Tools for Concurrency Bug Hunting

CS-E4110 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

December 7th, 2016

Slides by Keijo Heljanko and Olli Saarikivi

Tools for Concurrency Bug Hunting

Tools for Concurrency Bug Hunting

- Static Analysis Tools - Can warn about concurrency bugs, problems with false warnings
- Race Detection Tools - Can detect race conditions, usually needs a testing setup to obtain a least some level of coverage
- Automated Randomized Testing - Can miss concurrency bugs due large number of thread interleavings and large number of data values
- Dynamic Partial Order Reduction (DPOR) - A testing method systematically exploring all different interleavings in a systematic fashion
- Dynamic Symbolic Execution (DSE / Concolic Testing) - Testing approach exploring all different data values in a systematic fashion
- Model Checking - A systematic way to explore all behaviors of a finite state program. See Java Pathfinder2 (JPF, `http://babelfish.arc.nasa.gov/trac/jpf`), and last tutorial
- The rest of this lecture will not be part of exam requirements!

Introduction

- Testing programs is hard due to state space explosion.
- Some solutions:
  - Single threaded with inputs: dynamic symbolic execution (DSE)
  - Multithreaded with no inputs: partial order reduction.
  - Multithreaded with inputs: combination of both.

Contents

1. Introduction to testing multithreaded programs and partial order reduction
2. The approach is more closely described in the ACSD 2012 paper:
   Saarikivi, O., Kähkönen, K., and Heljanko, K.: Improving Dynamic Partial Order Reductions for Concolic Testing:
   - Our improvement to dynamic partial order reduction (DPOR)
   - How to combine DPOR with dynamic symbolic execution
   - Our implementation and experiments

Testing multithreaded programs

- Behavior is affected by schedule $\rightarrow$ we must be able to control scheduling.
- Scheduling can be done on the level of *visible operations*, which are operations that can affect other threads.
- In our approach an execution tree formed of the scheduling decisions is explored.
- Explore the execution tree by repeatedly exploring alternate interleavings.

Example: Exploring execution trees

- Globals:

  int a = 1;

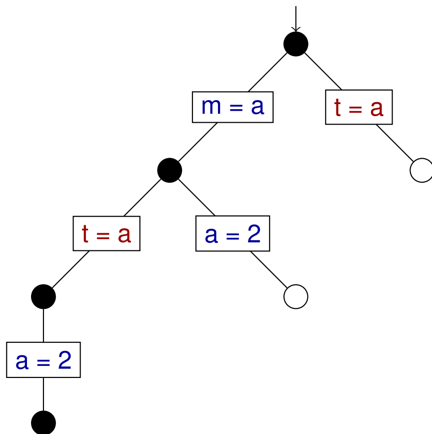
- Thread 1:

  int m = a;
  a = 2;

- Thread 2:

  int t = a;

- Final state:
  m==1
  t==1
  a==2



Example: Exploring execution trees

- Globals:

  int a = 1;

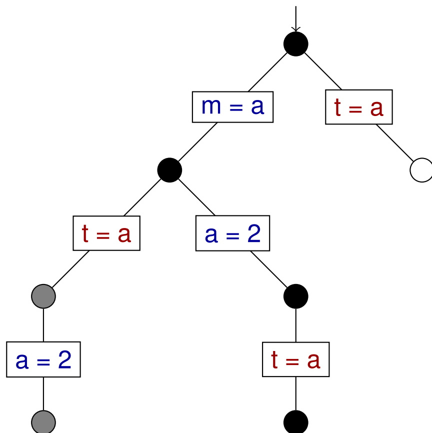
- Thread 1:

  int m = a;
  a = 2;

- Thread 2:

  int t = a;

- Final state:
  m==1
  t==2
  a==2



Example: Exploring execution trees

- Globals:

  int a = 1;

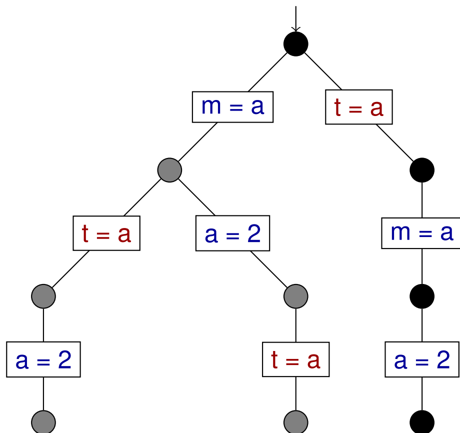
- Thread 1:

  int m = a;
  a = 2;

- Thread 2:

  int t = a;

- Final state:
  m==1
  t==1
  a==2



Partial order reduction

- In this work we consider the case of finding deadlocks and assertion errors.
- For some visible operations the order of execution doesn't matter.
- Partial order reduction methods exploit these independencies to reduce the amount of interleavings explored.

DPOR

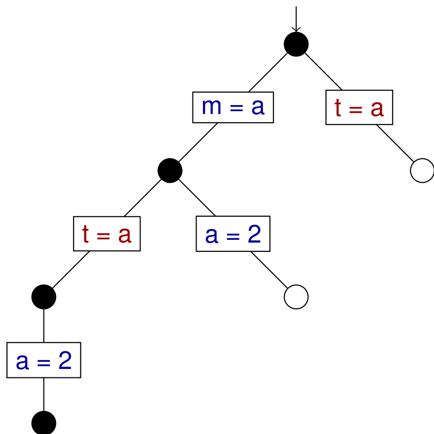
- The dynamic partial order reduction (DPOR) algorithm by Flanagan and Godefroid (2005) calculates what additional interleavings need to be explored from the history of the current execution.
- Once DPOR has fully explored the subtree from a state it will have explored a *persistent set* of operations from that state.
- When a race condition is identified during execution, a *backtracking point* is added to explore the alternate schedule later.
- Backtracking points are explored until no unexplored ones remain.

Identifying backtracking points

- DPOR tracks the causal relationships of visible operations for identifying backtracking points.
- Our implementation uses vector clocks for tracking the causality.
- Also last accesses to communication objects (COs) are tracked.
- A backtracking point is added if:
  1. A thread's next operation uses a previously accessed CO, and
  2. the two visible operations are concurrent.

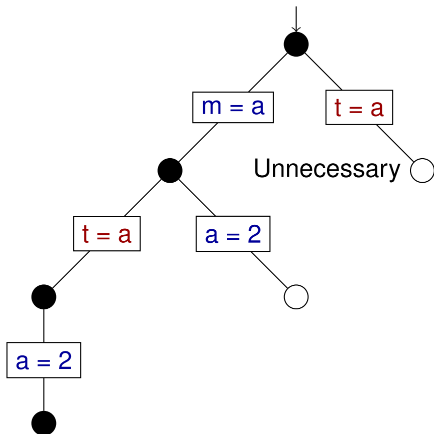
DPOR and concurrent reads

- DPOR uses vector clocks to detect independence of operations
- Original DPOR does not exploit the independence of multiple reads on the same shared variable.
- In the previous example the original DPOR would not have achieved any reduction.



Our modification to DPOR

- Extends DPOR to track the causal structure of reads and writes.
- We have refined the vector clock operations to implement the tracking.
- When identifying backtracking points:
  - For reads we only consider the previous write operation.
  - For writes all reads up to the previous write are examined.



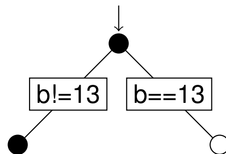
Dynamic Symbolic Execution

- Globals:

```
int a = 1;
int b = input;
```

- Code:

```
int m = a;
if (b == 13) {
    a = 2;
}
```



- First random execution with b = 109 on the right.
- Constraint from first execution: $\neg(b = 13)$
- Solve new inputs with an SMT solver.

Combining the DPOR and DSE

- Globals:

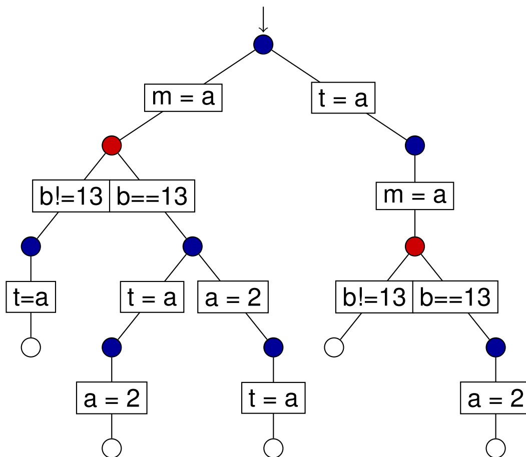
```
int a = 1;
int b = input;
```

- Thread 1:

```
int m = a;
if (b == 13) {
      a = 2;
}
```

- Thread 2:

```
int t = a;
```



About our tool

- The LIME Concolic Tester (LCT) is a tool that uses a client-server model to distribute work to multiple computers over a network.
- Open source and available for download at: `http://www.tcs.hut.fi/Software/lime/`
- We have also implemented *sleep sets* in our tool.
- Further details on how DPOR and sleep sets were implemented in the client-server model with multiple concurrent clients are available in our paper.

Experiments

- We have evaluated our modified DPOR against unmodified DPOR and the "race detection and flipping algorithm" of jCUTE by Koushik Sen and Gul Agha.
- The reduction achieved by DPOR depends on the first random schedules explored. We report the average of several independent measurements.
- For our modified DPOR the effect of using sleep sets was also evaluated.

Experiments cont.

Program	Number of test executions to achieve path coverage			
	DPOR	DPOR-CR	DPOR-CR, sleep sets	jCUTE
Indexer (12)	8614.6	154	27	**8**
Indexer (13)	$>$ 10000	$>$ 10000	722.4	**343**
File System (14)	6.8	3.2	2.6	**2**
File System (16)	568.4	26.8	**19.5**	31
File System (18)	$>$ 10000	250.2	**145.8**	2026
Parallel Pi (3)	$>$ 10000	3217.8	19.2	**6**
Parallel Pi (5)	$>$ 10000	$>$ 10000	1220.6	**120**
Bounded Buffer	64.4	67.2	16	**8**
Sync Queue	$>$ 10000	$>$ 10000	**9**	N/A

Numbers for DPOR columns are averages of 5 separate measurements.

DPOR+DSE Conclusions

- We have modified the DPOR algorithm to exploit the commutativity of read operations.
- We have implemented the modified DPOR algorithm with sleep sets in our testing tool LCT, an open source DSE tool designed for distributed use.
- Our modifications to DPOR allow it to achieve competitive amounts of reduction.

Detecting Data races in DPOR

- Data races are a notoriously common source of bugs in multithreaded programs.
- Traditional definition of data races: A program contains a data race if there is a reachable state where two different operations on the same resource are co-enabled and at least one of the operations is a write.
- Data races may be present only in some rarely reached interleavings of a program, which makes finding them with traditional testing techniques challenging.
- For more details, see:
  Olli Saarikivi, Keijo Heljanko: Reporting Races in Dynamic Partial Order Reduction. Nasa Formal Methods Symposium 2015: 450-456.

DPOR

- The dynamic partial order reduction algorithm (DPOR) by Flanagan and Godefroid (2005) is an efficient testing algorithm that can find assertion errors and deadlocks in multithreaded programs by repeatedly exploring alternate interleavings.
- To achieve this DPOR will explore different outcomes of data races.
- However, the algorithm was not originally designed for reporting data races.

Completeness of DPOR

- DPOR was originally proven to be sound and complete for a sequentially consistent memory model.
- Both Java and C++11 allow for non−sequentially consistent executions for those programs that have data races.
- Thus we would like for DPOR to report data races and also to warn users of possible incompleteness.

Rest of this talk

1. An outline of a proof that if a data race between two operations is reachable then DPOR will visit such a state.
2. A method derived from the proof for reporting data races in DPOR.
3. Experimental results.
4. Subtleties encountered in the Java memory model.

Mazurkiewicz traces

- A *Mazurkiewicz trace* is the set of executions obtainable from each other by swapping adjacent independent operations.
- For the program on the right, with the following dependency relation:

  x y

  a ⟷ b

  The Mazurkiewicz traces are: {xayb, xyab, yxab} and {ybxa, yxba, xyba}
- DPOR guarantees at least one execution from each Mazurkiewicz traces (Flanagan and Godefroid (2005)).

Thread 1:

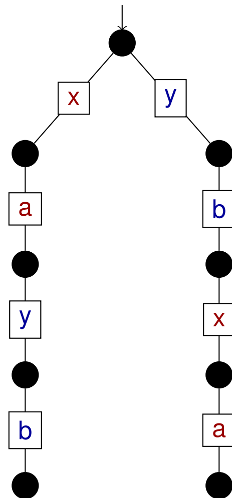
```
x;
a;
```

Thread 2:

```
y;
b;
```

Why Mazurkiewicz traces are not enough

- The execution tree on the right explores all Mazurkiewicz traces, but there is no state in which the racing operations a and b are co-enabled.
- However, DPOR can not produce this execution tree.

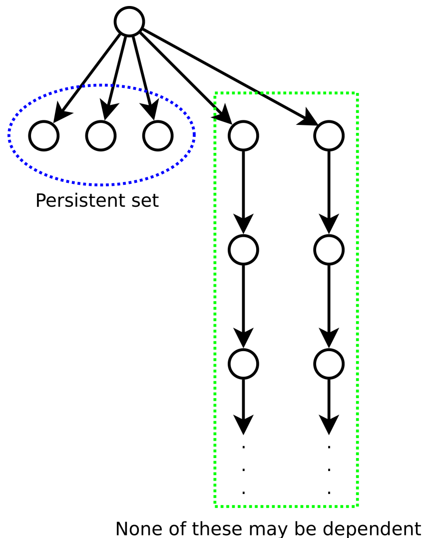


Persistent sets

Given a state s and a set of enabled operations E, a set of operations $P \subseteq E$ is *persistent* in s if:

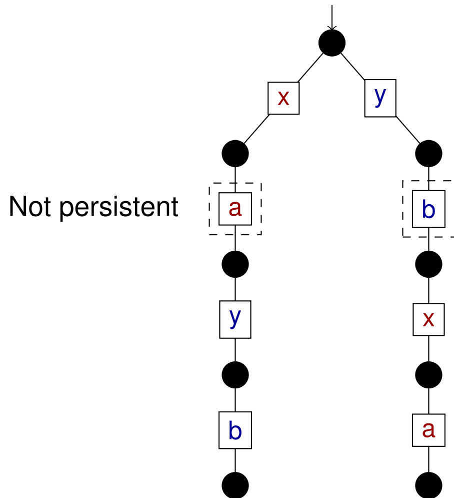
1. All operations in $E \setminus P$ are independent with all operations in P.
2. There is no operation $t \notin P$ in any state reachable from s with operations outside P that is dependent with an operation in P.
3. $P = \emptyset$ if and only if $E = \emptyset$.

DPOR explores a persistent set of operations from each state it explores.



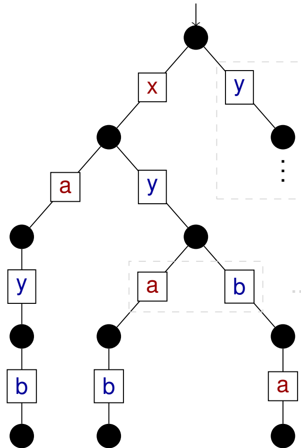
How persistent sets help

- In our running example not all sets are persistent.



How persistent sets help (cont.)

When the explored sets are expanded to be persistent...

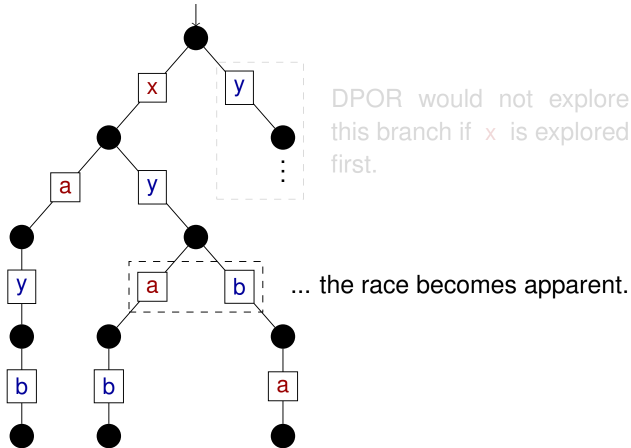


DPOR would not explore this branch if x is explored first.

... the race becomes apparent.

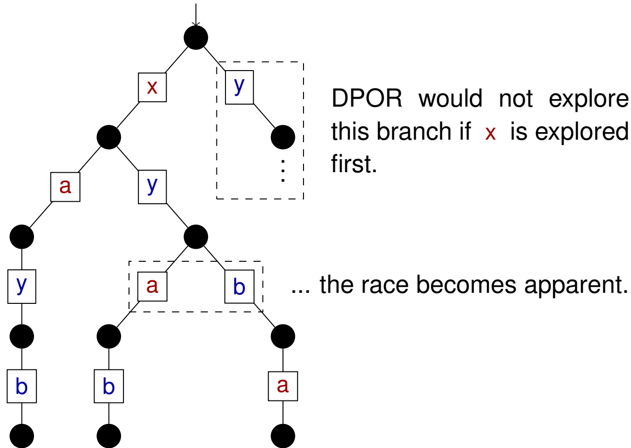
How persistent sets help (cont.)

When the explored sets are expanded to be persistent...



How persistent sets help (cont.)

When the explored sets are expanded to be persistent...



DPOR would not explore this branch if x is explored first.

... the race becomes apparent.

Reachability of Races

Theorem: If there exists a race between two operations a and b then DPOR will reach a state in which a and b are co-enabled.

The proof relies on these properties:

1. DPOR explores at least one execution from each Mazurkiewicz trace.
   This ensures that DPOR explores an execution from the same Mazurkiewicz trace as the one in which the operations are co-enabled.
2. DPOR explores a persistent set of operations from each visited state.
   This allows us to inductively prove that after seeing the execution given by Property 1 DPOR will eventually visit a state where a and b are co-enabled

Reachability of Races

Theorem: If there exists a race between two operations a and b then DPOR will reach a state in which a and b are co-enabled.

The proof relies on these properties:

1. DPOR explores at least one execution from each Mazurkiewicz trace.
   This ensures that DPOR explores an execution from the same Mazurkiewicz trace as the one in which the operations are co-enabled.
2. DPOR explores a persistent set of operations from each visited state.
   This allows us to inductively prove that after seeing the execution given by Property 1 DPOR will eventually visit a state where a and b are co-enabled

Reachability of Races

Theorem: If there exists a race between two operations a and b then DPOR will reach a state in which a and b are co-enabled.

The proof relies on these properties:

1. DPOR explores at least one execution from each Mazurkiewicz trace.
   This ensures that DPOR explores an execution from the same Mazurkiewicz trace as the one in which the operations are co-enabled.
2. DPOR explores a persistent set of operations from each visited state.
   This allows us to inductively prove that after seeing the execution given by Property 1 DPOR will eventually visit a state where a and b are co-enabled

Reporting data races

- Races can be reported by inspecting all enabled operations in each visited state for a pair of dependent operations.
  - This is extremely lightweight.
- It is also sufficient to only inspect pairs of events inside the persistent sets explored by DPOR (proof in paper).
- These methods are precise because the check for a data race is exactly the traditional definition of a data race.
- The method is also sound for C++11 and a large subset of Java (more on that subset after the experiments).

Experiments

We evaluated the overhead of reporting races on a set of benchmarks. The overhead of our method is negligible.

Benchmark	No race detection	With race detection
Szymanski small	89.55 s	**89.41 s**
Pi	**7.00 s**	7.02 s
File system	**1.57 s**	1.61 s
Indexer	11.12 s	**11.07 s**
Synthetic	28.13 s	**28.12 s**

Subtleties in the Java memory model

- The guarantee that Java behaves sequentially consistently is based on a different definition of data races.
- In Java a program has a data race if there exists a sequentially consistent execution in which two dependent operations are not ordered in the program's *happens before* relation.
- Data race free programs in Java are called *properly synchronized*.
- In Java there are programs that do not have traditional races but are still not properly synchronized, and can thus exhibit non-sequentially consistent semantics.

Covert communication in Java

Java guarantees that only the second call to `Thread.start()` throws an exception, but the calls are not considered synchronization w.r.t the memory model.

Thus threads can communicate without shared objects or synchronization primitives! Read and write to `y` are never co-enabled (`x.start()` appears in between) but they are not ordered by the Java memory model!

```
initially: x = new Thread(); y = 0
```

Thread 1:	Thread 2:
`y = 1;`	`try { x.start(); }`
`x.start();`	`catch (Exception e) { r = y; }`

Example from: Andreas Lochbihler, *Unified, Machine-Checked Formalisation of Java and the Java Memory Model*, Karlsruhe Reports in Informatics, Technical Report, Nr. 2011-34, December 2011.

DPOR Race Detection Conclusions

- We have presented a method for reporting races in the DPOR algorithm.
- If the program contains a traditional data race then this method will report it during the execution of DPOR.
- The method has negligible overhead.
- The method can also warn the user that DPOR might not be complete for the program due to weak memory model semantics.
- The method is precise and sound for C++11, and Java programs that do not use covert communication channels.