Bounding Techniques for Dynamic Partial Order Reduction

Γιάννης Σαχίνογλου

ΣΗΜΜΥ - ΕΜΠ

03112089

Περίληψη

Θέμα εργασίας

Concurrent Programming

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 $\to E$ to any execution sequence E, is valid if it satisfies the following properties for all execution sequences E.

1. $\to_E$ 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. $\langle p, i\rangle \to_E \langle p, i+1\rangle$ whenever $\langle p, i+1\rangle \in dom(E)$.
3. If E' is a prefix of E then $\to_E$ and $\to_{E'}$ 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(∅);
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 $E \vdash t \diamondsuit w$ 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(∅);
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 $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ then
        $backtrack(E) := p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep)$ do
            foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ 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 $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$;
            $Explore(E.p, Sleep')$ ;
            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 $\exists p \in (backtrack(s) \setminus done)$ 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 $\exists p \in (enabled(s_{[E]}) \setminus Sleep$ then
        backtrack(E) $:= p$ ;
        while $\exists p \in (backtrack(E) \setminus Sleep)$ do
            foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                let $CI = \{i \in I_{E'}(u) \mid i \rightarrow p\}$;
                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 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($\langle\rangle$,$\emptyset$,$b$);
Function Explore($E$,$Sleep$,$b$)
    if $\exists p \in (enabled(s_{[E]}) \backslash Sleep)$ such that $B_v(E.p) \leq b)$ then
        $backtrack(E) := p$ ;
        while $\exists p \in (backtrack(E) \backslash Sleep$ and $B_v(E.p) \leq b$ do
            foreach $e \in dom(E)$ such that $e \lesssim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                if $I_{E'}(u) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(u) to backtrack(E')$ ;
                end
            end
            let $Sleep' := \{q \in Sleep \mid E \models p \diamondsuit q\}$;
            $Explore(E.p, Sleep, b)$ ;
            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 $q \neq p$ 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 7: BPOR

Nidhugg-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
                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 $Sleep' := \{q \in Sleep \mid E \models p \diamondsuit q\}$ ;
            $Explore(E.p, Sleep)$ ;
            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($\langle\rangle$,$\emptyset$,$b$);
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 $u = notdep(e, E).p$;
                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 $CI = \{i \in I_{E''}(u) \mid i \to p\}$;
                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
    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

Benchmark	Traces for Source-DPOR	Traces for Optimal-DPOR	Time for Source-DPOR	Time for Optimal-DPOR
lastzero 11	60073	7168	50m39.201s	23m32.843s
indexer 15	4096	4096	18m40.386s	12m45.39
readers 15	32768	32768	39m17.533s	52m43.585s

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