Aalto University
School of Science

Testing Multithreaded Programs with DPOR

Keijo Heljanko, Olli Saarikivi, Kari Kähkönen

Helsinki Institute for Computer Science (HIIT) and
Department of Computer Science and Engineering,
Aalto University

{ Keijo.Heljanko, Olli.Saarikivi, Kari.Kahkonen}@aalto.fi

9th of June 2014

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.

Topics of this Slideset

1. Introduction to testing multithreaded programs and partial order reduction

2. Our contributions 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:

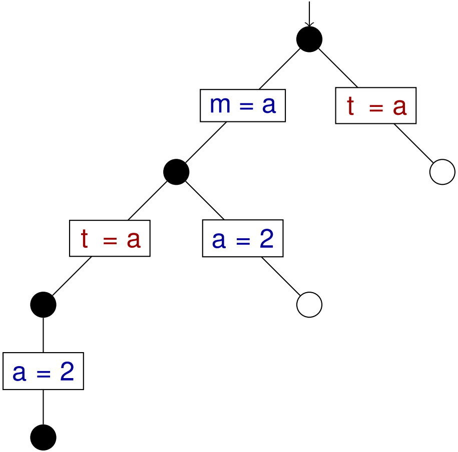
```
int t = a;
```

- 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:

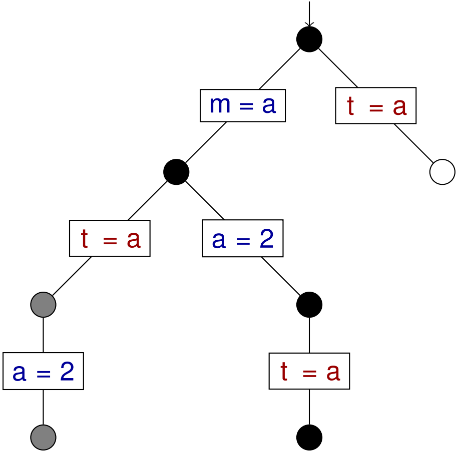
```
int t = a;
```

- 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:

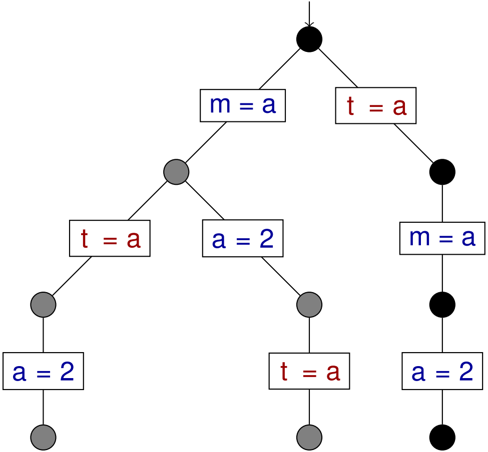
```
int t = a;
```

- 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

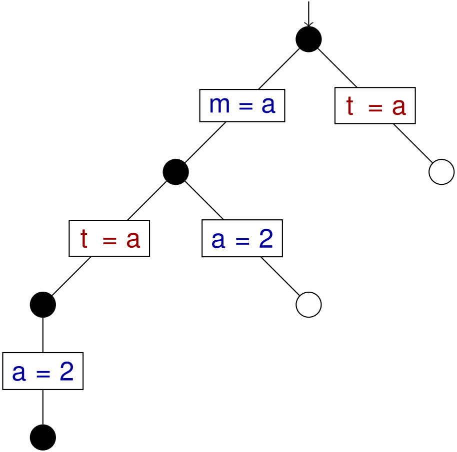
- The dynamic partial order reduction (DPOR) algorithm by Flanagan and Godefroid (2005) calculates what additional interleavings need to be explored from the history of the current execution.
- Once DPOR has fully explored the subtree from a state it will have explored a *persistent set* of operations 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.

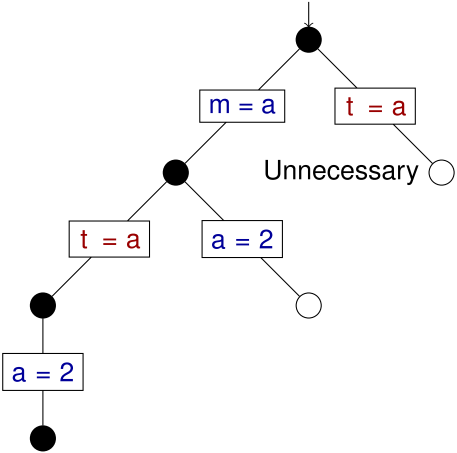
DPOR and concurrent reads

- 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.



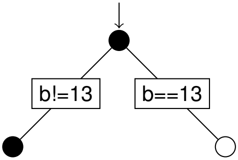
Dynamic Symbolic Execution

- 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:

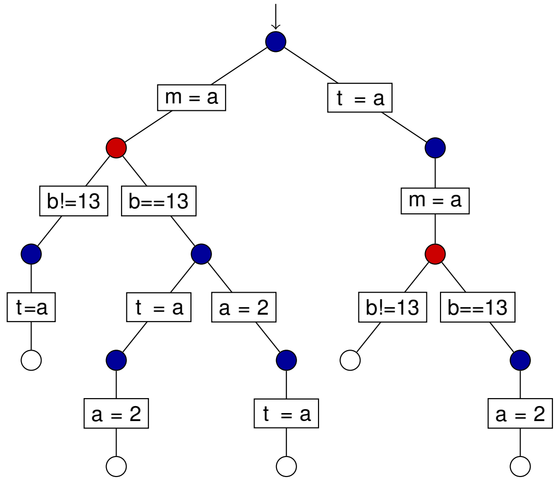
```
int a = 1;
int b = input;
```

- 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.

Conclusion

- 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.