Bounding Techniques for Dynamic Partial Order Reduction

Σαχίνογλου Γιάννης

ΣΗΜΜΥ - ΕΜΠ

03112089

Περίληψη

Θέμα εργασίας

Concurrent Computing and Problems

Concurrent Computing: Concurrent computing 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

A simple example:

```
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

Testing, Model Checking, and Verification

- Testing: For some given inputs check whether the output is correct.
- Verification: Prove formally that the output is correct.
- Model Checking: Explore all the possible states the system can be.

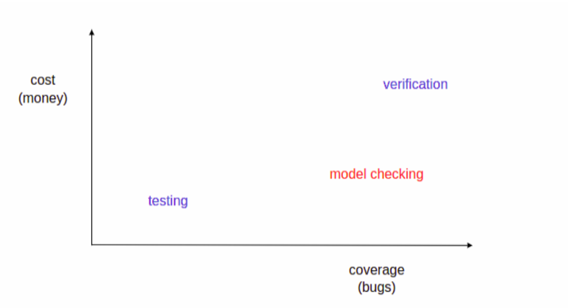

Figure: Comparing Testing, Model Checking and Verification

The Idea of Interleaving

- We need to model our state space!
- An Interleaving represents a scheduling of the concurrent program.
- In order to find an error of a concurrent program, one must examine every possible interleaving BUT leads to state explosion.

Stateless Model Checking and Partial Order Reduction

Partial order reduction aims to reduce the number of interleavings explored by eliminating the exploration of equivalent interleavings.
For example:

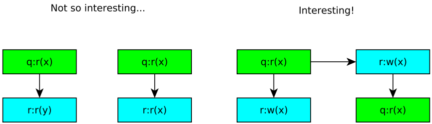

Figure: Examples of Interleavings

Stateless Model Checking and Partial Order Reduction

- Static Partial Order Reduction: Dependencies are tracked before execution.
- Dynamic Partial Order Reduction: Dependencies are observed during runtime.

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.
- Preemption Bounded and Delay Bounded exploration.
- Many of the concurrency bugs can be tracked even when the bound limit is set to be small.

We need to introduce some basic ideas and notation!

Vector Clocks

1. Each process experiencing an internal event, it increments its own logical clock in the vector by one.
2. Each time a process receives a message or performs an action on a shared variable, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message or the maximum value of all processes that share the same shared variable.

Useful Notation

Event Dependencies

Definition 1 (happens-before assignment)

A happens-before assignment, which assigns a unique happens-before relation $\rightarrow E$ to any execution sequence E, is valid if it satisfies the following properties for all execution sequences E.

1. $\rightarrow_E$ is a partial order on $dom(E)$, which is included in $<_E$. In other words every scheduling is part of the set of all possible partial order of the program.
2. The execution steps of each process are totally ordered, i.e. $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$ whenever $\langle p, i+1\rangle \in dom(E)$.
3. If E' is a prefix of E then $\rightarrow_E$ and $\rightarrow_{E'}$ are the same on $dom(E')$.

Event Dependencies

4. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence which has exactly the same "happens-before" relation $\rightarrow_{E'}$ as $\rightarrow_E$. This means that the relation $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same "happens-before" relation. We use $E \simeq E'$ to denote that E and E' are linearizations of the same "happens-before" relation, and $[E] \simeq$ to denote the equivalence class of E.
5. If $E \simeq E'$ then $s_{[E]} = s_{[E']}$ (i.e. two equivalent traces will lead to the same state).
6. For any sequences E, E' and w, such that $E.w$ is an execution sequence, we have $E \simeq E'$ if and only if $E.w \simeq' E'.w$.

Definition 2 (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

Sufficient Sets: Persistent Sets

Definition 3 (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:

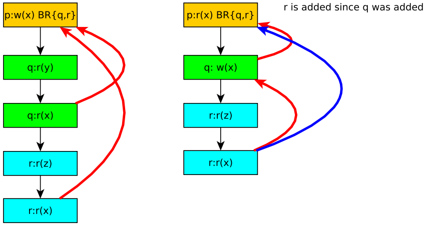

Figure: Construction of persistent sets

Sufficient Sets: Source Sets

Definition 4 ($dom(E)$)

The set of events-transitions happening during the scheduling of E.

Definition 5 (Initials after an execution sequence $E.w$, $I_{[E]}(w)$)

For an execution sequence $E.w$, let $I_{[E]}(w)$ denote the set of processes that perform events e in $dom_{[E]}(w)$ that have no "happens-before" predecessors in $dom_{[E]}(w)$. More formally, $p \in I_{[E]}(w)$ if $p \in w$ and there is no other event $e \in dom_{[E]}(w)$ with $e \rightarrow_{E.w} next_{[E]}(p)$.

By relaxing the definition of Initials we can get the definition of Weak Initials, WI.

Definition 6 (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)

For an execution sequence $E.w$, let $WI_{[E]}(w)$ denote the union of $I_{[E]}(w)$ and the set of processes that perform events p such that $p \in enabled(s_{[E]})$.

Sufficient Sets: Source Sets

Sufficient Sets: Source Sets

Definition 7 (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 $WI_{[E]}(w) \cap P = \emptyset$.

Souce Sets

An example:

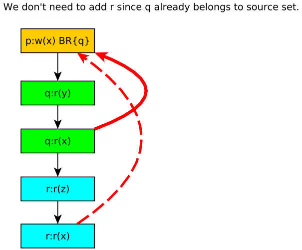

Figure: Construction of Source Sets

Further Optimizations: Sleep Sets

The idea behind Sleep Set Optimization:

- 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

Sleeps sets in action (Using Persistent Sets):

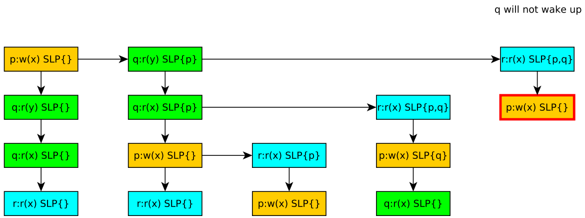

Figure: Example of Sleep Set Optimization

Bounded Dynamic Partial Order Reduction General Form

Given a bound evaluation function B_v and a bound c:

Result: Explore the whole statespace
Explore($\emptyset$);
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 2: Bounded-DPOR

Preemption Bounded Search

Definition 8 (Preemption bound)

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

Definition 9 ($ext(s, t)$)

Given a state $s = final(E)$ and a transition $t \in enabled(s)$, $ext(s, t)$ returns the unique sequence of transitions β from s such that

1. $\forall i \in dom(\beta) : \beta_i.tid = t.tid$
2. $t.tid \notin enabled(final(E.\beta))$

Preemption Bounded Persistent Sets

Definition 10 (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 $ext(s, t) \leftrightarrow next(final(E.a), last(a).tid)$

Source-DPOR

```
Explore(⟨⟩,∅);
Function Explore(E,Sleep)
  if ∃p ∈ (enabled(s_[E])\Sleep) then
    backtrack(E) := p ;
    while ∃p ∈ (backtrack(E)\Sleep) do
      foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
        let E′ = pre(E, e);
        let u = notdep(e, E).p;
        if I_{E′}(u) ∩ backtrack(E′) = ∅ then
          add some q′ ∈ I_[E′](u) to backtrack(E′) ;
        end
      end
      let Sleep′ := {q ∈ Sleep | E ⊨ p♢q};
      Explore(E.p, Sleep′) ;
      add p to Sleep ;
    end
  end
```

Algorithm 3: Source-DPOR Algorithm

DPOR using Clock Vectors (Classic-DPOR)

Function *Explore*(E,C)
  let $s := last(E)$;
  for *all process* 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* $i \nleq C(p)(proc(E_i))\}$ **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
  if $\exists p \in enabled(s)$ **then**
    $backtrack(s) := p$;
    let $done = \emptyset$;
    while $\exists p \in (backtrack(s) \backslash done)$ **do**
      add p to $done$;
      let $t = next(s,p)$;
      let $E' = E.t$;
      let $cu = max\{C(i) \mid i \in 1..|S|$ and E_i dependent with $t\}$;
      let $cu2 = cu[p := |E'|]$;
      let $C' = C[p := cu2, |E'| := cu2]$;
      $Explore(E', C')$;
    end
  end

Algorithm 4: DPOR using Clock Vectors (Classic-DPOR)

Source-DPOR vs Classic-DPOR

Similarities:

1. Consist of the same phases i.e., race detection and exploration
2. Both rely on Vector Clocks.

Differences:

1. Classic-DPOR "eager" i.e., adds more dependencies before scheduling.
2. Source-DPOR "lazy" i.e., adds branches after scheduling and thus avoids redundant additions.

Nidhugg-DPOR

```
Explore(⟨⟩,∅);
Function Explore(E,Sleep)
    if ∃p ∈ (enabled(s_[E]))\Sleep then
        backtrack(E) := p ;
        while ∃p ∈ (backtrack(E)\Sleep) do
            foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
                let E' = pre(E, e);
                let u = notdep(e, E).p;
                let CI = {i ∈ I_{E'}(u) | i → p};
                if CI ∩ backtrack(E') = ∅ then
                    if CI ≠ ∅ then
                        add some q' ∈ CI to backtrack(E') ;
                    end
                    else
                        add some q' I_{E'}(u) to backtrack(E')
                    end
                end
            end
            let Sleep' := {q ∈ Sleep | E ⊨ p◇q} ;
            Explore(E.p, Sleep) ;
            add p to Sleep ;
        end
    end
```

Algorithm 5: 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.

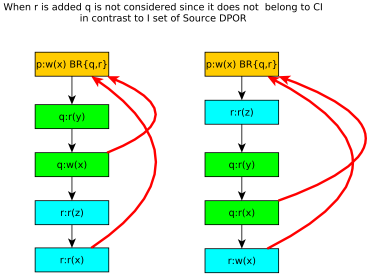

Figure: Construction of persistent sets in Nidhugg when there is a write process

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.

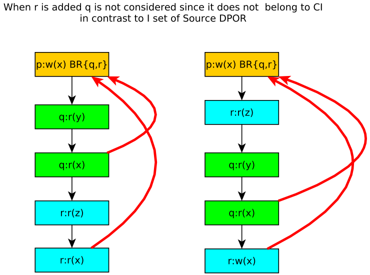

Figure: Construction of persistent sets in Nidhugg when both are read processes

Naive-BPOR

```
Explore(⟨⟩,∅,b);
Function Explore(E,Sleep,b)
    if ∃p ∈ (enabled(s_[E])\Sleep) such that B_v(E.p) ≤ b) then
        backtrack(E) := p ;
        while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) ≤ b do
            foreach e ∈ dom(E) such that e ≾_{E.p} next_[E](p) do
                let E' = pre(E, e);
                let u = notdep(e, E).p;
                if I_{E'}(u) ∩ backtrack(E') = ∅ then
                    add some q' ∈ I_[E'](u)tobacktrack(E') ;
                end
            end
            let Sleep' := {q ∈ Sleep | E ⊨ p◇q};
            Explore(E.p, Sleep, b) ;
            add p to Sleep ;
        end
    end
```

Algorithm 6: Naive-BPOR

Example execution of Naive-BPOR

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

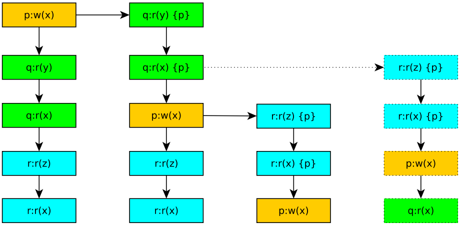

Figure: Naive-BPOR for bound=0

Classic-BPOR

```
Function Explore(E)
  let s := last(E);
  for all process p do
    for all process q ≠ p do
      if ∃i = max({i ∈ dom(E) | E_i is dependent and may be co-enabled
        with next(s, p) and E_i.tid = q} then
          if p ∈ 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 ∈ dom(E) | j = 0 or S_{j-1}.tid ≠ S_j.tid and j < i})
           then
            if p ∈ 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 ∈ enabled(s) then
    add p to backtrack(s) ;
  end
  else
    add any u ∈ enabled(s) to backtrack(s) ;
  end
  let visited = ∅;
  while ∃u ∈ (enabled(s) ∩ backtrack(s)\visited) do
    add u to visited ;
    if (B_v(S.next(s, u)) ≤ c) then
      Explore(S.next(s, u)) ;
  end
```

Algorithm 7: BPOR

Nidhugg-BPOR

```
Explore(⟨⟩,∅,b);
Function Explore(E,Sleep,b)
  if ∃p ∈ ((enabled(s_[E])\Sleep) and B_v(E.p) <= b then
    backtrack(E) := p ;
    while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) <= b do
      foreach e ∈ dom(E) such that e ≲_E.p next_[E](p) do
        let E' = pre(E, e);
        let u = notdep(e, E).p;
        let CI = {i ∈ I_E'(u) | i → p};
        if CI ∩ backtrack(E') = ∅ then
          if CI ≠ ∅ then
            add some q' ∈ CI to backtrack(E') ;
          end
          else
            add some q' ∈ I_[E'](u) to backtrack(E') ;
          end
        end
        let E'' = pre_block(e, E);
        let u = notdep(e, E).p;
        let CI = {i ∈ I_E''(u) | i → p};
        if CI ∩ backtrack(E') = ∅ then
          if CI ≠ ∅ then
            add some q' ∈ CI to backtrack(E') ;
          end
          else
            add some c(q') ∈ I_[E''](u) to backtrack(E'') ;
          end
        end
      end
      let Sleep' := {q ∈ Sleep | E ⊨ p◇q} ;
      Explore(E.p, Sleep) ;
      if p is not conservative then
        add p to Sleep ;
      end
    end
  end
```

Algorithm 8: Nidhugg-BPOR

The main question

Can we use source sets instead of persistent sets in order implement BPOR?

First approach

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

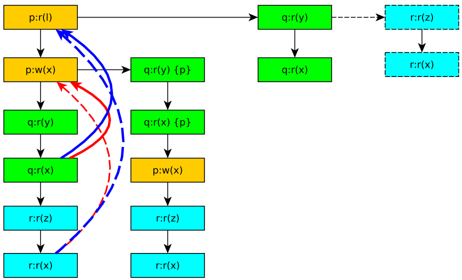

Figure: Following source sets for conservative branches

A Correct Approach

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

Source-BPOR

```
Explore(⟨⟩,∅,b);
Function Explore(E,Sleep,b)
    if ∃p ∈ ((enabled(s_[E])\Sleep) and B_v(E.p) <= b then
        backtrack(E) := p ;
        while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) <= b do
            foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
                let E′ = pre(E, e);
                let u = notdep(e, E).p;
                if I_{E′}(u) ∩ backtrack(E′) = ∅ then
                    add some q′ ∈ I_[E′](u) to backtrack(E′) ;
                end
                let E″ = pre_block(e, E);
                let u = notdep(e, E).p;
                let CI = {i ∈ I_{E″}(u) | i → p};
                if CI ∩ backtrack(E′) = ∅ then
                    if CI ≠ ∅ then
                        add some q′ ∈ CI to backtrack(E′) ;
                    end
                    else
                        add some c(q′) ∈ I_[E″](u) to backtrack(E″) ;
                    end
                end
            end
            let Sleep′ := {q ∈ Sleep | E ⊨ p◇q} ;
            Explore(E.p, Sleep) ;
            if p is not conservative then
                add p to Sleep ;
            end
        end
    end
```

Algorithm 9: 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.

Conservative Branches

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

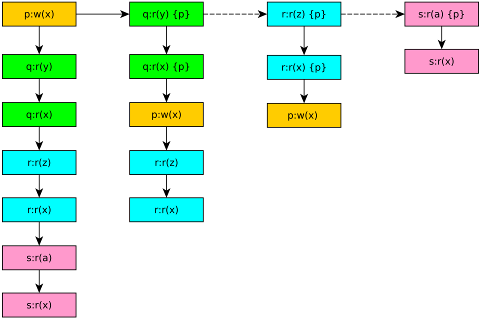

Figure: writer-3-readers explosion

Sleep Sets are no longer that useful:

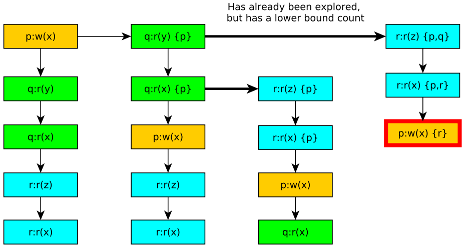

Figure: Sleep set contradiction

Concluding Remarks

The Preformance - Soundness Tradeoff

The Nidhugg Flow Chart

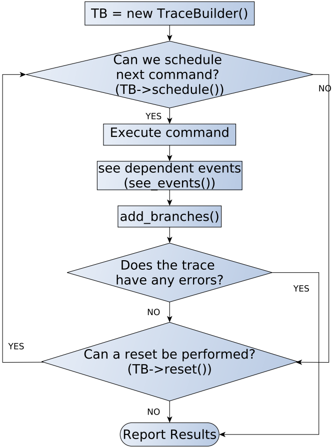

Figure: Nidhugg's Flow Chart

The implementation mainly is focused, as expected, on see_events() and add_branches()

Nidhugg-DPOR Evaluation

Evaluation of Nidhugg-DPOR on Synthetic Tests

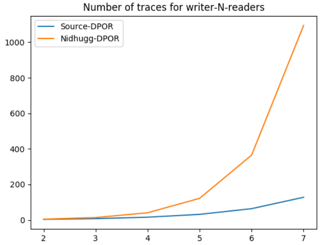

Figure: writer-N-readers

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

Evaluation of Nidhugg-BPOR on Synthetic Tests

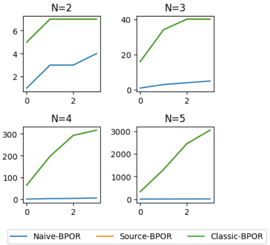

Figure: writer-N-readers bounded

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 with the bug

Equivalence of Source-BPOR with Nidhugg-BPOR

Equivalence Case1:

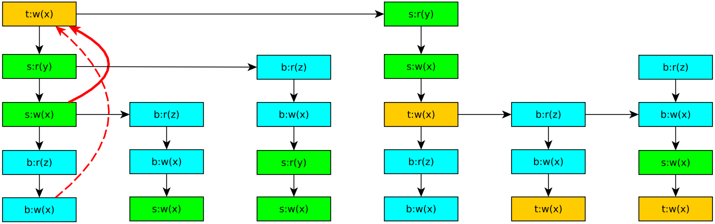

Figure: Source-BPOR and Nidhugg-BPOR equivalence Case 1

Equivalence of Source-BPOR with Nidhugg-BPOR

Equivalence Case2:

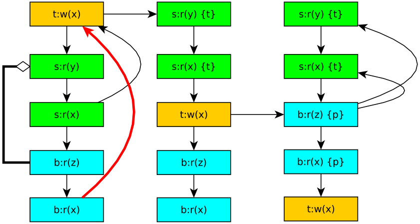

Figure: Source-BPOR and Nidhugg-BPOR equivalence Case 2

Motivation

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

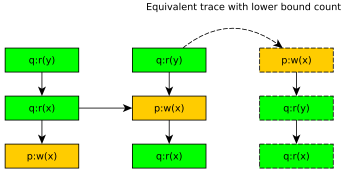

Figure: An example of avoidable preemption-switch

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 10: General form of the BPOR without branch addition

Construct a Block-Graph using Partial Order Reduction

```
Function AddBlock(block,graph)
    if previous block of the same thread was not blocked then
        increase the weigh of the edges coming from the previous block to 1 ;
    end
    for each thread t do
        list:= preceding blocks t;
        for l in reversed(list) do
            if l ↔ block then
                add edge from block to l with weight 0 ;
                if l is not last then
                    add edge from l to block with weight 1 ;
                end
                else
                    add edge from l to block with weight 0 ;
                end
            end
            if l → block then
                if l is not last then
                    add edge from l to block with weight 1 ;
                end
                else
                    add edge from l to block with weight 0 ;
                end
                break ;
            end
        end
    end
```

Algorithm 11: Adding a new block to the dependencies' graph

Applying the Graph construction

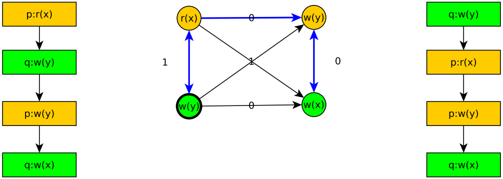

Figure: Graph example

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 12: Lazy-BPOR

Evaluation of Lazy-BPOR

Evaluation on Synthetic Tests:

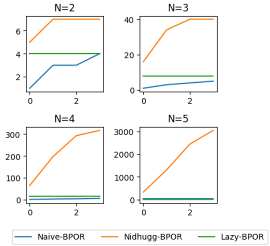

Figure: writer-N-readers bounded by the first estimation algorithm

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 (DPOR vs Lazy-BPOR):

ver:	3.0						3.19					
method:	DPOR			LBPOR			DPOR			LBPOR		
	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound
-DASSERT_0	246	3.83	F	104	2.79	2	512	17.67	F	73	4.06	2
-DFORCE_FAILURE_1	247	3.55	F	141	3.45	3	515	18.21	F	121	8.68	3
-DFORCE_FAILURE_2	4	0.34	F	4	0.35	1	3	0.55	F	3	0.52	0
-DFORCE_FAILURE_3			NF			NF	17094	636.25	F	200	54.62	1
-DFORCE_FAILURE_4	78	1.43	F	51	1.38	2	61	2.74	F	24	2.1	1
-DFORCE_FAILURE_5			NF			NF	118	4.1	F	52	3.58	3
-DFORCE_FAILURE_6	1	0.98	F	1	0.94	0	2	2.93	F	2	2.77	0

ver:	4.3						4.7					
method:	DPOR			LBPOR			DPOR			LBPOR		
	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound
-DASSERT_0	858	37.31	F	85	8.57	2	338	15.94	F	75	6.28	2
-DFORCE_FAILURE_1	861	37.8	F	163	21.73	3	341	15.9	F	123	11.28	3
-DFORCE_FAILURE_2	3	0.7	F	3	0.71	0	3	0.86	F	3	0.87	0
-DFORCE_FAILURE_3	15349	736.84	F	233	103.89	1	15349	714.01	F	233	107.1	1
-DFORCE_FAILURE_4	16	1.67	F	14	1.79	1	27	2.48	F	17	2.27	1
-DFORCE_FAILURE_5	112	5.12	F	52	5.26	3	112	5.51	F	52	5.66	3
-DFORCE_FAILURE_6	2	4.21	F	2	4.33	0	2	8.13	F	2	8.45	0

ver:	4.9.6					
method:	DPOR			LBPOR		
	traces	time	error	traces	time	bound
-DASSERT_0	858	40.42	F	85	9.44	2
-DFORCE_FAILURE_1	861	40.52	F	163	23.54	3
-DFORCE_FAILURE_2	3	0.88	F	3	0.9	0
-DFORCE_FAILURE_3	15349	793.75	F	233	111.37	1
-DFORCE_FAILURE_4	27	2.6	F	17	2.34	1
-DFORCE_FAILURE_5	112	5.8	F	52	5.92	3
-DFORCE_FAILURE_6	2	8.62	F	2	8.56	0

Table: Comparison between DPOR and Lazy-BPOR without the bug

Evaluation of Lazy-BPOR

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

ver.	3.0					
method	Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound
-ASSERT_0	2.65	183	3	2.79	104	2
-DFORCE_FAILURE_0	3.74	275	4	3.45	141	3
-DFORCE_FAILURE_1	0.35	6	1	0.35	4	1
-DFORCE_FAILURE_2			NF			
-DFORCE_FAILURE_3	0.91	47	2	1.38	51	2
-DFORCE_FAILURE_4			NF			
-DFORCE_FAILURE_5	0.95	1	0	0.94	1	0

ver.	3.19					
method	Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound
-ASSERT_0	2.96	106	3	4.06	73	2
-DFORCE_FAILURE_0	5.02	182	4	8.68	121	3
-DFORCE_FAILURE_1	0.54	5	1	0.52	3	0
-DFORCE_FAILURE_2	6.49	201	2	54.62	200	1
-DFORCE_FAILURE_3	1.78	41	2	2.1	24	1
-DFORCE_FAILURE_4	2.26	60	4	3.58	52	3
-DFORCE_FAILURE_5	2.74	2	0	2.77	2	0

ver.	4.3					
method	Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound
-ASSERT_0	5.39	128	3	8.57	85	2
-DFORCE_FAILURE_0	12.69	300	4	21.73	163	3
-DFORCE_FAILURE_1	0.75	5	1	0.71	3	0
-DFORCE_FAILURE_2	12.11	258	2	103.89	233	1
-DFORCE_FAILURE_3	1.89	21	2	1.79	14	1
-DFORCE_FAILURE_4	3.12	60	4	5.26	52	3
-DFORCE_FAILURE_5	4.47	2	0	4.33	2	0

ver.	4.7					
method	Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound
-ASSERT_0	5.28	118	3	6.28	75	2
-DFORCE_FAILURE_0	9.73	220	4	11.28	123	3
-DFORCE_FAILURE_1	0.91	5	1	0.87	3	0
-DFORCE_FAILURE_2	12.59	258	2	107.1	233	1
-DFORCE_FAILURE_3	2.3	24	2	2.27	17	1
-DFORCE_FAILURE_4	3.47	60	4	5.66	52	3
-DFORCE_FAILURE_5	8.7	2	0	8.45	2	0

ver.	4.9.6					
method	Nidhugg-BPOR			Lazy-BPOR		
	time	traces	bound	time	traces	bound
-ASSERT_0	5.91	128	3	9.44	85	2
-DFORCE_FAILURE_0	13.93	300	4	23.54	163	3
-DFORCE_FAILURE_1	0.95	5	1	0.9	3	0
-DFORCE_FAILURE_2	12.84	258	2	111.37	233	1
-DFORCE_FAILURE_3	2.39	24	2	2.34	17	1
-DFORCE_FAILURE_4	3.61	60	4	5.92	52	3
-DFORCE_FAILURE_5	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.
- 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 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.