can register Java listeners to specific actions. If a listener (or a set of listeners) is registered to the event, the code of the listener (or listeners) is executed when the event arises. ConTest provides two events to most of the actions—the so-called *before* and *after events*. The before action events allow listeners to be executed before an action is performed. The after action events allow listeners to be executed after the action is performed. This allows tools using the ConTest listeners architecture to have enough control over the instrumented software. Variable access, thread events, monitor events, and some synchronisation primitives are supported by the ConTest listeners architecture in the current version.

The ConTest tool can be used by classical testing tools with a higher probability of revealing a concurrent bug. Another possibility is to develop specialised tools for detecting particular classes of concurrent bugs. As an example of this approach, we have built a data race detector on top of ConTest as described in the following.

### 6.2 Data Race Detection

Java contains more synchronisation primitives than only (explicit) locks [8], including, for instance, implicit locks, wait-notify, and join synchronisation. Java implicit locks are mutexes implementing monitor-like critical sections quoted by the `synchronised` statement. Out of these synchronisation mechanisms, we currently concentrate on implicit locks and we also have some support for the join synchronisation. Moreover, the restriction of our race detector to implicit locks is only due to the ConTest listener architecture, on top of which our date race detector is built, whose current version does not provide listeners for Java explicit locks (defined in the `java.util.concurrent` package)—once this support is added, our race detector is ready to deal with explicit locks too.

Using the ConTest listeners architecture, our race detector detects thread activation and termination events via the `ThreadBegin` and `ThreadEnd` listeners. A use of Java implicit locks is detected by the `MonitorEnter` and `MonitorExit` listeners which are called when a thread tries to lock some object (i.e. to access its monitor). The access to variables is monitored by the `BeforeVarRead` and `BeforeVarWrite` listeners and join synchronisation is detected by the `AfterJoin` listener. The listeners provide us with useful information concerning the events—e.g. the identification of the object over which an event is performed and the relevant line in the Java source code.

Our race detector implementation is based on the Eraser algorithm [35] extended with the ownership model [18, 38, 11] as described in Sec. 2. We have modified this version of Eraser to add some support for Java join synchronisation to further reduce the false alarms ratio.

The simple extension we did to Eraser to add some support of the *Java join synchronisation mechanism* was motivated by a relatively high rate of false alarms that we were receiving in some Java programs building on this standard Java synchronisation. In Java, if a thread $t_1$ calls the `join()` method of another thread $t_2$, it ensures that all the events of the thread $t_2$ are executed before the events following the `join()` call in the thread $t_1$. Based on this, we can make the following assumption: *there is no race possible between successfully join-synchronised threads after a successful join.* This assumption based on the *happened-before* relation has been reflected in our data race detector as follows.

Each thread $t$ maintains a set of threads $S(t)$. A terminated thread $t_1$ is added into the set $S(t)$ after a successful join synchronisation with the thread $t$. Each variable $v$ maintains a set of threads $T(v)$. A thread $t$ is added to the set $T(v)$ when $t$ accesses $v$. If a thread $t$ is accessing a variable $v$ and $S(t) \cup \{t\} \supseteq T(v)$, we know that the thread $t$ is the last currently existing thread accessing $v$ and all others have been successfully join synchronised with $t$. Then, the variable $v$ changes its status back to `Exclusive2`, its $C(v)$ is set to contain all possible locks, and $T(v)$ contains only the current thread $t$. This modification helps us to rapidly decrease the number of false alarms produced in environments based on loops or in situations where the last thread finalises shared global variables.

In our race detector, we have also implemented a simple heuristics that *suggests which lock should have been used* by the thread that caused a data race. This information can subsequently be used in self-healing as described in Sec. 3.3 or, during the software development phase, be provided to the developer as a hint of what lock was probably omitted. The heuristics is inspired by an assumption that if some threads have already used a lock for accessing a shared variable $v$, then the same lock should be used by all other threads. The set of locks to be used in self-healing (or to be suggested to a developer) is the set $C(v)$ of candidate locks just before it becomes empty (by an access to $v$ without holding a proper lock).

Of course, the above approach does not guarantee that the suggestion is always correct. For instance, when the first thread accessing a shared variable uses a wrong lock, other threads are suggested to use the same wrong lock too. The number of such wrong suggestions can be reduced by postponing the suggestion after the run when we have more information. For that reason, we maintain an additional set $C(v,t)$ of candidate locks for each shared variable $v \in V$ and each thread $t \in T$ where $V$ is the set of all shared variables and $T$ is the set of all threads. After the run, we calculate the intersection $C'(v)$ of $C(v,t)$ of all threads excluding the threads $T_{buggy}$ causing a race:

$$C'(v) = \bigcap_{t \in T \setminus \{T_{buggy}\}} C(v,t)$$

In most cases, the set $C'(v)$ contains a lock or locks that should be used when accessing the variable $v$.

We have implemented our race detector also as an *Eclipse plug-in* [7]. So developers can test their applications using our race detector straight from their integrated development environment.

## 6.3 Data Race Healing

A healing action can have the form of a source code modification followed by recompiling and restarting the application. However, this is quite invasive and expensive approach which can be applied for some types of applications only. Currently, we exploit a more gentle approach that consists in *performing healing actions on-the-fly* through the ConTest listener architecture.

We enriched our data race detector to not only use the `BeforeVarRead` and `BeforeVarWrite` listeners, but also the `AfterVarRead` and `AfterVarWrite` ones. This allows us to work with a quite a fine granularity over the byte-code. Our implementation is able to start a selected healing technique just before the first access of a critical variable and end it just after the last access within the block that should be atomic.

So far, we implemented a self-healing approach for simple *load-store bug patterns* like `x++` only. The race detector is monitoring the access to variables with respect to their positions in the Java source code. For each variable $v$, a set $A(v)$ of lines in the source code is maintained. Whenever there are at least two consequent accesses to the variable $v$ at the same line of the source code (the information about the line is given to us by ConTest) and the first access is a read and the last is a write, the line number is added into $A(v)$. The lines in $A(v)$ are used as those about which we assume that they should be executed atomically when a race on $v$ happens. As an alternative to building $A(v)$ at the run-time, it can also be obtained in advance using static analysis or simply by saving the set from the previous runs of the race detector.

When the race detector identifies a race over a variable $v$, the variable gets into the Eraser's *Race* state. From then on, if we detect via the listeners that such a variable $v$ is about to be accessed on a line which is in $A(v)$, we assume that a *critical section*, which should be run atomically, is about to be performed, and one of the healing techniques described in Sec. 3.2 and 3.3 can be applied. Using the listener architecture, we then inject the appropriate healing actions before the critical section and (unless we use the simple `yield()/sleep()`-based healing) also after it (i.e. behind the last write access to the critical variable in the section). Moreover, in the case of healing by adding a new lock, it is not enough to only cover accesses to the critical variable $v$ from lines in $A(v)$. We have to apply locking for all the other accesses too, the only difference being that in their case we assume the critical section to span only the appropriate single access to the variable. Also, in the case when we force a context switch in threads that are not the ones that entered, but not yet left the critical section, a healing action is needed at all accesses to the critical variable.

As was mentioned in Sec. 3, we can choose between two classes of healing techniques—influencing the scheduler or adding synchronisation. We have implemented several techniques from both classes. In particular, we implemented *forced context switches* via calling `yield()` method both in all threads accessing a critical variable $v$ as well as only in those who are about to start using $v$ while somebody else is already using $v$. We also implemented the healing by *changing priorities* and by *adding new synchronisation locks*.[3] As an extension of healing by changing priorities, we additionally implemented a technique intended for multi-processor computers where lower priority threads could still run on other processors. In order to tackle this problem, we have a pre-prepared group of *high-priority, dummy, but load producing processes* that are normally blocked. Once a healing starts, they are unblocked for a while to block other processors than the one on which a critical thread is running.

## 6.4 Experiments

We have evaluated our data race detector on several examples including those from real software—e.g. a web crawler algorithm with 19 classes and 1200 lines of code, embedded in an IBM product. Our data race detector proved to be able to detect even rarely exhibited data races. Of course, the race detector still produces some false alarms. This is typically the case when some peculiar synchronisation mechanisms and/or various optimisations are used (e.g. some true races are left in the code because they are harmless).

So far, we have made self-healing experiments for the *load-store* bug pattern only. We have used a simple bank account program simulating bank accounts where multiple account threads perform simple changes to the particular accounts and the total balance of the bank (using simple arithmetic operations like `BankTotal += sum;`) without a proper

[3]We have not implemented the re-use of already existing locks suggested by the data race detector for healing. The reason is that ConTest currently does not listen on events on explicit locks and so it can suggests us with implicit locks only, but those cannot be fully manipulated explicitly.

synchronisation over the global balance variable `BankTotal`. When no such operation is interrupted, the final balance corresponds to the performed operations. However, if some operation with a bank account is interrupted, the final balance may get wrong.

We made all the tests on 1-, 2-, and 4-processor computers for 600 executions of the problematic line and for 2, 3, 5, 10, and 15 working threads. We have also simulated different frequencies of executing the problematic piece of code by adding some safe code to the account thread loop. The percentage of the execution time spent in the dangerous section ranged from 5% to $1,8*10^{-3}$%. Each of the tests was done 100 times, and the number of cases where the balance was wrong was measured.

The best results from the point of view of removing the race were—not surprisingly—achieved when using a new explicit lock as in its case, a success is guaranteed. Results achieved by the techniques influencing the scheduler were not so positive. Some of them were able to significantly decrease the probability of the race to reveal when they were added directly to the code. However, when used on-the-fly in conjunction with the race detector, they did not provide very positive results. This is probably because the race detector is called before and after each access to the healed variable which significantly extends the critical section and therefore increases the probability of a context switch and a race manifestation. Despite this and the limited number of tests we performed as yet, we witnessed a scenario where this approach was also quite successful as described below. In the future, we need to tune our implementation in order not to interfere with the healing effects we want to achieve.

Figure 8 shows the situation with influencing the scheduler on a 2-processor machine (2x Intel Pentium Xeon 1.7 GHz, 1 GB RAM) with $1,8*10^{-3}$% of the execution time spent in the dangerous section described above. Four sets of results are presented concerning (1) the original code without the race detector, (2) the situation with the race detector enabled, but without healing, (3) healing by increasing the priority of a critical thread, and (4) healing by a yield in threads that see that somebody is in the critical section. Healing by a new explicit lock is not presented because it heals the race in all cases. As we can see, in the original code, the race manifestation dramatically changes between 2 and 5 threads in conflict, but still does not get over 50%. A use of the race detector increases the manifestation to nearly 100% and healing then decreases this influence again. Due to the reason described in the previous paragraph, we do not get below the original fault rate when using healing based on influencing the scheduling. An exception is the case with 2 working threads and the use of `yield()` on threads that notice that some other thread is in the critical section. The decrease when compared with the use of a data race detector without healing is, however, significant and gives us an indication that even the simple healing techniques based on influencing scheduling can achieve much better results when we optimise our implementation. Moreover, measurements on more different programs are still needed.

Our tests also approved that the probability of a race manifestation rapidly increases with the number of processors and the number of threads in conflict. If these numbers are high, healing by influencing the scheduler does not help too much. The situation becomes better if the problematic code is very rarely executed (which is, however, the case of many

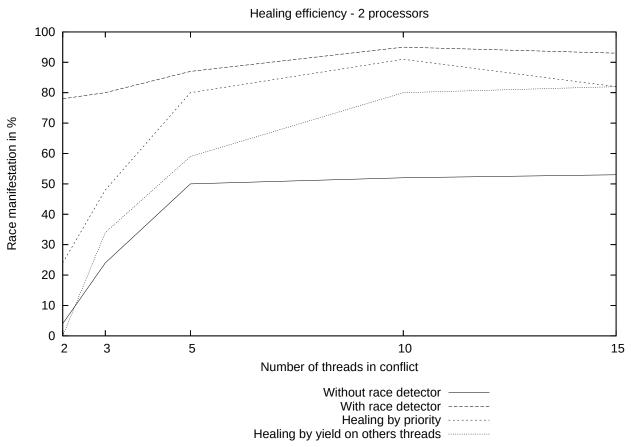

**Figure 8: Race healing efficiency**

bugs that really escape from the development phase to the field). Then there are usually only two threads in conflict and influencing the scheduler helps.

The slowdown of the application run in conjunction with our techniques depends on the number of instrumented points in the code and on the additional synchronisation overhead introduced by the race detector caused by gathering information about locks, threads, and variables. The slowdown we witnessed on our simple case study ranged from almost negligible values up to about a 3.75-times slower run (for 4 processors and 15 working threads).[4] To optimise these values, the number of instrumented points can be reduced by a suitable static analysis done in advance. The synchronisation overhead can then decreased by a further optimisation of our algorithms.

## 7. RELATED WORK

Research on methods of self-healing of various aspects of software (such as concurrency, functional aspects, performance, etc.) is currently getting momentum. As for what concerns self-healing (or fault tolerance) of problems in concurrency, there appeared multiple works in the area of distributed systems (such as [36, 26]). However, the closest work to what we present here is probably the approach applied by the *ToleRace* tool [27]. ToleRace concentrates on the so-called asymmetric races (a data race between a region of code accessing some shared variables after certain locks were taken and a region of code where these variables are accessed without the locks). The approach is based on transforming the critical regions of code such that they manipulate local copies of the shared variables and at the end of the region check whether a race on a shared variable happened. If this was the case and it was a read-write race, ToleRace can "tolerate" it by producing the correct result based on the local copies of shared variables. If a write-write race occurs, ToleRace cannot heal it and just announces the problem. ToleRace has a low overhead and can in safe way significantly decrease the appearance of read-write races. On the other hand, it is less general than our approach both in

[4]Note that we were not yet using any bounded model checking nor static analysis to ensure safety of healing by adding new synchronisation locks.

terms of the races it can detect as well as in terms of what it can heal—as we have already said, ToleRace does not heal write-write races and it does also not solve races on variables referring to external resources (such as files) whose local copy cannot be created.

## 8. CONCLUSION

We have shown in this work how we apply self-healing in the context of fixing data races in concurrent Java programs. We have discussed multiple common bug patterns leading to data races and proposed possible self-healing actions to be taken when such a bug pattern is detected. We have implemented some of our ideas and obtained a preliminary experimental evidence about their behaviour showing that they indeed have a high potential to be useful in practice despite there is still a large space for improving our implementation.

Concurrency self-healing (and self-healing in general) can be used within development as well as within production phase of the software life cycle. It seems that it is not suitable to apply the whole self-healing cycle for mission critical applications (like avionics, aviation, nuclear plants, military, etc.) due to huge impacts of a possible application failure. Developers as well as customers would risk a failure caused by not healing the system rather than to risk a failure caused by the self-healing. However, even for critical applications, concurrency self-healing can be very useful—it may help in finding concurrency bugs and also to suggest their possible solution. Moreover, even for critical applications, the self-healing approach can be applied also within the production phase. When some hard to reveal problem occurs at the customer place (it can be thousands kilometers far away from the development team), the application should be with some carefulness remotely switched to the self-healing monitoring and debugging mode (without taking the healing actions) in order to report the bug to the developers.

Despite we have already achieved first results in self-healing, we are still at the beginning. Within the problem detection, we intend to make the tool ConTest working also on C/C++ code in order to make it available for a wide range of applications. Oracles for other common concurrency problems can be build and localisation methods can be improved. The healing techniques that we have not yet implemented are to be implemented and tested. Efficient implementation strategies for the healing techniques that we have proposed are to be sought to (1) increase their healing efficiency and (2) to decrease the slowdown they impose when applied. We will also extend the set of possible healing actions. Last in the list but definitely not last in the importance is a practical application of formal methods (model checking and/or static analysis) within self-healing. It includes their application for problem localisation in order to reduce the number of false alarms, and for healing assurance to prove that a bug was covered by a healing action and a new one has not been introduced.

## 9. ACKNOWLEDGEMENTS.

This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD — project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein. This work is also supported by the Czech Ministry of Education under the project *Security-Oriented Research in Information Technology*, contract CEZ MSM 0021630528, and by the Czech Science Foundation under the contracts 102/07/0322 and 102/06/P076.

## 10. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proc. of ISCA'91*, 1991. ACM Press.

[2] C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *Proc. of VVEIS'03*, Angers, France, 2003.

[3] C. Artho, K. Havelund, and A. Biere. Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors. In *Proc. of ATVA'04*, LNCS 3299, 2004. Springer.

[4] V. Balasundaram and K. Kennedy. Compile-time Detection of Race Conditions in a Parallel Program. In *Proc. of ICS'89*, 1989. ACM Press.

[5] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proc. of CAV'01*, LNCS 2102. Springer, 2001.

[6] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A Theory of Data Race Detection. In *Proc. of PADTAD'06*, 2006. ACM Press.

[7] W. Beaton and J. d. Rivieres. Eclipse Platform Technical Overview. Technical report, The Eclipse Foundation, 2006.

[8] T. P. e. a. Brian Goetz. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[9] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2–3):129–166, 2004.

[10] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting Data Races in CILK Programs that Use Locks. In *Proc. of SPAA'98*, 1998. ACM Press.

[11] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Data Race Detection for Multithreaded Object-Oriented Programs, 2002.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[13] A. Dinning and E. Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proc. of PPOPP'90*, 1990. ACM Press.

[14] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *Proc. of PADD'91*, 1991. ACM Press.

[15] P. A. Emrath and D. A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proc. of PADD'88*, 1988. ACM Press.

[16] E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How To Test Them. In *Proc. of IPDPS'03*, 2003. IEEE Computer Society.

[17] C. Flanagan and S. N. Freund. Detecting Race Conditions in Large Programs. In *Proc. of PASTE'01*, 2001. ACM Press.

[18] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of POPL'04*, 2004. ACM Press.

[19] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification*, 2006.

[20] C. Flanagan and S. Qadeer. Types for Atomicity. In *Proc. of TLDI'03*, 2003. ACM Press.

[21] S. N. Freund and S. Qadeer. Exploiting Purity for Atomicity. *IEEE Transaction on Software Engineering*, 31(4):275–291, 2005.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proc. of 10th SPIN Workshop*, LNCS 2648, 2003. Springer.

[23] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward Integration of Data Race Detection in DSM Systems. *Journal of Parallel and Distributed Computing*, 59(2):180–203, 1999.

[24] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.

[25] J. Mellor-Crummey. Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs. In *Proc. of PADD'93*, 1993. ACM Press.

[26] N. Mittal and V.K. Garg. Finding Missing Synchronization in a Distributed Computation Using Controlled Re-Execution. In *Distributed Computation*, 2004.

[27] R. Nagpaly, K. Pattabiramanz, D. Kirovski, and B. Zorn. ToleRace: Tolerating and Detecting Races. In *P*roc. of STMCS'07, 2007.

[28] R. Netzer and B. Miller. Detecting Data Races in Parallel Program Executions. In *Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, 1990. MIT Press.

[29] R. H. B. Netzer and B. P. Miller. Improving the Accuracy of Data Race Detection. *Proc. of PPOPP'91*, published in ACM SIGPLAN NOTICES, 26(7):133–144, 1991.

[30] R. H. B. Netzer and B. P. Miller. What Are Race Conditions?: Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[31] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to On-the-Fly Race Detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10), 2000.

[32] E. Pozniansky and A. Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*, 2003. ACM Press.

[33] B. Richards and J. R. Larus. Protocol-based Data Race Detection. In *Proc. SIGMETRICS Symposium on Parallel and Sistributed Tools*, 1998. ACM Press.

[34] M. Ronsse and K. D. Bosschere. Recplay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[36] A. Tarafdar and V.K. Garg VK. Software Fault Tolerance of Concurrent Programs Using Controlled Re-Execution. In *Proc. of DISC'99*, 1999.

[37] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where it Hurts—An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*, to appear. ACM Press, 2007.

[38] C. von Praun and T. R. Gross. Object Race Detection. In *Proc. of OOPSLA'01*, 2001. ACM Press.

[39] L. Wang and S. D. Stoller. Run-time Analysis for Atomicity. In *Proc. of RV'03*, ENTCS 89(2), 2003. Elsevier.

[40] L. Wang and S. D. Stoller. Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In *PPoPP'05*, 2005. ACM Press.