Preemption Bounding Techniques for Dynamic Partial Order Reduction

Σαχίνογλου Γιάννης

ΣΗΜΜΥ - ΕΜΠ

03112089

Summary

Background

Concurrent Programming

Concurrent Programming is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Potential problems include:

Concurrent Programming

Concurrent Programming is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Potential problems include:

- Race Conditions

Concurrent Programming

Concurrent Programming is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Potential problems include:

- Race Conditions
- Deadlocks

Concurrent Programming

Concurrent Programming is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Potential problems include:

- Race Conditions
- Deadlocks
- Livelocks

Concurrent Programming

Concurrent Programming is a form of computing in which several computations are executed during overlapping time periods concurrently, instead of sequentially (one completing before the next starts).
Potential problems include:

- Race Conditions
- Deadlocks
- Livelocks
- Resource Starvation

Concurrency Errors

To be a Concurrency Error or not to be...

```
void *divider(void* arg) {
 int x = 0;
 return 42/x;
}
```

Listing 1: Example of non-concurrency error

```
volatile int x = 1;

void *divider() {
 return 42 / x;
}

void *zero() {
 x = 0;
}
```

Listing 2: Example of concurrency error

Tracking Errors caused by Scheduler's non-determinism

- An interleaving represents a scheduling of the concurrent program.

Tracking Errors caused by Scheduler's non-determinism

- An interleaving represents a scheduling of the concurrent program.
- The state space is the set of all possible interleavings.

Tracking Errors caused by Scheduler's non-determinism

- An interleaving represents a scheduling of the concurrent program.
- The state space is the set of all possible interleavings.
- In order to find an error of a concurrent program, one must examine every possible interleaving.

Tracking Errors caused by Scheduler's non-determinism

- An interleaving represents a scheduling of the concurrent program.
- The state space is the set of all possible interleavings.
- In order to find an error of a concurrent program, one must examine every possible interleaving.
- STATE SPACE EXPLOSION!

Stateless Model Checking and Partial Order Reduction

- We need to preform Systematic Model Checking.

Stateless Model Checking and Partial Order Reduction

- We need to preform Systematic Model Checking.
- Partial Order Reduction aims to reduce the number of interleavings explored by eliminating the exploration of equivalent interleavings.

Stateless Model Checking and Partial Order Reduction

- We need to preform Systematic Model Checking.
- Partial Order Reduction aims to reduce the number of interleavings explored by eliminating the exploration of equivalent interleavings.
- Dynamic Partial Order Reduction: record conflicts during runtime and add appropriate rescheduling points.

An example of Partial Order Reduction (Optimal)

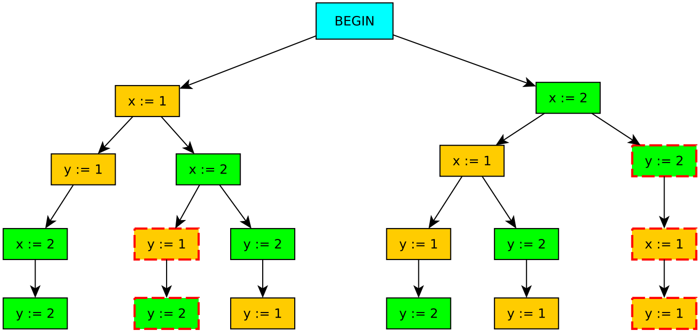

Figure: DPOR tries to avoid red nodes

Sufficient Sets

Definition 1 (Sufficient Sets)

A set of transitions is sufficient in a state s if any relevant state reachable via an enabled transition from s is also reachable from s via at least one of the transitions in the sufficient set. A search can thus explore only the transitions in the sufficient set from s because all relevant states still remain reachable. The set containing all enabled threads is trivially sufficient in s, but smaller sufficient sets enable more state space reduction.

General form of DPOR

```
Explore($\emptyset$);
**Function** *Explore($E$)*
    **let** $T = Sufficient\_set(final(E))$;
    **for** *all* $t \in T$ **do**
        Explore($E.t$) ;
    **end**
```

Algorithm 1: General form of DPOR

Implementation of DPOR Algorithm

```
Explore($\emptyset$);
**Function** *Explore($E$)*
    **if** *suitable $p$ exists* **then**
        $backtrack(E) := \{p\}$ ;
        add to previous $backtracks$ threads due to dependencies revealed by $p$;
        **while** $\exists p \in backtrack$ **do**
            $Explore(E.p)$ ;
        **end**
    **end**
```

Algorithm 2: Implementation DPOR Algorithm

Sufficient Sets: Persistent Sets

Definition 2 (Persistent Sets)

Let s be a state, and let $W \subseteq E(s)$ be a set of execution sequences from s. A set T of transitions is a Persistent Set for W after s if for each prefix w of some sequence in W, which contains no occurrence of a transition in T, we have $E \vdash t \Diamond w$ for each $t \in T$.

Sufficient Sets: Persistent Sets

A simple example:

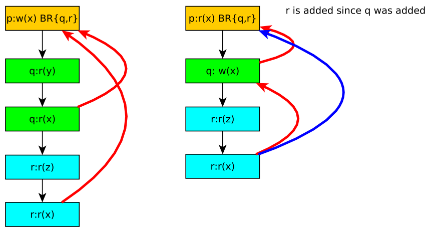

Figure: Construction of Persistent Sets

Sufficient Sets: Source Sets

Definition 3 (Initials after an execution sequence $E.w$, $I_{[E]}(w)$)

$p \in I_{[E]}(w)$ if and only if there is a sequence w' such that $E.w \simeq E.p.w'$.

Definition 4 (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)

$p \in WI_{[E]}(w)$ if and only if there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

Sufficient Sets: Source Sets

Definition 5 (Source Sets)

Let E be an execution sequence, and let W be a set of sequences, such that $E.w$ is an execution sequence for each $w \in W$. A set T of processes is a source set for W after E if for each $w \in W$ we have $WI_{[E]}(w) \cap T \neq \emptyset$.

Source Sets

An example:

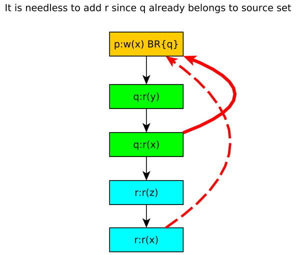
It is needless to add r since q already belongs to source set
p:w(x) BR{q}
q:r(y)
q:r(x)
r:r(z)
r:r(x)

Figure: Construction of Source Sets

Further Optimizations: Sleep Sets

The idea behind Sleep Set Optimization:

- Sleep Sets prevent redundant explorations.

Further Optimizations: Sleep Sets

The idea behind Sleep Set Optimization:

- Sleep Sets prevent redundant explorations.
- Assume that the search explores transition t from state s, backtracks t, then explores t_0 from s instead. Unless the search explores a transition that is dependent with t, no states are reachable via t_0 that were not already reachable via t from s. Thus, t “sleeps” unless a dependent transition is explored.

Sleep Sets

Sleep sets in action (Using Persistent Sets):

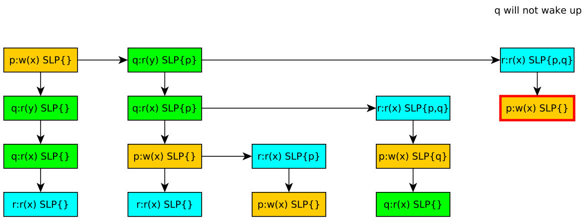

Figure: Example of Sleep Set Optimization

Source-DPOR

```
Explore($\langle\rangle$,$\emptyset$);
Function Explore($E$,$Sleep$)
    if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ then
        $backtrack(E) := p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep)$ do
            foreach $e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                if $I_{E'}(u) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(u)$ to $backtrack(E')$ ;
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \vDash p \Diamond q\}$;
            $Explore(E.p, Sleep')$ ;
            add $p$ to $Sleep$ ;
        end
    end
```

Algorithm 3: Source-DPOR Algorithm

Bounding Techniques for DPOR

- For larger programs DPOR often runs longer than developers are willing to wait.

Bounding Techniques for DPOR

- For larger programs DPOR often runs longer than developers are willing to wait.
- Bounded techniques, alleviate state-space explosion by pruning the executions that exceed a bound.

Bounding Techniques for DPOR

- For larger programs DPOR often runs longer than developers are willing to wait.
- Bounded techniques, alleviate state-space explosion by pruning the executions that exceed a bound.
- Many of the concurrency bugs can be tracked even when the bound limit is set to be small.

Bounding Techniques for DPOR

- For larger programs DPOR often runs longer than developers are willing to wait.
- Bounded techniques, alleviate state-space explosion by pruning the executions that exceed a bound.
- Many of the concurrency bugs can be tracked even when the bound limit is set to be small.
- Preemption Bounded and Delay Bounded exploration.

Preemption Bounded Search

Definition 6 (Preemption bound)

$P_b(\emptyset) = 0$
$P_b(E.t) =$
$$\begin{cases} P_b(E) + 1 & \text{if } t.tid = last(E).tid \text{ and } last(E).tid \in enabled(final(E)) \\ P_b(E) & \text{otherwise} \end{cases}$$

Bounded Dynamic Partial Order Reduction General Form

Given a bound evaluation function B_v and a bound c:

```
Result: Explore the whole statespace
Explore(∅);
Function Explore(E)
    $T = Sufficient\_set(final(E))$
    for all $t \in T$ do
        if $B_v(E.t) \leq c$ then
            Explore($E.t$)
        end
    end
```

Algorithm 4: Bounded-DPOR

Preemption Bounded Persistent Sets

Definition 7 (Preemption Bounded Persistent Set)

A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = final(E)$ is preemption-bound persistent in s iff for all nonempty sequences a of transitions from s in $A_G(P_b, c)$ such that $\forall i \in dom(a), a_i \notin T$ for all $t \in T$,

1. $Pb(E.t) \leq Pb(E.a_1)$
2. if $Pb(E.t) < Pb(E.a_1)$, then $t \leftrightarrow last(a)$ and $t \leftrightarrow next(final(E.a), last(a).tid)$
3. if $Pb(E.t) = Pb(E.a_1)$, then $ext(s,t) \leftrightarrow last(a)$ and $ext(s,t) \leftrightarrow next(final(E.a), last(a).tid)$

Preemption Bounded Persistent Sets

Simplified version:

- Add conservative branches at the beginning of the block.

Preemption Bounded Persistent Sets

Simplified version:

- Add conservative branches at the beginning of the block.
- But what's a block?

Preemption Bounded Persistent Sets

Simplified version:

- Add conservative branches at the beginning of the block.
- But what's a block?

Preemption Bounded Persistent Sets

Simplified version:

- Add conservative branches at the beginning of the block.
- But what's a block?

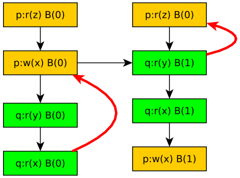

Figure: Example of Blocks and Bound Persistent Sets

Nidhugg

- Nidhugg is a bug-finding tool which targets bugs caused by concurrency and relaxed memory consistency in concurrent programs. It works on the level of LLVM internal representation, which means that it can be used for programs written in languages such as C or C++.
- Nidhugg relies on Source-Sets.

The Nidhugg Flow Chart

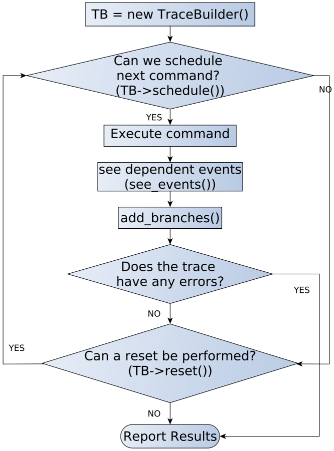
TB = new TraceBuilder()
Can we schedule next command? (TB->schedule())
NO
YES
Execute command
see dependent events (see_events())
add_branches()
Does the trace have any errors?
YES
NO
YES
Can a reset be performed? (TB->reset())
NO
Report Results

Figure: Nidhugg's Flow Chart

Aim of Thesis

- Implement Preemption Bounded Search (BPOR) for Nidhugg.

Aim of Thesis

- Implement Preemption Bounded Search (BPOR) for Nidhugg.
- Explore the potential of Source-Sets for BPOR.

Aim of Thesis

- Implement Preemption Bounded Search (BPOR) for Nidhugg.
- Explore the potential of Source-Sets for BPOR.
- Evaluate the performance of various BPOR implementations.

Aim of Thesis

- Implement Preemption Bounded Search (BPOR) for Nidhugg.
- Explore the potential of Source-Sets for BPOR.
- Evaluate the performance of various BPOR implementations.
- Explore alternative approaches to BPOR problem.

Algorithms

Naive-BPOR

```
Explore($\langle\rangle$,$\emptyset$,$b$);
Function Explore($E$,$Sleep$,$b$)
    if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ such that $B_v(E.p) \leq b)$ then
        $backtrack(E) := p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep$ and $B_v(E.p) \leq b$ do
            foreach $e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                if $I_{E'}(u) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(u) to backtrack(E')$ ;
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$;
            $Explore(E.p, Sleep, b)$ ;
            add $p$ to $Sleep$ ;
        end
    end
```

Algorithm 5: Naive-BPOR

Example execution of Naive-BPOR

A Naive-BPOR execution example and the problem with it.

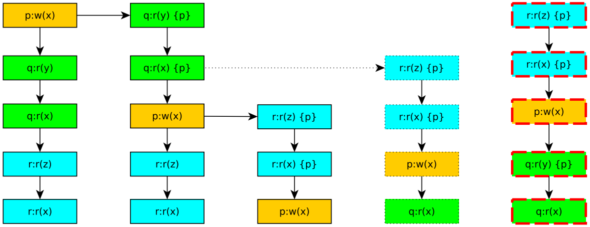

Figure: Naive-BPOR for bound=0

Nidhugg-DPOR

Requirement for BPOR

```
Explore(⟨⟩,∅);
Function Explore($E$,$Sleep$)
    if $\exists p \in (enabled(s_{[E]}) \backslash Sleep$ then
        backtrack(E) := $\{p\}$ ;
        while $\exists p \in (backtrack(E) \backslash Sleep)$ do
            foreach $e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                let $CI = \{i \in I_{E'}(u) \mid i \rightarrow p\}$;
                if $CI \cap backtrack(E') = \emptyset$ then
                    if $CI \neq \emptyset$ then
                        add some $q' \in CI$ to $backtrack(E')$ ;
                    end
                    else
                        add some $q' I_{E'}(u)$ to $backtrack(E')$
                    end
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$ ;
            $Explore(E.p, Sleep)$ ;
            add $p$ to $Sleep$ ;
        end
    end
```

Algorithm 6: Nidhugg-DPOR

Correctness of Nidhugg-DPOR

Case 1: At least one process contains a write command. We know that the two processes will be inverted at some point. Since Nidhugg-DPOR ignores weak initials it will branch both processes. In Source-DPOR only one of the two processes should be branched since they share the same initials. However, in Nidhugg-DPOR this is not true since the CI set does not contain steps from the other process.

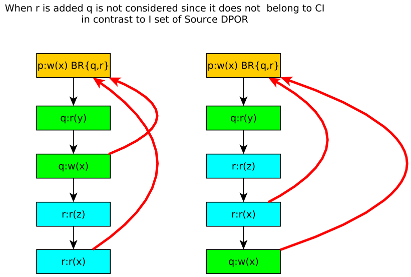
When r is added q is not considered since it does not belong to CI
in contrast to I set of Source DPOR
p:w(x) BR{q,r}
q:r(y)
q:w(x)
r:r(z)
r:r(x)
p:w(x) BR{q,r}
q:r(y)
r:r(z)
r:r(x)
q:w(x)

Figure: Construction of Persistent Sets in Nidhugg when there is a write process

Correctness of Nidhugg-DPOR

Case 2: Both processes are read operations. Since we do not calculate I but CI the first read operation will not be considered as it does not happen before the second read operation and as result both processes will be added to $backtrack$. We notice that by calculating the CI set when the race between p and r is detected q process will be ignored and, thus, r will be added as a branch.

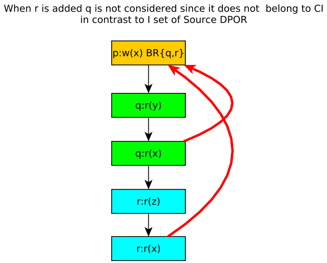
When r is added q is not considered since it does not belong to CI
in contrast to I set of Source DPOR
p:w(x) BR{q,r}
q:r(y)
q:r(x)
r:r(z)
r:r(x)

Figure: Construction of Persistent Sets in Nidhugg when both are read processes

Nidhugg-BPOR

```
Explore(⟨⟩,∅,b);
Function Explore(E,Sleep,b)
    if $\exists p \in ((enabled(s_{[E]}) \setminus Sleep)$ and $B_v(E.p) <= b$ then
        backtrack(E) := $p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep$ and $B_v(E.p) <= b$ do
            foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ do
                add non-conservative branches according to Persistent Sets ;
                add conservative branches according to Persistent Sets at the
                  beginning of blocks ;
            end
            let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$ ;
            $Explore(E.p, Sleep)$ ;
            if $p$ is not conservative then
                add $p$ to $Sleep$ ;
            end
        end
    end
```

Algorithm 7: Nidhugg-BPOR

Source-BPOR: The main question

Can we use source sets instead of Persistent Sets in order to implement BPOR?

Source-BPOR: First approach

We should use Source Sets for both conservative and non-conservative branches.

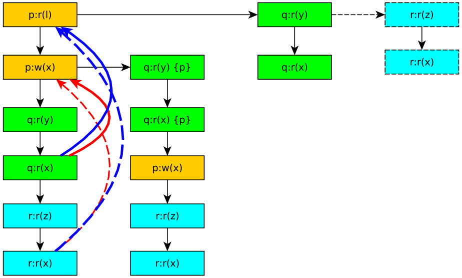

Figure: Following source sets for conservative branches

Source-BPOR: A Correct Approach

We should use Source Sets for non-conservative branches and Persistent Sets for conservative branches.

Source-BPOR

```
Explore($\langle\rangle$,$\emptyset$,$b$);
Function Explore($E$,$Sleep$,$b$)
  if $\exists p \in ((enabled(s_{[E]})\setminus Sleep)$ and $B_v(E.p) <= b$ then
    backtrack(E) $:= p$ ;
    while $\exists p \in (backtrack(E)\setminus Sleep$ and $B_v(E.p) <= b$ do
      foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ do
        add non-conservative backtracks according to Source Sets ;
        add conservative backtracks according to Persistent Sets at the
         beginning of blocks ;
      end
      let $Sleep' := \{q \in Sleep \mid E \models p\Diamond q\}$ ;
      $Explore(E.p, Sleep)$ ;
      if $p$ is not conservative then
        add $p$ to $Sleep$ ;
      end
    end
  end
```

Algorithm 8: Source-BPOR

Nidhugg-BPOR vs Source-BPOR

Similarities:

- Same structure.

Differences:

- Source-BPOR relies on Source Sets for the addition of non-conservative branches while Nidhugg-BPOR relies on Persistent Sets.

Challenges with Conservative Branches

The usage of conservative branches leads to explosion of the state space:

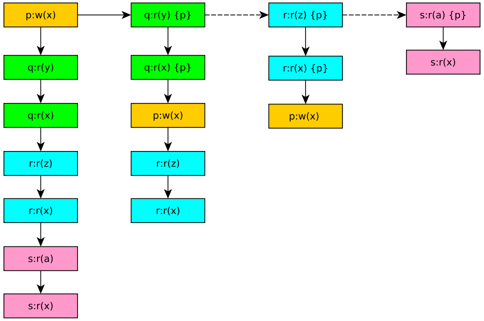

Figure: writer-3-readers explosion

Challenges with Conservative Branches

Sleep Sets are no longer that useful:

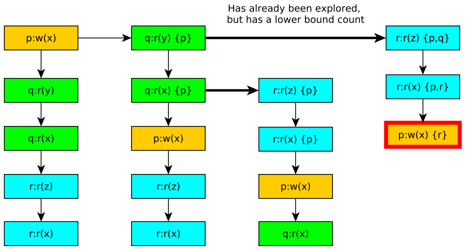

Figure: Sleep set contradiction

Concluding Remarks

The Performance - Soundness Tradeoff: Some algorithms are faster but compromise the soundness of the exploration while others are slower but sound as well.

Evaluation

Nidhugg-DPOR Evaluation

Evaluation of Nidhugg-DPOR on Synthetic Tests

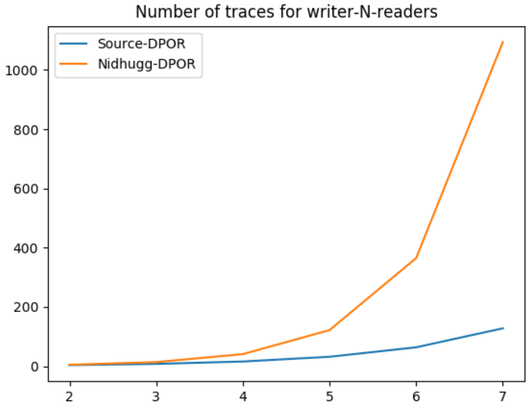

Figure: writer-N-readers

Nidhugg-DPOR Evaluation

Test case	Traces for Source-DPOR	Traces for Classic-DPOR
account.c	6	7
lazy.c	6	7
micro.c	52495	53084
lastzero.c	97	97
lastzeromod.ll	13	17
indexer0.c	8	8
indexermod.c	120	226

Table: Source-DPOR vs Nidhugg-DPOR for Synthetic tests

Nidhugg-BPOR Evaluation

Evaluation of Nidhugg-BPOR on Synthetic Tests

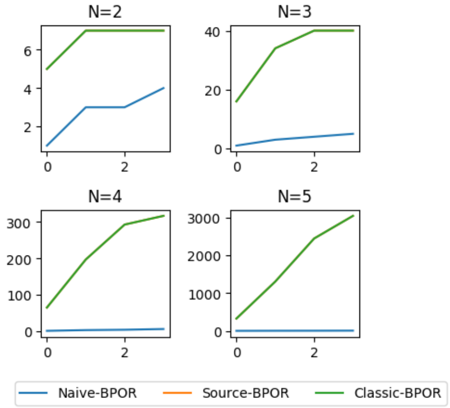

Figure: writer-N-readers bounded

Nidhugg-BPOR Evaluation

Evaluation of Nidhugg-BPOR on Synthetic Tests

Technique:	Naive-BPOR			Nidhugg-BPOR			Source-BPOR		
Bound:	0	1	2	0	1	2	0	1	2
account.c	1	1	4	6	27	42	6	27	42
lazy.c	1	1	4	6	27	42	6	27	42
micro.c	1	1	10	6	93	886	6	93	886
lastzero.c	1	2	5	252	2444	10614	252	2444	10614
lastzeromod.ll	1	1	6	64	290	651	64	290	651
indexer0.c	1	4	1	2	8	14	2	8	14
indexermod.c	1	1	5	120	1320	7920	120	1320	7920

Table: Traces for various bound limits

Evalution of BPOR on RCU

Read-Copy-Update (RCU): Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002.
Let's start with a small bound...

ver:	3.0						3.19						4.9.6					
method:	Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	3	0.2	NF	44	0.72	NF	2	0.32	NF	28	0.76	NF	2	0.61	NF	24	1.21	NF
-DFORCE_FAILURE_1	3	0.2	NF	44	0.72	NF	2	0.32	NF	28	0.76	NF	2	0.6	NF	24	1.21	NF
-DFORCE_FAILURE_3	3	0.2	NF	44	0.72	NF	2	0.32	NF	33	1.06	NF	2	0.61	NF	41	2.11	NF
-DFORCE_FAILURE_5	3	0.2	NF	44	0.71	NF	2	0.31	NF	18	0.55	NF	2	0.6	NF	16	0.93	NF
-DLIVENESS_CHECK_1	3	0.2	NF	44	0.72	NF	2	0.32	NF	28	0.74	NF	2	0.61	NF	24	1.19	NF
-DLIVENESS_CHECK_2	3	0.2	NF	52	0.84	NF	2	0.32	NF	28	0.73	NF	2	0.6	NF	24	1.2	NF
-DLIVENESS_CHECK_3	3	0.2	NF	44	0.71	NF	2	0.31	NF	28	0.75	NF	2	0.6	NF	24	1.19	NF

Table: RCU results for bound $b = 1$

Evalution of BPOR on RCU

Let's increase the bound...

ver:	3.0						3.19						4.9.6					
method:	Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	50	1.18	NF	5634	88.78	NF	10	0.49	NF	2083	60.48	NF	10	0.89	NF	2469	122.71	NF
-DFORCE_FAILURE_1	50	1.06	NF	275	4.2	F	10	0.49	NF	182	5.51	F	10	0.89	NF	300	15.42	F
-DFORCE_FAILURE_3	50	1.05	NF	1627	23.09	NF	15	0.72	NF	100000	0.0	NF	15	1.2	NF	100000	0.0	NF
-DFORCE_FAILURE_5	49	1.05	NF	4155	59.47	NF	9	0.45	NF	60	2.34	F	9	0.81	NF	60	3.92	F
-DLIVENESS_CHECK_1	48	1.04	NF	1493	21.19	NF	10	0.5	NF	517	10.66	NF	10	0.88	NF	404	13.58	NF
-DLIVENESS_CHECK_2	61	1.28	NF	2105	30.5	NF	10	0.5	NF	517	10.61	NF	10	0.88	NF	582	20.28	NF
-DLIVENESS_CHECK_3	49	1.04	NF	1788	24.98	NF	10	0.5	NF	655	14.04	NF	10	0.88	NF	506	17.32	NF

Table: RCU results for bound $b = 4$

Evalution of BPOR on RCU

What did we achieve?

ver:	3.0						3.19						4.9.6					
method:	Source-DPOR			Classic-BPOR			Source-DPOR			Classic-BPOR			Source-DPOR			Classic-BPOR		
	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound
-DFORCE_FAILURE_1	247	3.81	F	275	4.2	4	515	16.88	F	182	5.51	4	861	45.69	F	300	15.42	4
-DFORCE_FAILURE_3	2372	33.42	NF				17094	626.4	F	201	7.03	2	15349	883.98	F	258	14.24	2
-DFORCE_FAILURE_5	12426	178.8	NF				118	3.99	F	60	2.34	4	112	6.34	F	60	3.92	4

Table: Comparison between DPOR and BPOR

Equivalence of Source-BPOR with Nidhugg-BPOR

Equivalence Case1:

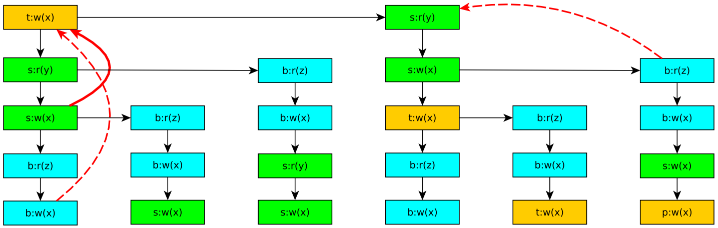

Figure: Source-BPOR and Nidhugg-BPOR equivalence Case 1

Equivalence of Source-BPOR with Nidhugg-BPOR

Equivalence Case 2:

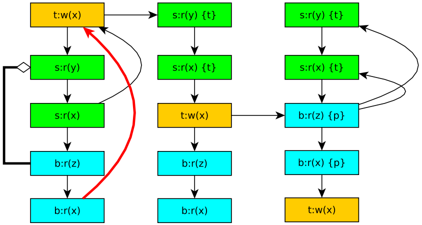

Figure: Source-BPOR and Nidhugg-BPOR equivalence Case 2

Discussion

Motivation

Some preemption-switches can be easily avoided. For example:

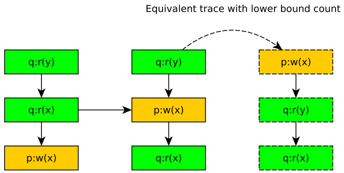

Figure: An example of avoidable preemption-switch

Alternating the General Form of BPOR

What if calculate something more than the preemption-bound?

```
Result: Explore the whole state space within the bound
Explore($\emptyset$);
Function Explore($S$)
    T = Sufficient_set($final(S)$) for all $t \in T$ do
        if $min\{B_v([S.t])\} \leq c$ then
            Explore($S.t$)
        end
    end
```

Algorithm 9: General form of the BPOR without branch addition

Applying the Graph Construction Algorithm

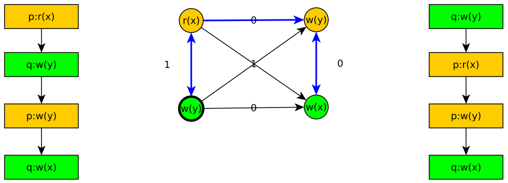

Figure: Graph example

Introducing Lazy-BPOR

```
**let** $G =: \emptyset$;
Explore($\langle\rangle$,$\emptyset$,$G$,$b$);
**Function** *Explore($E$,$Sleep$,$G$,$b$)*
  **if** $\exists p \in (enabled(s_{[E]})\backslash Sleep)$ *such that* $B_v(E.p) \leq b$ **then**
    backtrack(E) $:= p$ ;
    **while** $\exists p \in (backtrack(E)\backslash Sleep$ **do**
      **foreach** $e \in dom(E)$ *such that* $e \lesssim_{E.p} next_{[E]}(p)$ **do**
        **let** $E' = pre(E, e)$;
        **let** $u = notdep(e, E).p$;
        **if** $I_{E'}(u) \cap backtrack(E') = \emptyset$ **then**
          add some $q' \in I_{[E']}(u) to backtrack(E')$ ;
        **end**
      **end**
      **let** $Sleep' := \{q \in Sleep \mid E \vDash p \Diamond q\}$;
      **if** $p$ *creates a new block* **then**
        **let** $block = last\_block(E)$;
        **let** $G' =$ add_block($block$,$G$);
      **end**
      **if**
       $min\{Ham\_path(G')$ *which compensate with all happens-before relations of* $E\} \leq$
       $b$ **then**
        $Explore(E.p, Sleep, G', b)$ ;
        add $p$ to $Sleep$ ;
      **end**
    **end**
  **end**
```

Algorithm 10: Lazy-BPOR

Evaluation of Lazy-BPOR

Evaluation on Synthetic Tests:

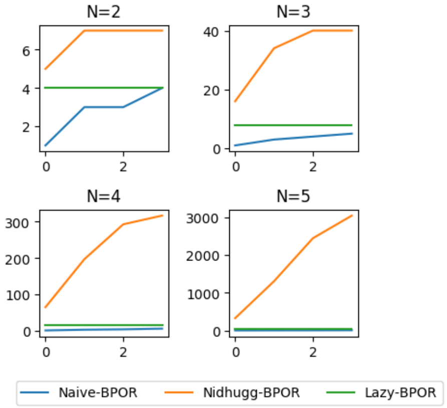

Figure: writer-N-readers bounded by the first estimation algorithm

Evaluation of Lazy-BPOR

Evaluation on Synthetic Tests:

Technique:	Naive-BPOR			Lazy-BPOR			Nidhugg-BPOR		
Bound:	0	1	2	0	1	2	0	1	2
account.c	1	1	4	6	6	6	6	27	42
lazy.c	1	1	4	6	6	6	6	27	42
micro.c	1	1	10	60	805	4362	6	93	886
lastzero.c	1	2	5	97	97	97	252	2444	10610
lastzeromod.ll	1	1	6	13	13	13	64	290	651
indexer0.c	1	4	1	4	8	8	2	8	14
indexermod.c	1	1	5	120	120	120	120	1320	7920

Table: Traces for the first estimation algorithm for various bound limits

Evaluation of Lazy-BPOR

Evaluation on RCU (Nidhugg-BPOR vs Lazy-BPOR):

ver.	3.0						3.19						4.3						4.7						4.9.6					
method	Nidhugg-BPOR			Lazy-BPOR			Nidhugg-BPOR			Lazy-BPOR			Nidhugg-BPOR			Lazy-BPOR			Nidhugg-BPOR			Lazy-BPOR			Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound
-ASSERT_0	2.65	183	3	2.79	104	2	2.96	106	3	4.06	73	2	5.39	128	3	8.57	85	2	5.28	118	3	6.28	75	2	5.91	128	3	9.44	85	2
-DFORCE_FAILURE_0	3.74	275	4	3.45	141	3	5.02	182	4	8.68	121	3	12.69	300	4	21.73	163	3	9.73	220	4	11.28	123	3	13.93	300	4	23.54	163	3
-DFORCE_FAILURE_1	0.35	6	1	0.35	4	1	0.54	5	1	0.52	3	0	0.75	5	1	0.71	3	0	0.91	5	1	0.87	3	0	0.95	5	1	0.9	3	0
-DFORCE_FAILURE_2			NF				6.49	201	2	54.62	200	1	12.11	258	2	103.89	233	1	12.59	258	2	107.1	233	1	12.84	258	2	111.37	233	1
-DFORCE_FAILURE_3	0.91	47	2	1.38	51	2	1.78	41	2	2.1	24	1	1.89	21	2	1.79	14	1	2.3	24	2	2.27	17	1	2.39	24	2	2.34	17	1
-DFORCE_FAILURE_4			NF				2.26	60	4	3.58	52	3	3.12	60	4	5.26	52	3	3.47	60	4	5.66	52	3	3.61	60	4	5.92	52	3
-DFORCE_FAILURE_5	0.95	1	0	0.94	1	0	2.74	2	0	2.77	2	0	4.47	2	0	4.33	2	0	8.7	2	0	8.45	2	0	8.73	2	0	8.56	2	0

Table: Comparison between BPOR and Lazy-BPOR

Evaluation of Lazy-BPOR

The Big Picture

Evaluation on RCU (Nidhugg-BPOR vs Lazy-BPOR):

ver.	4.7						4.9.6					
method	Nidhugg-BPOR			Lazy-BPOR			Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound	time	traces	bound	time	traces	bound
-ASSERT_0	5.28	118	3	6.28	75	2	5.91	128	3	9.44	85	2
-DFORCE_FAILURE_0	9.73	220	4	11.28	123	3	13.93	300	4	23.54	163	3
-DFORCE_FAILURE_1	0.91	5	1	0.87	3	0	0.95	5	1	0.9	3	0
-DFORCE_FAILURE_2	12.59	258	2	107.1	233	1	12.84	258	2	111.37	233	1
-DFORCE_FAILURE_3	2.3	24	2	2.27	17	1	2.39	24	2	2.34	17	1
-DFORCE_FAILURE_4	3.47	60	4	5.66	52	3	3.61	60	4	5.92	52	3
-DFORCE_FAILURE_5	8.7	2	0	8.45	2	0	8.73	2	0	8.56	2	0

Table: Comparison between BPOR and Lazy-BPOR

Conclusion:

- It is possible to explore a preemption-bounded state space without the addition of conservative branches.

Conclusion:

- It is possible to explore a preemption-bounded state space without the addition of conservative branches.
- It provides an upper bound for the number of traces explored in BPOR no matter the bound. In fact the number of traces explored by Lazy-BPOR at worst case equal to the number of traces explored by the unbounded DPOR. This is true since no conservative branches are added.

Conclusion:

- It is possible to explore a preemption-bounded state space without the addition of conservative branches.
- It provides an upper bound for the number of traces explored in BPOR no matter the bound. In fact the number of traces explored by Lazy-BPOR at worst case equal to the number of traces explored by the unbounded DPOR. This is true since no conservative branches are added.
- The most important is that it provides a reduction of the preemption-bounded search to a well known graph problem where many heuristics can be applied in order to expedite the calculation of the minimum hamiltonian path.

Bibliography

- Abdulla P, Aronis S., Jonsson B., Sagonas K. (2014) Optimal Dynamic Partial Order Reduction
- Coons K., Musuvathi M, McKinley K., (2013) Bounded Partial-Order Reduction
- Kokologiannakis M, Sagonas K., (2017) Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy-Update (Tree RCU)
- Thomson P., Donaldson A., Betts A., (2016) Testing Using Controlled Schedulers: An Empirical Study
- Flanagan C., Godefroid P. (2005) Dynamic Partial-Order Reduction for Model Checking Software

Thank you for your attention!

Nidhugg-BPOR (Detailed)

```
Explore($\langle\rangle$,$\emptyset$,$b$);
Function Explore($E$,$Sleep$,$b$)
    if $\exists p \in ((enabled(s_{[E]})\backslash Sleep)$ and $B_v(E.p) <= b$ then
        backtrack(E) $:= p$ ;
        while $\exists p \in (backtrack(E)\backslash Sleep$ and $B_v(E.p) <= b$ do
            foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                let $CI = \{i \in I_{E'}(u) \mid i \to p\}$;
                if $CI \cap backtrack(E') = \emptyset$ then
                    if $CI \neq \emptyset$ then
                        add some $q' \in CI$ to $backtrack(E')$ ;
                    end
                    else
                        add some $q' \in I_{[E']}(u)$ to $backtrack(E')$ ;
                    end
                end
                let $E'' = pre\_block(e, E)$;
                let $u = notdep(e, E).p$;
                let $CI = \{i \in I_{E''}(u) \mid i \to p\}$;
                if $CI \cap backtrack(E') = \emptyset$ then
                    if $CI \neq \emptyset$ then
                        add some $q' \in CI$ to $backtrack(E')$ ;
                    end
                    else
                        add some $c(q') \in I_{[E'']}(u)$ to $backtrack(E'')$ ;
                    end
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \models p\Diamond q\}$ ;
            $Explore(E.p, Sleep)$ ;
            if $p$ is not conservative then
                add $p$ to $Sleep$ ;
            end
        end
```

Source-BPOR (Detailed)

```
Explore($\langle\rangle$,$\emptyset$,$b$);
Function Explore($E$,$Sleep$,$b$)
    if $\exists p \in ((enabled(s_{[E]}) \setminus Sleep)$ and $B_v(E.p) <= b$ then
        backtrack(E) := $p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep$ and $B_v(E.p) <= b$ do
            foreach $e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                if $I_{E'}(u) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(u)$ to $backtrack(E')$ ;
                end
                let $E'' = pre\_block(e, E)$;
                let $u = notdep(e, E).p$;
                let $CI = \{i \in I_{E''}(u) \mid i \rightarrow p\}$;
                if $CI \cap backtrack(E') = \emptyset$ then
                    if $CI \neq \emptyset$ then
                        add some $q' \in CI$ to $backtrack(E')$ ;
                    end
                    else
                        add some $c(q') \in I_{[E'']}(u)$ to $backtrack(E'')$ ;
                    end
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \vDash p \Diamond q\}$ ;
            $Explore(E.p, Sleep)$ ;
            if $p$ is not conservative then
                add $p$ to $Sleep$ ;
            end
        end
    end
```

Algorithm 12: Source-BPOR

Classic-BPOR

```
Function Explore($E$)
    let $s := last(E)$;
    for all process $p$ do
        for all process $q \neq p$ do
            if $\exists i = max(\{i \in dom(E) \mid E_i$ is dependent and may be co-enabled
             with $next(s, p)$ and $E_i.tid = q\}$ then
                if $p \in enabled(pre(E, i)))$ then
                    add $p$ to $backtrack(pre(E, i))$ ;
                else
                    add $enabled(pre(E, i))$ to $backtrack(pre(E, i))$ ;
                end
                if $j = max(\{j \in dom(E) \mid j = 0$ or $S_{j-1}.tid \neq S_j.tid$ and $j < i\})$
                 then
                    if $p \in enabled(pre(E, i)))$ then
                        add $p$ to $backtrack(pre(E, i))$ ;
                    else
                        add $enabled(pre(E, i))$ to $backtrack(pre(E, i))$ ;
                    end
            end
        end
    end
    if $p \in enabled(s)$ then
        add $p$ to $backtrack(s)$ ;
    end
    else
        add any $u \in enabled(s)$ to $backtrack(s)$ ;
    end
    let $visited = \emptyset$;
    while $\exists u \in (enabled(s) \cap backtrack(s) \setminus visited)$ do
        add $u$ to $visited$ ;
        if $(B_v(S.next(s, u)) \leq c)$ then
            Explore($S.next(s, u)$) ;
    end
```

Algorithm 13: BPOR