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

September 19, 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.

Endingen, 19.09.22

Place, Date

[illegible]

Signature

Contents

1 Introduction

This project aims to further explore the PWR Relation, which was introduced by Martin Sulzmann & Kai Stadtmüller in "Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction" [4].
The PWR Relation is used in an algorithm for efficient data race prediction on a trace of a single program run; the relation itself is complete, but unsound.
Instead of exploring more efficient data race prediction itself, we want to explore the relation they defined and create useful tools for working with it in Haskell, building in part on existing implementations by Martin Sulzmann.
The first requirement for being able to work with PWR in Haskell was an already provided DSL for representing traces. Using this, PWR itself had to be implemented. The implementation was later modified to be more general and work with different data representations, namely vector clocks and sets of events.
After implementing PWR, a tool for printing traces and annotating those traces with the computed relation was implemented. By printing annotated traces in Markdown, the results can be made more easily readable, and the output can be filtered. Furthermore, there is the option of automatically generating LaTeX code snippets to automate the process of making graphs.
The last part of the project is about reordering traces. By creating valid reorderings, we can improve testing capabilities. We aimed to use the PWR relation while computing reorderings - because the relation is complete, the relation between events will stay the same for each valid reordering. We are able to use this property to check which reorderings are valid, and which are not. The focus is on generating all possible reorderings by trying to add events sequentially and using backtracking. As a future improvement, it might be interesting to try and only generate reorderings that are likely to show data races. This could be achieved using heuristics; some ideas for such heuristics and general comparisons to other existing approaches are included at the end of this report.

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; we assume that the first thread is always the main thread, and there can be an arbitrary number of other threads that are created by calling a forking function in the main thread. Because concurrent programs are non-deterministic, the traces obtained from different executions can be different each time.

Traces, for our purposes, are lists of events, where only read/write events on shared variables and operations on locks are of importance. We include fork/join 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.

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

	$0\sharp$	$1\sharp$
1.	$fork(2)$	
2.	$acq(y)$	
3.	$w(x)$	
4.	$rel(y)$	
5.		$r(x)$
6.		$acq(y)$
7.		$r(x)$
8.		$rel(y)$

Each column in the table stands for a different thread, here there are two. Each row contains one event, and each row number stands for the trace position of the event. The trace position is sufficient to identify events, but we will usually write down the thread id, event and trace position, like in this example: $0\sharp w(x)_3$. This stands for the write event in row three. The notation follows the convention from the original paper on PWR [4].

We use abbreviations for the possible events:

- $w(x)$: Writes some value to the variable x
- $r(x)$: Reads the value of the variable x
- $acq(y)$: Acquires the lock variable y, starting a critical section
- $rel(y)$: Releases the lock variable y, ending a critical section
- $fork(t)$: Creates a new thread with identification number t
- $join(t)$: Merges the thread with number t with the thread that called join

In the example trace above, the first event creates thread $1\sharp$ from the main thread $0\sharp$. $0\sharp acq(y)_2$is an acquire event on the lock variable y, followed by a write on the shared variable x and a release of the lock variable y. Afterwards, the events in $1\sharp$ were logged, which are a read on the previously written variable x that is not protected by a lock, followed by another lock-read-release sequence.

2.1.1 Data Races

If two events are 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, we consider them conflicting. This is not enough to predict if the events are in a possible data race, but all data races happen on conflicting events.
If conflicting events could appear directly next to each other in a trace, they are in a data race. To prove that events are in a data race, we want to find valid reorderings of the trace in which they do appear next to each other. Such reorderings represent traces that could be obtained from a successful execution of the original program.
A valid reordering of a trace is one that respects the following criteria:

- Program order (PO): The order of events in each thread is not changed.
- Read-write dependency (ROD): Before every read, there is the same last write as in the original trace.
- Lock semantics: Before every release of a lock, there is a corresponding acquire of the same lock in the same thread; each lock is held by at most one thread at the same time.

Considering the trace above, there does not 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, $1\sharp r(x)_5$, from occurring right after $0\sharp w(x)_3$. Putting these events next to each other would be a valid reordering because the three criteria are respected: No events are moved before or after other events in the same thread, the read still happens after the last write, and no lock semantics are violated.
Therefore, there is a valid trace reordering where two conflicting events are next to each other, and we can predict that the original program contains a data race.
As stated before, not all events that are conflicting have to be part of a data race, because in valid traces some events must happen in a particular order. For example, $1\sharp r(x)_7$ could not happen right after $0\sharp w(x)_3$, because the resulting reordered trace would violate lock semantics.
To efficiently find out which events may participate in a data race, based only on a single trace, is the goal of algorithms like PWR.

2.2 The PWR Algorithm

The PWR Algorithm is a complete, often sound data race prediction method and works with traces like the ones explained above. The algorithm used in the original paper [4] is only near-complete, because for efficiency completeness is sacrificed in the actual implementation. This work is based on the theoretical, actually complete PWR relation. The relation tries to create an ordering on events by finding out which of them must happen before each other in valid trace reorderings. There is already a lot of similar work on the subject:
Happens-Before relations are efficient, but neither sound nor complete [1]. Lamport's Happens-Before relation (HB) works by ordering critical sections based on when they appear in the original trace. All events that are unordered are reported as a potential data race. This can lead to both false positives and false negatives. For example, it might be possible to reorder critical sections, which could lead to false negatives; and events outside of critical sections could also be unable to occur next to each other due to write-read dependency, but HB would report a false positive. The relation can be strengthened by including write-read dependencies, but would still be incomplete, because critical sections are ordered by their position in the trace. [4]

Another approach are lockset methods, where we associate each event with the set of locks that was held by the thread when the event was executed. If these locksets for conflicting events are disjoint, they are considered to be in a data race. This is a complete method, but results in many false positives [4].
Hybrid methods pair happens-before with lockset. PWR is such a hybrid method; it maintains completeness while attempting to strengthen the relation, resulting in less false positives.
Two events e and f are in the PWR-relation $<_{PWR}$ if the following conditions are satisfied [4]:

- Program order (PO): If two events are in the same thread, and e is positioned before f, then $e <_{PWR} f$.
- Write-read dependency (WRD): If e is a write, f is a read, and e is the last write before f (e is positioned before f and there is no other write in between), then $e <_{PWR} f$.
- Release-order dependency (ROD): If e and f are in different critical sections CS_e and CS_f on the same lock variable y, and $e <_{PWR} f$, then $rel(y)$ of $CS_e <_{PWR} f$.

Locksets are needed to check if ROD holds. Events are processed one after the other, and we save information on the computed relation $<_{PWR}$ after each event that is processed. We represent the relation in two ways: Vector clocks, and sets of previous events.
Vector clocks associate a time stamp with each thread. Every time an event in a thread is processed, the time stamp for the corresponding thread is incremented by one in the vector clock of that thread. If we know an event must happen after another one, clocks are synchronised, i.e. we take the maximum of each time stamp.
For two vector clocks V_1 and V_2, $V_1 < V_2$ if all time stamps in V_1 are smaller or equal to those in V_2, and at least one time stamp is strictly smaller. If the vector clocks for two events are in the $<$ relation, the events themselves are in the $<_{PWR}$ relation.
For easier readability, we will later also use sets of events to represent the results of the PWR computation; each event is associated with a set of events that must happen before it (the event itself is also included in this set). If the set S_1 for an event e is a subset of a S_2 for an event f, then $e <_{PWR} f$. Using sets results in a is a less compact, but more easily readable representation.

3 Approach

We are creating a Haskell DSL for creating and manipulating traces, as well as working with PWR on those traces. The code discussed here can be found on `https://github.com/lh535/datarace-prediction`.
Files that are based on previous work by Martin Sulzmann, found at `https://github.com/sulzmann/source`, have a note in the title referring to the original file.

3.1 Implementing Traces (Trace.hs, based on source/Trace.hs)

The implementation of traces was largely based on existing code by Martin Sulzmann. A new addition is the distinction between pre-traces, which are lists of events without correct positions, and valid traces, which are a separate type.
Traces have the type `newtype Trace = Trace [Event]`. Events are defined like this:

```
data Event = Event { loc :: Loc,        -- event position. Def.: newtype Loc = Loc Int
                     op :: Op,          -- Type of operation. See data Op
                     thread :: Thread } -- Definition: newtype Thread = Thread Int
```

The operations are defined like this:

```
data Op = Read Var      -- Definition of Var: newtype Var = Var String
        | Write Var
        | Acquire Lock  -- Definition of Lock: newtype Lock = Lock String
        | Release Lock
        | Fork Thread   -- Definition of Thread: newtype Thread = Thread Int
        | Join Thread
```

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 y` (acquire event with t=current thread, y=lock variable)

- `relE t y` (release event with t=current thread, y=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 position, however the functions above do not automatically add one. To create a valid trace from a list of events, `addLoc t` can be used, which takes a list `[Event]` and returns a trace of type `Trace` with added positions.

As an example, consider the following small trace:

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

Figure 1: Example 1

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 addLoc $ [ 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 [4]. Like in the paper, we maintain vector clocks for every thread, and increment and sync these vector clocks in accordance with the given pseudo code. Some adjustments had to be made, because we are only computing the PWR relation, not trying to output potential races.

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

This is the definition of the data type `PWRGlobal` used for storing the current state:

```
newtype TStamp = TStamp Int  deriving (Ord, Eq, Num)
newtype VClock = VClock {clock :: Map Thread TStamp}
data Epoch     = Epoch {t ::  Thread, tstamp :: TStamp}
data HistEl    = HistEl {epoch :: Epoch, vclock :: VClock}

data PWRGlobal = PWRGlobal {
    lockS       :: Map Thread (Set Lock), -- current lockset per thread
    vClocks     :: Map Thread VClock,     -- current vector clock per thread
    lastW       :: Map Var VClock,        -- vector clock of last write on var
    acq         :: Map Lock Epoch,        -- epoch of last acquire for lock
    hist        :: Map Lock [HistEl]}     -- lock history
```

Because we work through each event sequentially, the state and vector clocks need to be modified often and we mostly use the Map type instead of lists.

The function `pwr` takes a trace and processes each event with a function that matches the operation that is performed in the event. The function for each type of event increments the clock of the thread and updates the state. After changing the global variables accordingly, the vector clock for the processed event is saved in the result type `newtype EventVC = EventVC {clocks :: Map Event VClock}`.

3.2.2 Current Implementation (PWR.hs)

The currently used implementation of PWR mainly builds on the previous version and generalizes it.

Using vector clocks to compute the relation is compact, however it is not easily readable at a glance. Therefore, it could be useful to use sets instead; one observation was that the operations on vector clocks and sets are fairly similar. The general operations we need to do on vector clocks were rather limited: We maintain a clock for each thread, and mostly increment time stamps for processed events and synchronise clocks with different threads when events are ordered. Incrementing corresponds to adding an event to a set, and synchronising is the same as generating the union of the sets of events of two threads. To take advantage of this similarity, a new class `PWRType` is defined; it specifies the operations that need to be supported by a type to compute the PWR relation on it.

```
class PWRType a where
    startState :: a           -- state at the beginning
    inc :: Event -> a -> a    -- incrementing
    union :: a -> a -> a      -- to synchronise
    before :: a -> a -> Bool  -- corresponds to < relation
    extend :: Thread -> Map Thread a -> Map Thread a  -- for forking
```

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` with definition `data R a = PWRType a => EventMap {eventMap :: Map Event a}`.

The final type of PWR is `pwr :: PWRType a => Trace -> R a`.

`pwrClock t` and `pwrSet t` can be used if either vector clocks or sets should be used.

There are also specialized functions for pwr printing:

- `annotatedWithPWR :: Trace -> IO ()` prints a markdown table of the trace and the PWR vector clocks
- `annotatedWithPWRSet :: Trace -> IO ()` prints a markdown table of the trace and the PWR sets
- `interactivePWR :: Trace -> IO ()` is the PWR version of `interactiveSet` and can print markdown for only some events. See 3.3.1
- `interactiveLatexPWR :: Trace -> IO ()` can print latex code of the trace, with the option to add arrows for events that are in the PWR relation. See 3.3.2

As an example, we'll look at the trace Example 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
```

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 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.

3.3.1 Markdown

- `toMD :: Bool -> Trace -> IO ()` Prints a markdown table for a trace. If the boolean argument is True, fork and join events are removed.

-
  ```
  annotTrace :: (Trace -> Map Event a) -- computes relations between events
             -> (a -> String)          -- show the result of the computed relation
             -> String                 -- column name
             -> Trace
             -> IO ()
  ```

  Like with `toMD`, a trace is printed, but with an additional column to the right of the trace with information on a computed relation between events.

- `annotTraceSet :: (Trace -> Map Event (Set Event)) -> String -> Trace -> IO ()` The same as `annotTrace`, but without the function for showing the result. A predefined function for printing sets is used instead.

- `interactiveSet :: (Trace -> Map Event (Set Event)) -> String -> Trace -> IO()` 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 inputting 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.

Here are a few usage examples:

```
$ annotTrace (eventMap . pwrClock) show "Vector Clocks" ex2 -- = annotatedWithPWR
   T0        T1        Vector Clocks
1.  wr(x)               t0: 2
2.  fork(t1)            t0: 3, t1: 0
3.            wr(x)     t0: 3, t1: 1
4.  join(t1)            t0: 4, t1: 1
5.  rd(x)               t0: 5, t1: 1

$ toMD True ex2  -- removes Fork/Join
   T0        T1
1.  wr(x)
3.            wr(x)
5.  rd(x)

$ interactiveSet (eventMap . pwr) "PWR Sets" ex2
   T0        T1
1.  wr(x)
2.  fork(t1)
3.            wr(x)
4.  join(t1)
5.  rd(x)
 Choose Events to show relation for (type event locations, separated by a space):
> 1 3 5
   T0        T1        PWR Sets
1.  wr(x)               wr(x)_1
3.            wr(x)     wr(x)_1, wr(x)_3
5.  rd(x)               wr(x)_1, wr(x)_3, rd(x)_5
```

3.3.2 Latex

A number of imports and macros is being used in the latex code. They are listed in the "Imports" section 3.3.2.

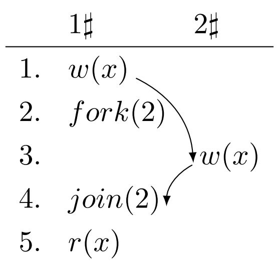
- `latexTrace :: Bool -> Trace -> IO ()` : prints latex code that displays the trace as a string. If `remFork` is True, fork/join is omitted in the output.
- `interactiveLatex :: (Trace -> Map Event (Set Event)) -> Trace -> IO ()` : prints latex code for Trace `t`, but can add arrows between events that are in relation. The function argument is used to compute the relation between events. 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 behavior 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 by hand. Because the ideal bend would be different for every arrow and it is mostly a choice of aesthetics, automating this part of the process would be difficult.
If more or different latex code should be used for displaying events in the trace, the latex output for trace events can be changed in the function `eventL`.
Some usage examples:

```
$ latexTrace False ex2
\bda{lll}
 & \thread{1} & \thread{2}\\ \hline
1. & \writeE{x}&\\
2. & \forkE{2}&\\
3. & &\writeE{x}\\
4. & \joinE{2}&\\
5. & \readE{x}&\\
 \eda{}
```

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$fork(2)$	
3.		$w(x)$
4.	$join(2)$	
5.	$r(x)$	

```
 $ interactiveLatex (eventMap . pwr) ex2
   T0        T1
1.  wr(x)
2.  fork(t1)
3.            wr(x)
4.  join(t1)
5.  rd(x)

Choose a unique graph name:
> pwr
Do you want to remove fork/join? (y/n)
> n
Do you want to include all possible pairs? (y/n)
> n

These are the pairs of Events in the Relation. Type index of pairs to include in
latex graphic, separated by a space.

1: (wr(x)_1,wr(x)_3)
2: (fork(t1)_2,wr(x)_3)
3: (wr(x)_3,join(t1)_4)
4: (wr(x)_3,rd(x)_5)

> 1 3

\bda{lll}
 & \thread{1} & \thread{2}\\ \hline
1. & \writeE{x} \hspace{0.2em}  \tikzmark{r-pwr1}&\\
2. & \forkE{2}&\\
3. & & \tikzmark{l-pwr3} \hspace{0.2em} \writeE{x}\\
4. & \joinE{2} \hspace{0.2em}  \tikzmark{r-pwr4}&\\
5. & \readE{x}&\\

 \eda{}
 \begin{tikzpicture}[overlay, remember picture, yshift=.25\baselineskip,
...  -- rest of output omitted
```



Imports

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

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

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

3.4 Trace Reordering

3.4.1 Valid Trace Reorderings

In the section about data races 2.1.1 we briefly discussed what requirements a valid reordering of a trace must fulfill. The criteria that should not be violated are program order (PO), read-write dependency (RWD), and lock semantics.

We want to compute all possible valid trace reorderings, given one trace as a starting point. We use PWR in addition to some additional checks to ensure that the reorderings fulfill the criteria above. Multiple ideas are possible; as a baseline we use an already existing implementation by Martin Sulzmann.

3.4.2 Baseline: Checking all Permutations (ReorderNaive, based on source/Reorder.hs)

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 relation. We also need to check that the traces still respects WRD (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 not behave the same. Lastly, we also need to check that lock semantics are respected, because otherwise the trace would not correspond to a valid program execution.

For efficiency, changes in the PWR relation are computed only if WRD 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 suboptimal 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 completeness of our new method, and can serve as a benchmark.

3.4.3 Own Implementation: Naive, Complete Search (TraceReorder.hs)

This implementation aims to improve on the benchmark by cutting out the computation of all permutations. By building potential reorderings step by step and therefore only considering traces that preserve program order, there are many invalid interleavings of events that can already be ruled out and won't be considered by this method.
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 check for every event if it can be added without violating the computed constraints from PWR.
The implementation splits the trace into multiple lists of events, one for every thread. The order of events within each thread is preserved. Events are added to an attempted trace reordering by going through the lists sequentially, trying every possibility and stopping immediately if PWR, WRD or lock semantics are violated.
We keep track of where we are in each list by saving the index of the event that hasn't yet been added for every thread. If a full reordering was constructed, i.e. all events from every thread have been used, the reordering 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 one step and tries to add an event from the next thread.
The algorithm ends when In the end, like with the naive algorithm, a list of all possible reorderings is returned. Here is an example of the computation for a trace:

$ex5$

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

```
$ reorder ex5
[Trace [fork(t1)_1,acq(x)_5,wr(y)_6,rel(x)_7,acq(x)_2,wr(y)_3,rel(x)_4],
 Trace [fork(t1)_1,acq(x)_2,wr(y)_3,rel(x)_4,acq(x)_5,wr(y)_6,rel(x)_7]]
```

3.4.4 Benchmarking (ReorderBenchmark.hs, Profiling.hs, reorder-benchmark.html, profiling.prof)

We want to test the performance of the backtracking approach in general, and against the naive approach which checks all permutations.
The expectation is that the backtracking approach will perform significantly better for longer traces, but still won't be very fast, since we exhaustively search for all options.
We also expect to see some slight differences depending on how the trace looks. For example, if there are not many threads, most interleavings will be ruled out because they violate program order, which would be fast for the backtracking algorithm.
We use a few short examples to measure CPU time and RAM usage. Here is an overview of the examples:

- "Short" (length: 5): A simple trace consisting only of writes and one read in two threads
- "Write-Read" (length: 10): A slightly longer trace, consisting of alternating writes and reads in two threads
- "Regular" (length: 10): The same length as Write-Read, but with three threads and more types of events, such as acquire and release.
- "NaiveStress" (length: 11): Similar to Regular, but with one more event, more lock/unlocks and conflicting write/reads. For longer examples, the naive approach starts to struggle to finish (cutoff: 10 minutes)
- "Lock-Unlock" (length: 14): Mostly consists of locks and unlocks in three threads.
- "More-Threads" (length: 14): Consists of mostly writes and reads in five threads.
- "BacktrackStress" (length: 27): A stresstest for the backtracking algorithm with a variety of events

To measure the CPU time used by the algorithms we use criterion. The benchmark can be run with `cabal run reorder-benchmark`. The files were compiled with `-O2`. The detailed result is already saved in "benchmark.html". Here is an overview:

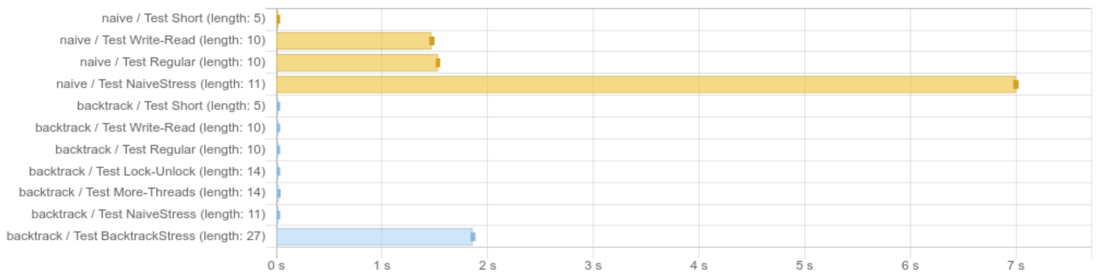


Figure 2: All tests. Yellow: Naive Algorithm, Blue: Own Algorithm using backtracking

We can see that the backtracking algorithm significantly outperforms the naive algorithm which checks all permutations, once the number of events in the trace increases. We can also see that for larger traces such as BacktrackStress, our algorithm also already starts to slow down. These observations match our expectations. We also want to look more closely at differences between the tests Lock-Unlock and More-Traces:

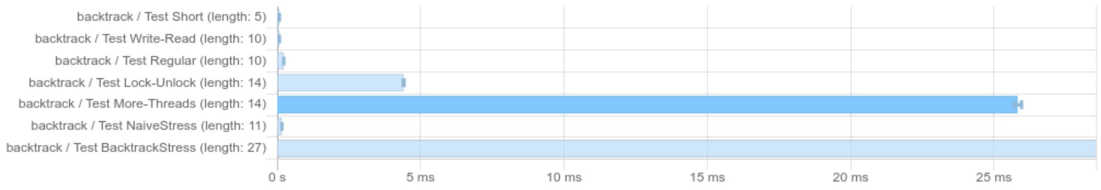
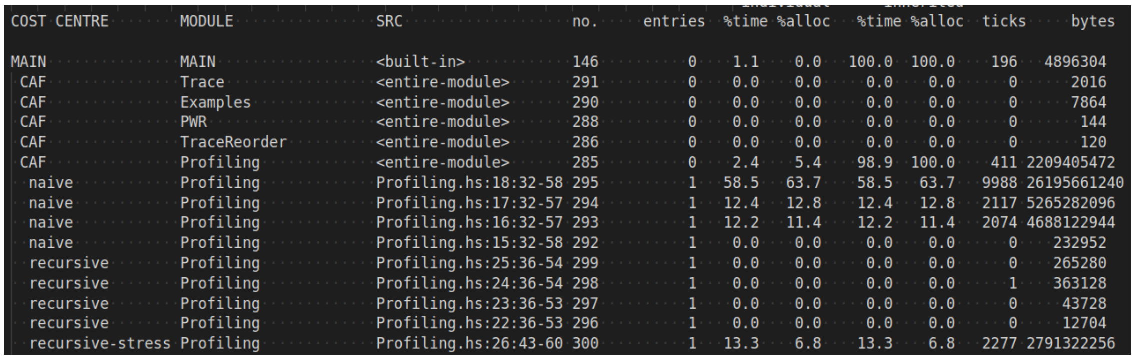


Figure 3: Zoomed-in version of the tests for the backtracking algorithm

Here, we can see that more threads for the same amount of events result in a worse performance. This also matches what we expected - more threads mean more options to try, instead of just being able to choose the next event from the same thread.
The test Regular is also slower than Write-Read, which is probably because there are few possible valid reorderings. The algorithm doesn't have to backtrack often.

For a rudimentary RAM usage analysis, we use GHC's profiling system. The profiling can be run with `cabal run profiling -- +RTS -P`, the result is already saved in "profiling.prof".
We run just the four tests that can be run on by the naive algorithm and compare the usage of the two algorithms on all tests together against each other.



```
COST CENTRE      MODULE            SRC                      no.     entries  %time %alloc   %time %alloc  ticks       bytes

MAIN             MAIN              <built-in>               146           0    1.1    0.0   100.0  100.0    196     4896304
 CAF             Trace             <entire-module>          291           0    0.0    0.0     0.0    0.0      0        2016
 CAF             Examples          <entire-module>          290           0    0.0    0.0     0.0    0.0      0        7864
 CAF             PWR               <entire-module>          288           0    0.0    0.0     0.0    0.0      0         144
 CAF             TraceReorder      <entire-module>          286           0    0.0    0.0     0.0    0.0      0         120
 CAF             Profiling         <entire-module>          285           0    2.4    5.4    98.9  100.0    411  2209405472
  naive          Profiling         Profiling.hs:18:32-58    295           1   58.5   63.7    58.5   63.7   9988 26195661240
  naive          Profiling         Profiling.hs:17:32-57    294           1   12.4   12.8    12.4   12.8   2117  5265282096
  naive          Profiling         Profiling.hs:16:32-57    293           1   12.2   11.4    12.2   11.4   2074  4688122944
  naive          Profiling         Profiling.hs:15:32-58    292           1    0.0    0.0     0.0    0.0      0      232952
  recursive      Profiling         Profiling.hs:25:36-54    299           1    0.0    0.0     0.0    0.0      0      265280
  recursive      Profiling         Profiling.hs:24:36-54    298           1    0.0    0.0     0.0    0.0      1      363128
  recursive      Profiling         Profiling.hs:23:36-53    297           1    0.0    0.0     0.0    0.0      0       43728
  recursive      Profiling         Profiling.hs:22:36-53    296           1    0.0    0.0     0.0    0.0      0       12704
  recursive-stress Profiling       Profiling.hs:26:43-60    300           1   13.3    6.8    13.3    6.8   2277  2791322256
```

Figure 4: Excerpt from profiling.prof

We analyse the rows starting with "naive" and "recursion", which correspond to the execution of each of the tests. The tests with the most events are at the top of each group. Again, as expected, the naive algorithm takes most of the time, but also uses much more memory.
In the last row, the test StressRecursive is also included.

3.4.5 Further Idea: Limited Search with Heuristics ([2])

In this and the next sections, we discuss potential improvements and alternatives to the current implementation for computing trace reorderings.
When looking at reordering traces, the search space is potentially very large. Specifically, when looking at a trace with n threads and k total events between all threads, there are n^k possible ways to interleave the events, while keeping program order intact. This is because for each one of the k positions in the trace, we have the option to choose the next event from each one of the n traces, so in total we consider $\underbrace{n * n * ... * n}_{k-times}$ options.
Even with a more efficient algorithm, this number of different traces is still infeasible to consider once the traces get larger. It would be ideal to reduce the search space further. The following method sacrifices completeness; this shouldn't be a big issue, because a large number of trace reorderings will be uninteresting. By uninteresting we mean that the reordering will look very similar to the original trace, or will be unlikely to show a data race.
As noted by Madanlal Musuvathi et. al in their paper about the tool *CHESS* [2], 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" ([2], pg. 9), which would allow us to schedule the rest of the trace automatically, after c

preemptions have been chosen.
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.
Using these or similar heuristics to reduce the search space, we could generate only a set of interesting trace reorderings in a much smaller amonut of time.

3.4.6 Further Idea: Greedy Search ([3])

Aside from introducing heuristics to limit the search space, we could also try a greedy algorithm, which avoids backtracking but risks failing to find a correctly reordered trace, if there is one. Again, the focus should be on finding a valid reordering that could be considered interesting because it is likely to show a data race.
"High-Coverage, Unbounded Sound Predictive Race Detection" [3] is a paper that is also about data race detection on traces and attempts to construct reordered traces. Mainly using a different relation, DC, a constraint graph is created. The algorithm attempts to create a valid reordering from this constraint graph.
The reordering is built in reverse order, where 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 fewer failed constructions.
The paper builds on slightly different concepts from this work, but could still serve as inspiration for improving our own current algorithm.
The idea would be to switch to a greedy algorithm while still trying to maintain the benefits from previous implementations. To avoid failure, we could 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 could still be computed from PWR, and we might be able to use PWR to check for validity as well.

4 Conclusion

In this project, we first implemented an algorithm to compute the PWR relation and improved it; we also implemented useful tools for working with traces, mainly for printing and filtering results.
This should make it easier to work with PWR in Haskell in the future. We implemented a method for generating all possible valid trace reorderings, which uses backtracking to improve on the efficiency of an naive algorithm. We also presented further ideas for improving the generation of reorderings.
All code and some more detailed results can be found at `https://github.com/lh535/datarace-prediction`; credit for the base of the files "Trace.hs" and "ReorderNaive.hs" goes to Martin Sulzmann. The repository of those original files can be found at `https://github.com/sulzmann/source`.

4.1 Personal Experiences

This project was very useful for getting introduced more to program analysis, with the focus on data races, tracing, relations between events and computing reorderings.
It was also worthwhile to implement the PWR algorithm based on the original paper, to get more familiar with actually applying the concepts that are discussed in papers in my own algorithms. Since the algorithm in the paper differs in some aspects to the one I implemented (it only computes the relation and saves the results), there were some tricky issues with finding out which variables are needed and how the data can be represented in Haskell.
The first part about trying to understand and implement PWR went well and formed a good baseline for the following work. It was also a good exercise for getting more familiar with Haskell, a language which I had not used much previously.
Figuring out the functions for printing traces in Markdown and Latex took a bit of

debugging and trial and error, but was certainly useful during the workflow. It is often hard to visualize results from functions when they are returned as just a long list of events. Implementing latex printing in particular was also helpful for creating this report.

The last part of the project, creating trace reorderings, was considerably more complex, because there was little reference to draw from. Many papers didn't go in quite the same direction and used different concepts. I would have liked to implement more ideas for creating valid reorderings, but struggled with the first implementation that uses backtracking, delaying further work. The other ideas for creating reorderings were presented as theories; it might be worthwhile to revisit them at a later date.

I also tried to use the fact that there are two implementations for generating all possible trace reorderings as an opportunity to attempt some benchmarking, but definitely noticed my lack of experience in this area. In the future, I would like to spend more time with analysing results and creating suitable test cases.

Lastly, I struggled with writing the report for my work because I was not quite sure about the workflow and what would be good practice. I hope to improve the preciseness of language and structure in future work. It might be interesting to continue working with PWR or data race analysis in the bachelor thesis, where I might find useful applications for the existing work or further explore aspects of PWR. It would also be possible to look at subjects outside of PWR and look into other areas of program analysis, since some of my knowledge might carry over.

Bibliography

[1] Matilde D'Arpino, Martin Villing, Jeffrey P. Chrstos, and Giorgio Rizzoni. Dynamic modeling for electric vehicle land speed record performance prediction. In *2017 IEEE Transportation Electrification Conference and Expo, Asia-Pacific (ITEC Asia-Pacific)*, pages 1–6, 2017.

[2] 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.

[3] 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.

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