Bachelor Project

Static Race Detection and the PWR Algorithm

Lisa Hofert

Examiners: Prof. Dr. Martin Sulzmann,
Prof. Dr. Peter Thiemann

Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Programming Languages

July 4, 2022

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Contents

1 Introduction

This project aims to further explore the PWR Relation, which was detailed by Martin Sulzmann & Kai Stadtmüller in "Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction" Sulzmann and Stadtmüller [2020].
The PWR Relation is used in an algorithm for efficient data race prediction on a trace of a singular program run; the relation itself used in this work is complete and unsound.
Instead of exploring more efficient data race prediction itself, we want to explore the relation and create useful tools for working with it in Haskell, building in part on existing implementations by Martin Sulzmann which can be found on https://github.com/sulzmann/source.
The first requirement for being able to work with PWR in Haskell was an already provided DSL for representing traces. Building on that, PWR itself had to be implemented. The implementation was later modified to be more general and work with different data represenatations.
Building on this, a tool for printing traces and annotating said traces could be implemented. By printing traces and computed relations in Markdown, the results can be made more easily readable, and the output can be filtered. Furthermore, there's the option of automatically generating LaTeX code snippets to reduce tedious busywork. In this case, the relations between events in a trace are shown using arrows.
The last part of the project is about reordering traces. By creating more valid reorderings, we can improve testing capabilities. We want to use PWR in the functions - since it is complete, it should be maintained. Since this is a more independent part of the project, and time complexity can easily explode, the work regarding this is largely experimentation.
Apart from calculating every possible valid reordering, it can be more useful to only generate ones that can be considered useful or interesting. Some comparisons to other existing approaches is included at the end of this report as well.

2 Background

2.1 Tracing and Data Races

A trace is a log of the events occurring in a single execution of a (here concurrent) program. Events can be in different threads, and a trace obtained from different executions is most probably also different each time, since concurrent programs are non-deterministic.

Traces, for our purposes, are lists of events, where only read/write events on shared variables and operations on locks are of importance. Additionally, we include fork/join to for creating and removing threads. Recording these traces was not in the scope of the project, so it is assumed that traces are already provided.

The following is an example of a representation of a possible trace we could use:

	$1\sharp$	$2\sharp$
1.	$fork(2)$	
2.	$acq(x)$	
3.	$w(y)$	
4.	$rel(x)$	
5.		$r(y)$
6.		$acq(x)$
7.		$r(y)$
8.		$rel(x)$

Every row and therefore every event is associated with a unique trace position number. Every column represents a different thread.

In this specific example, the first event creates a second thread from the already existing main thread. The second event is an aquire event on the lock variable x, followed by a write on the shared variable y and a release of the lock variable from before. Afterwards, the events in the second thread were logged, which is a read on the previously written variable x that is not protected by a lock. It is followed by another lock-read-release sequence.

2.1.1 Data Races

Two events are considered to be conflicting if they're read/write events on the same shared variable, the two events are in different threads, and at least one of the events is a write. If conflicting events appear in direct succession, a data race occurs. Considering the trace above, there doesn't seem to be a data race at first glance; however, we are looking at just one valid execution of a non-deterministic program. There is nothing preventing the fifth event, the read on x, from occuring right after third event, a write on x. This would be a data race.
Finding out which events could be part of a data race, based only on a single trace and in an efficient manner, is the goal of algorithms like PWR.

(TODO: explain what qualifies as correctly reordered trace)

2.2 The PWR Algorithm

probably around 1-2 pages?

The PWR Algorithm is near complete, often sound data race prediction method and works with traces like the ones explained above.
It builds on previous methods such as happens-before and lockset. In summary: hb is weaker. combo methods. lockset **(TODO: explanation of terms - including vector clocks, epoch)**
(TODO: PO/WRD/ROD)
(TODO: general idea of the algorithm)
(TODO: short comparison to other algorithms/improvements)

3 Approach

The code discussed here can be found on `https://github.com/lh535/datarace-prediction`

3.1 Implementing Traces (Trace.hs)

The implementation of traces was largely based on existing code by Martin Sulzmann, which may also be found at `https://github.com/sulzmann/source/blob/main/Trace.hs`.
Traces are represented as a list of Events `[Event]`. There are six types of events used: Reading a Variable, Writing a Variable, Acquiring a Lock, Releasing a Lock, Forking a Thread, and Joining a Thread.
These Events can be be created using the following functions:

- `rdE t x` (read event with t=current thread, x=variable)
- `wrE t x` (write event with t=current thread, x=variable)
- `acqE t x` (acquire event with t=current thread, x=lock variable)
- `relE t x` (release event with t=current thread, x=lock variable)
- `forkE t1 t2` (fork event with t1=current thread, t2=created thread)
- `joinE t1 t2` (join event with t1=current thread, t2=joined thread)

We start counting threads at 0, where thread 0 is the main thread. Every Trace must contain a main thread. There are functions for creating threads:
`mainThread` has no arguments and returns the main thread. `nextThread t` takes a thread `t` and returns a thread with an incremented index number.

Each Event also should have a location number, however the functions above do not automatically add this location number. To add correct location numbers to a trace, `addLoc t` can be used, which takes a trace `t` and returns a trace.
Many of the printing functions apply `addLoc` automatically for convenience.

	1♯	2♯
1.	$fork(2)$	
2.		$w(x)$
3.	$w(x)$	
4.		$r(x)$

Figure 1: Example 1

Lastly, let's look at the following small trace: To represent it in a Haskell, this trace can be created using the following code:

```
ex1 =
  let t0 = mainThread
      t1 = nextThread t0
      x  = Var "x"
  in [ forkE t0 t1,
                        wrE t1 x,       -- w1
       wrE t0 x,                        -- w2
                        rdE t1 x        -- r3
     ]
```

3.2 The PWR Algorithm

3.2.1 First Approach (PWR_old.hs)

The first idea for implementing PWR was to stick close to the paper where it was defined (Sulzmann and Stadtmüller [2020]). This meant maintaining vector clocks for every thread, and incrementing and syncing in accordance with the given pseudo code. Some adjustments had to be made, since we are only computing the PWR relation, not trying to output potential races themselves.

We use a state monad to keep track of five global variables (reduced from the twelve needed for the PWR algorithm in the paper).

These variables are the lockset per thread, the vector clock per thread, the vector clock of the last write on each variable, the epoch of the last aquire for each lock variable, and a history of each aquire/release time for each lock variable. Because the variables need to be modified often, the Map type is used to represent them in Haskell. The function *pwr* then takes a trace and processes its events sequentially. After changing the global variables accordingly, each event is saved in the result type (a Map from Events to their current Vector Clock)

(TODO: explain why post-clocks were used) (TODO: explain why specific global

variables are needed, some important functions)

3.2.2 Current Implementation (PWR.hs)

This is the current used implementation of PWR, which mainly builds on the previous version and generalizes it.
Using vector clocks to compute the relation is compact, however it's not easily readable at a glance. Therefore, it could be useful to use sets instead; a notable observation was that the operations on vector clocks and sets are fairly similar. To take advantage of this, a new class `PWRType` is defined; it specifies the operations that need to be supported by a type to be able to compute the PWR relation.
The class for the code is the following:

```
class PWRType a where
startState :: a
inc :: Event -> a -> a
union :: a -> a -> a
before :: a -> a -> Bool
extend :: Thread -> Map Thread a -> Map Thread a
```

Two instances of `PWRType` are predefined: `VClock`, which is a vector clock representation, and `Set Event`. For Sets, the current event itself is also in the set of events that are in relation to it. `pwr t`, as before, takes a trace `t` as the only argument, but now computes a result `r` of type `R a`, which is a map from each event to the computed PWR relation. `eventMap r` is the unpacking function to get the map from the return type. `pwrClock t` and `pwrSet t` can be used if either vector clocks or sets should explicitly be used.

There are also specialized functions for pwr printing: - `annotatedWithPWR t` prints a markdown table of the trace `t` and the pwr vector clocks - `annotatedWithPWRSet t` prints a markdown table of the trace `t` and the pwr sets - `interactivePWR t` is the PWR version of `interactiveSet` and can print markdown for only some events. See section about markdown printing for more details. - `interactiveLatexPWR t` can print latex code of the trace `t`, with the option to add arrows for events that are in the pwr relation. See section about latex printing for more details.

As an example, we'll look at the trace from figure 1 again and show the code for computing and printing the different PWR represenations. `ex1` refers to the Haskell representation of the example.

```
> annotatedWithPWR ex1
       T0        T1        Vector Clocks
1.  fork(t1)            t0: 2, t1: 0
2.            wr(x)     t0: 2, t1: 1
3.  wr(x)               t0: 3, t1: 0
4.            rd(x)     t0: 3, t1: 2

> annotatedWithPWRSet
   T0        T1        PWR Set
1.  fork(t1)            fork(t1)_1
2.            wr(x)     fork(t1)_1, wr(x)_2
3.  wr(x)               fork(t1)_1, wr(x)_3
4.            rd(x)     fork(t1)_1, wr(x)_2, wr(x)_3, rd(x)_4
```

(TODO: printing is introduced later, but is useful for an example now. Reorder some things/change wording?)

3.3 Printing (PrintTrace.hs

General functions for printing traces in Markdown and Latex were the next step. If the trace is supposed to be annotated with a relation, usually a function f that takes a trace and returns a map from events to a set of events has to be supplied. The following is an overview of the most important functions that are provided.

3.4 Markdown

- `toMD remFork t` : prints a markdown table for a trace `t`. If `remFork` is True, fork/join is removed.

- `annotTrace f fShow name t` : like with `toMD`, a trace `t` is printed. Additionally, a function `f :: Show a => ([Event] -> Map Event a)` should be supplied, which computes relations between events.
  `f_show :: a -> String` is a function to show the result of f, and can be just `show` if `a` is an instance of Show. Lastly, String `name` is used as a column name. All this information will be used for an additional column of information on the left of the trace. (Note: this is the only function where f can return a map to arbitrary types of values)

- `annotTraceSet f f_show name t` : the same as `annotTrace`, but without `fShow`. A predefined function for printing sets is used instead, therefore `f` also has to map to sets.

- `interactiveSet f name t` : The same as `annotTraceSet`, but as an interactive prompt where you can choose which events to show the relation for. The events can be chosen by inputing the corresponding location numbers after the prompt. For example, `1 3 10` would later show the fist, third and tenth event. Invalid input leads to an error! Invalid input includes letters, symbols, and numbers that do not have an associated event. Only the chosen events will be shown in the markdown table, and the relation set will also only contain events that were chosen.

(TODO: add examples for usage)

3.5 Latex

A number of imports and macros is being used in the latex code. They are listed in the "Imports" section.

- `latexTrace remFork t` : prints latex code that displays the trace `t` as a string. If `remFork` is True, fork/join is omitted in the output.
- `interactiveLatex f t` : prints latex code for Trace `t`, but can add arrows between events that are in relation. The function `f` has to have the type `f :: [Event] -> Map Event (Set Even` again. There are multiple prompts: The first asks for a name of the graph, which should not be the same as for previously generated graphs (the name is just used to avoid issues with conflicting graph marks and isn't shown). The next prompt asks if fork/join should be included, and the last one if arrows should be drawn for everything that is in relation (y) or not (n). If "n" is chosen for the third prompt, a numbered list of every relation pair is shown. The numbers can be used to choose which arrows should be drawn in the latex graph. Invalid input at this point leads to an error! Invalid input includes letters, symbols, and numbers that do not have an associated pair.

  Some additional notes on the latex output: Relations between events in the same thread are entirely omitted, as it is assumed that they are not of interest. This can technically be changed by removing `delSameThread` from `interactiveLatex`. The curving direction and strength of the bend of arrows is estimated loosely, and might need to be adjusted. Furthermore, labels are not added because they usually just end up covering part of the trace, but the automatic placement choice can be tested by writing some text in `\footnotesize{}`. Lastly, if more or different macros are used for events, the latex output for trace events can be changed in the function `eventL`.

(TODO: add example for latex output)

3.5.1 Imports

```
\usepackage{tikz}
\usetikzlibrary{tikzmark}
\usetikzlibrary{arrows,automata}
\usetikzlibrary{trees,shapes,decorations}

\newcommand{\thread}[2]{#1 \sharp #2}
\newcommand{\lockE}[1]{\mathit{acq}(#1)}
\newcommand{\unlockE}[1]{\mathit{rel}(#1)}
\newcommand{\readE}[1]{r(#1)}
\newcommand{\writeE}[1]{w(#1)}
\newcommand{\forkE}[1]{fork(#1)}
\newcommand{\joinE}[1]{join(#1)}

\newcommand{\ba}{\begin{array}}
\newcommand{\ea}{\end{array}}
\newcommand{\bda}{\[\ba}
\newcommand{\eda}{\ea\]}
```

3.6 Trace Reordering

3.6.1 Valid Trace Reorderings

First, let's review which trace reorderings are valid and what kind of functions we need to ensure validity.

3.6.2 Baseline: Checking all Permutations

Our baseline for comparing new implementations is an already existing approach, which computes all possible permutations of a trace and checks each one for validity.
When checking for validity, the PWR relation can be used; because it is complete, a valid reordering will not change the computed relations. We also need to check that the traces still respect write-read dependencies (the last write of each read in the original trace must

also be the last write of each read in the new trace), because otherwise the program would probably not behave the same. Lastly, we also need to check that lock semantics (for every release of a lock there is a proceeding aquire, and there are no two consecutive aquires), because such traces would not represent a valid program execution.

For efficiency, changes in the PWR relation are computed only if the write-read dependencies and lock semantics are correct, because it is the most resource-intensive task in the computation.

As a result, we obtain a complete list of all valid trace reorderings, but at the cost of a significantly long runtime. Because we consider all permutations, the complexity of this method is $O(n!)$ where n is the amount of events in the trace.

This clearly isn't a feasible method for larger traces, however it can still be used to verify the degree of completeness of our new methods, and can serve as a benchmark.

3.6.3 First Implementation: Naive, Complete Search

The first implemention aims to improve on the benchmark by cutting out the computation of all permutations by building potential reorderings step by step and only considering traces that preserve program order. The thought behind this was that a majority of permutations of the original trace won't be valid due to a violation of program order, so we should avoid considering them in the first place. By going step by step, we should also be able to stop as soon as a violation occurs. To do this, we avoid computing PWR each time, and instead compute it once and see for every event if it can be added without violating the constraints from PWR.

The implementation splits the trace into multiple lists, one for every thread. The order of events within each thread is preserved. Then events are added by going through the lists sequentially, trying every possibilty and stopping immediately if PWR, write-read dependencies or lock semantics are violated. If a full reordering was constructed (= all events from every thread have been used), it is added to the result list. At this point, or if no valid event could be added before all lists were empty, the algorithm backtracks and tries more options.

In the end, like with the benchmark algorithm, a list of all possible reorderings is returned.

3.6.4 Second Implementation: Limited Search with Heuristics

When looking at reordering traces, we already observed that the search space is potentially very large. When looking at a trace withwith k threads and n total events between all threads, there are k^n possible ways to interleave the events, while keeping program order intact. This is still an unfeasible number of different traces to consider, so the search space needs to be further reduced. The following ideas and methods will sacrifice completeness; this isn't a big issue, because a large number of trace reorderings will be uninteresting.
As noted by Madanlal Musuvathi et. al in their paper about the tool *CHESS* Musuvathi et al. [2008], it is a good idea to limit the number of possible preemptions c, as "[it can be expected] that most concurrency bugs happen because of few preemptions happening at the right places" (Musuvathi et al. [2008], pg. 9), which would also reduce the number of possible reorderings to n^c.
Further, most errors probably occur due to a reordering of critical sections, as they would contain data operations which are potentially unsafe. We want to consider such reorderings as a priority.
It is also uninteresting to consider reorderings which introduce preemptions after blocks in which only reads have occured, as the state has not been manipulated during this time.

3.6.5 Third Implementation: Greedy Search

(TODO: this section currently is stated like it is already implemented, which might be more time-intensive than expected. Might have to rewrite to reflect that it is only theoretical) Above, we introduced some heuristics to limit the search space, but still methodically tried all options. A different idea would be to use a greedy algorithm, which avoids backtracking but risks failure for correctly reordered traces. "High-Coverage, Unbounded Sound Predictive Race Detection" Roemer et al. [2018] is a paper that's also about data race detection on traces and attempts to construct reordered traces: A different relation, DC, is used to create a constraint graph; then additional constraints are added to ensure reorderings of critical sections; and lastly, the algorithm attempts to create a valid reordering. The reordering is built in reverse order; two events that might conflict are added first, then events that satisfy the computed restraints are prepended sequentially. Events that preserve the original order of critical sections are prepended first,

because preferring this order results in less failed constructions.
The paper builds on many entirely different ideas on concepts from this work, but still we want to take inspiration from the general ideas for reorderings.
As already mentioned, we are switching to a greedy algorithm while still trying to use the ideas from previous implementations. To avoid failure, we also use the idea of trying to keep the order of critical sections intact. To ensure that the traces are still relevant, i.e. they contain potential races, we use the idea of identifying potential race pairs and making sure they're included next to each other in the reordering. Potential conflicting pairs are computed from PWR.

4 Related Work

most of this was already touched on, summarize again?

5 Conclusion

(TODO: thoughts on what went well and what didn't)

(TODO: thoughts on further work that could be done)

Bibliography

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 267–280. USENIX Association, 2008. ISBN 978-1-931971-65-2. URL `http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf`.

Jake Roemer, Kaan Genç, and Michael D. Bond. High-coverage, unbounded sound predictive race detection. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 374–389. ACM, 2018. ISBN 978-1-4503-5698-5.

Martin Sulzmann and Kai Stadtmüller. Efficient, near complete and often sound hybrid dynamic data race prediction (extended version). *CoRR*, abs/2004.06969, 2020.