National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Preemption Bounding Techniques for Dynamic Partial Order Reduction

Diploma Thesis

IOANNIS-PETROS SACHINOGLOU

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Athens, 0000

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Preemption Bounding Techniques for Dynamic Partial Order Reduction

Diploma Thesis

IOANNIS-PETROS SACHINOGLOU

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Approved by the examining committee on the 00, 0000.

.....	.....	.....
Konstantinos Sagonas Associate Professor NTUA	Nikolaos S. Papaspyrou Associate Professor NTUA	Nectarios Koziris Professor NTUA

Athens, 0000

Ioannis-Petros Sachinoglou
Electrical and Computer Engineer

Abstract

Thorough verification and testing of concurrent programs is an important, but also challenging task. In order to verify a concurrent program one must examine all possible different interleavings the scheduler can produce. Stateless model checking with Dynamic Partial Order Reduction is a technique proposed to deal with state space explosion. Nevertheless, for larger programs the verification takes longer than the developers are willing to wait. In these cases, bounded search can be proved useful. Bounded search, in contrast to the DPOR, alleviates state-space explosion by pruning the executions that exceed a bound.

This thesis describes the implementation of the preemption bounded DPOR (BPOR) on Nidhugg, a bug finding tool which targets bugs caused by concurrency and relaxed memory consistency in concurrent programs. Specifically three bounding techniques were implemented: the Naive-BPOR, the BPOR, and the Source-BPOR. The three techniques were evaluated both in synthetic and in real world software. Specifically Read-Copy-Update mechanism of Linux Kernel was verified again. Moreover it is examined whether optimizations that have been suggested for the unbounded DPOR can improve the efficiency of BPOR.

Key words

Formal Verification, Stateless Model Checking, Systematic Concurrency Testing, RCU, Read-Copy-Update, Bounded Dynamic Partial Order Reduction,

Acknowledgements

First of all, I would like to thank my advisor, Kostis Sagonas, for his help and support during the preparation of this diploma thesis. Not only did he encourage me and support me all this time, but he also inspired me by his ceaseless enthusiasm and his avid interest in my work. I would also like to thank all members of Kostis Sagonas' team both in NTUA and Upsalla University and particulary Stavros Aronis for their vivid support.

Finally, I would like to extend my thanks to my family for encouraging me and supporting me all these years. Without them I would not be able to accomplish my goals.

Ioannis-Petros Sachinoglou,

Athens, 00, 0000

This thesis is also available as Technical Report CSD-SW-TR-1-16, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, 0000.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Tables

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

Moore's Law, named after Intel's co-founder Gordon Moore, states that the number of transistors that can be placed on an integrated circuit doubles roughly every two years. For decades, chipmakers have succeeded in shrinking chip geometries, allowing Moore's Law to remain on track and consumers to get their hands on even more powerful laptops, tablets, and smartphones. Software developers could just lay back and wait for the Moore's Law to take effect. However, constraints such as heat, clock speeds have largely stood still, and the incremental increase of the performance of each individual processor core impede the further acceleration of software execution. In order for developers to compensate with the demand of efficient software, programming paradigms such as concurrent programming have become a necessity. However, new challenges arise from concurrent programming since it is harder and more error-prone than its sequential counterpart. When programming with multiple execution threads many errors may occur due to the fact the many execution threads may access and edit the shared memory or require to execute lines of code excluding other threads.

More specifically, the typical problems with concurrency can be outlined as follows:

- Race condition: A strange interleaving of processes has an unintended effect.
- Deadlock: Two or more processes stop and wait for each other.
- Livelock: Two or more processes keep executing without making any progress.
- Resource starvation: Two or more processes are stuck in circular waiting for the resources

These problems are usually Heisenbugs [Musu08] – they can alter their behavior or completely disappear when one tries to isolate them – since they go hand in hand with the order of execution of the processes involved.

1.1 Testing and Verification of Concurrent Programs

Testing and verifying the correctness of a concurrent program can be proved a demanding task. A technique used for the systematic exploration of a program's state space is model checking [Wikib]. Model checking is a method for formally verifying concurrent systems through specifications about the system expressed as temporal logic formulas and efficient algorithms that can traverse the model defined by the system and check whether the specifications hold. The major problem model checking tools have to face is the combinatorial explosion of the state space since a vast number of global states have to be captured and stored. Many techniques have been proposed in order to tackle this problem. Stateless model checking, for example, avoids storing global states. This technique has been implemented in tools such as Verisoft [Gode97, Gode05], CHESS [Musu08], Concuerror [Chri13], Nidhugg [Abdu15] and [Koko17a]. The observation that two interleavings are equivalent

if one can be obtained from the other by swapping adjacent, independent execution steps is the core of the partial order reduction [Valm91, Pele93, Gode96, Clar99, Abdu17b] techniques used by many of these tools. Dynamic Partial Order Reduction (DPOR) techniques capture dependencies between operations of concurrent threads while the program is running [Flan05, Abdu17b]. The exploration begins with an arbitrary interleaving whose steps are then used to identify operations and points where alternative interleavings need to be explored in order to capture all program behaviors. Another approach is the bounded model checking [Bier03] where the finite state machine is unrolled for a fixed number of steps and the specifications are checked within these steps. Bounded model checking can be combined with the partial order reduction for modeling executions [Algl13] and was effectively implemented in tools such as CBMC [Clar04],Nidhugg [Abdu14, Koko17a]. Unfortunately all these techniques still have to deal with the problem of the state space explosion. In order to deal with this problem further bounding of the exploration is required. Many different bounding techniques have been examined [Thom16] such as preemption bounding, delay bounding, a controlled random scheduler, and probabilistic concurrency testing (PCT).

1.2 Aim of this Thesis

The purpose of this thesis is to:

- Implement a preemption bounding technique [Coon13] for Nidhugg.
- Examine whether the techniques introduced in [Abdu14] for optimal unbounded dynamic partial order reduction can be used for the implementation of bounded partial order reduction.
- Confirm or disapprove the capability of bounded dynamic partial order reduction to track errors faster than unbounded partial order reduction.
- Examine whether the empirical observation that most concurrency errors can manifest themselves in a small number of preemptions is correct [Musu07].
- Explore alternative algorithms that can perform bounded partial order reduction.

1.3 Overview

In Chapter 2 the theoretical background utilized in Nidhugg for both unbounded and bounded DPOR is given. In Chapters 3 4 the unbounded DPOR and bounded DPOR algorithms implemented and evaluated are described in further detail. In Chapter 5 the technical details of the implementations are discussed. The evaluation of the each algorithm is given in Chapter 6 where the algorithms are tested using both synthetic tests and real-world software (RCU). In Chapter 7 we present and evaluate an alternative approach to the bounding problem. Finally, in Chapter 8 we summarize the previous chapters and the conclusions we drew from this thesis, and present some possible extensions to our work.

Chapter 2

Background

2.1 Concurrent Programming

Concurrent computing, which is implemented by concurrent programming paradigm, 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). This is a property of a system—this may be an individual program, a computer, or a network—and there is a separate execution point or "thread of control" for each computation ("process"). A concurrent system is one where a computation can advance without waiting for all other computations to complete. The main challenge in designing concurrent programs is concurrency control: ensuring the correct sequencing of the interactions or communications between different computational executions, and coordinating access to resources that are shared among executions. Potential problems include race conditions, deadlocks, livelocks and resource starvation. The scheduler is usually responsible for running a thread. Due to this scheduling non-determinism the programmer cannot always be aware of which thread will be scheduled next.

An important aspect of a concurrent program is the notion of the set of interleavings which is the set of all the execution paths a program can follow. Intuitively, if we imagine a process as a (possibly infinite) sequence/trace of statements (e.g. obtained by loop unfolding), then the set of possible interleavings of several processes consists of all possible sequences of statements of any of these processes.

As it can be inferred, debugging this kind of programs can be proved extremely challenging. The challenge mainly emerges from the fact that it is not always clear which thread command will be executed. Moreover the error may not always occur during debugging since there may be only a limited number of interleavings that produce an error.

2.2 Concurrency Errors

At this point, it is important to introduce the notion of concurrency error and how explain how it differs from any other error.

Definition 2.1. (Concurrency Error) Concurrency error is an error caused by the scheduler's non-determinism.

An example of a program containing a concurrency error is shown in the Listing 2.2. In this program variable x equals 1 in the beginning of the program. However, if thread $zero$ is scheduled before $divider$ a division by zero will take place. On the other hand, a division by x cannot be considered a concurrency error in other cases such as the one shown in Listing 2.2 where a division by zero occurs without the scheduler's intervention.

```
1
2  void *divider(void* arg){
3      int x = 0;
4      return 42/x;
5  }
```

Listing 2.1: Example of non-concurrency error

```
1  volatile int  x = 1;
2  void *divider(){
3      return 42/x;
4  }
5
6  void *zero(){
7      x = 0;
8  }
```

Listing 2.2: Example of concurrency error

2.3 Testing, Model Checking, and Verification

Dynamic software model checking is a form of systematic testing which is applicable to industrial-size software and consists of adapting model checking. Many tools have been developed over the last decades that use this paradigm which aim to test concurrent and data-driven software. Model checking is more computationally expensive than traditional software testing since it provides better coverage. However, it is cheaper than more general forms of verification such as interactive theorem proving, which provides more extended verification guarantees, since it can be automated in a higher level. Hence, dynamic software model checking offers an attractive practical trade-off between testing and formal verification.

The graph shown in Figure 2.1 [Gode15] greatly demonstrates the differences between testing, model checking and verification.

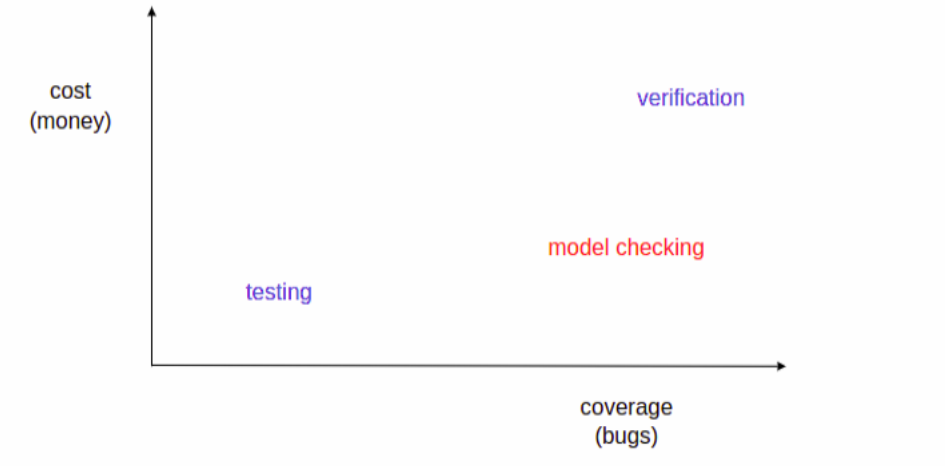

Figure 2.1: Comparing Testing, Model Checking and Verification

2.4 Stateless Model Checking and Partial Order Reduction

In order to find an error of a concurrent program, one must examine every possible interleaving, (i.e., all the different ways that a program can be executed) this program can produce. Usually the error would occur only under some interleaving the programmer did not take into consideration, making its detection extremely difficult. Stateless model checking is based on the idea of driving the program along all these possible interleavings. However, this approach suffers from state explosion, i.e., the number of all possible interleavings grows exponentially with the size of the program and the number of threads. Several approaches to this problem have been proposed in order to deal with this challenge such as partial order reduction [Gode96] and bounding techniques [Coon13].

Partial order reduction aims to reduce the number of interleavings explored by eliminating the exploration of equivalent interleavings. Each interleaving can be represented by a trace. These equivalent traces are produced by the inversion of independent events which do not affect the results of the program. For example, the scheduling of two threads that read a local variable can be inverted since the result of the operation is not affected by the order under which these operations occurs. There are two ways that a partial order reduction algorithm can be implemented. The first is a static partial order reduction algorithm [Kurs98] where the dependencies between two threads are tracked before the execution of the concurrent program. The second is the Dynamic partial order reduction (DPOR) [Flan05] which observes the program's dependencies of runtime. It is important to notice that the size of the state space still grows exponentially.

For larger programs DPOR often runs longer than developers are willing to wait. In these cases, bounding techniques can be proved useful. Bounded techniques, in contrast to DPOR, alleviates state-space explosion by pruning the executions that exceed a bound [Thom16]. There have been proposed many bounded techniques such as preemption bounded exploration [Coon13] or delay bounded exploration [Emmi10]. All bounded search techniques are based on the notion that many of the concurrency bugs can be tracked even when the bound limit is set to be small, thus the time required for a bug to be found is significantly smaller.

When it comes to bounded search, new challenges arise [Coon13] since more dependencies, related to the bound limit, are introduced. Many redundant traces must be explored in order to compensate with the bound since a bounded search algorithm cannot be aware if an equivalent trace with greater bound count has already been explored.

It is important to notice that some bounded techniques can be regarded as testing, in the sense that explore only a subset of total state space, while others can be considered as verification, in the sense that can verify that no error can occur in the given bound.

2.5 Vector Clocks

A vector clock is an algorithm for generating a partial ordering of events in a distributed or concurrent system and detecting causality violations. Just as in Lamport timestamps [Lamp78], interprocess messages contain the state of the sending process's logical clock. A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process, with the following rules for clock updates:

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.

An example execution of the algorithm is shown in Figure 2.2 where both the source code and an explored trace are given. As we can easily notice for each command of thread <0> (main thread) the clock of the main thread increases. The thread <0.0> starts to run its clock for the thread <0> is 8 since that is the moment when the thread was spawned. When the value of y is read the clock for <0> increases again so it corresponds with the y=1 event. When the first thread is scheduled again then its clock for the <0.0> is 7 since `pthread_join()` command takes place.

```
volatile int x = 0, y = 0, c = 0;
void *thr1(void *arg){
        y = 1;
        if(!x){
                c = 1;
        }
        return NULL;
}
int main(int argc, char *argv[]){
        pthread_t t;
        pthread_create(&t,NULL,thr1,NULL);
        x = 1;
        if(!y){
                c = 0;
        }
        pthread_join(t,NULL);
        return 0;
}
```

```
(<0>,1-4)       [1]
(<0>,5)         [5]
(<0>,6)         [6]
(<0>,7-8)       [7]
(<0>,9)         [9]
(<0>,10-12)     [10]
(<0>,13-14)     [13]
(<0>,15)        [15]
    (<0.0>,1)   [8, 0, 1]
    (<0.0>,2)   [8, 0, 2]
    (<0.0>,3)   [10, 0, 3]
    (<0.0>,4-7) [10, 0, 4]
(<0>,16-17)     [16, 0, 7]
```

Figure 2.2: Clock example

Every algorithm that is presented in this thesis is based on vector clocks.

2.6 Notation

Before delving in the problem of dynamic partial order reduction it is crucial to explain the notation that will be used. An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from the initial state which we denote as s_0. Since each execution step is deterministic, an execution sequence E is uniquely characterized by the sequence of processes that perform steps in E. For instance, $p.p.q$ denotes the execution sequence where first p performs two steps, followed by a step of q. The sequence of processes that perform steps in E also uniquely determine the (global) state of the system after E, which is denoted $s_{[E]}$. For a state s, let $enabled(s)$ denote the set of processes p that are enabled in s (i.e., for which execute $p(s)$ is defined). We use . to denote concatenation of sequences of processes. Thus, if p is not blocked after E, then $E.p$ is an execution sequence. An event of E is a particular occurrence of a process in E. We use $\langle p, i\rangle$ to denote the ith event of process p in the execution sequence E. In other words, the event $\langle p, i\rangle$ is the ith execution step of process p in the execution sequence E. We use $dom(E)$ to denote the set of events $\langle p, i\rangle$ which are in E, i.e., $\langle p, i\rangle \in dom(E)$ iff E contains at least i steps of p. We will use $e, e', \ldots$, to range over events. We use $proc(e)$ to denote the process p of an event $e = \langle p, i\rangle$. If $E.w$ is an execution sequence, obtained by concatenating E and w, then $dom_{[E]}(w)$ denotes $dom(E.w)$ $dom(E)$, i.e. the events in $E.w$ which are in w. As a special case, we use $next_{[E]}(p)$ to denote $dom_{[E]}(p)$. We use $<_E$ to

denote the total order between events in E, i.e. $e <_E e'$ denotes that e occurs before e' in E. We use $E' \leq E$ to denote that the sequence E' is a prefix of the sequence E.

2.7 Event Dependencies

One of the most important concepts when we have to deal with an algorithm that searches the whole state space of the different schedulings is the happens-before relation in an execution sequence. Usually this relation is denoted with $\rightarrow$ symbol. For example, if the relation $\rightarrow$ for two events e, e' in $dom(E)$ holds true then the event e happens-before e'. This relation usually appears in the message exchange, when e is the message transmission and e' is the event when the message is received. For the context of Nidhugg $e \rightarrow e'$ would hold true when at least one of the two events is a write operation on the same shared variable. It is fathomable that any DPOR algorithm should be able to assign this happens-before relations. In practice, the happens-before assignment is implemented with the use of vector clocks.

Definition 2.2. (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 $dom(E')$.
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 $E.w \simeq' E'.w$.

The first six properties should be obvious for any reasonable happens-before relation. The only non-obvious is the last. Intuitively, if the next step of p happens before the next step of r after the sequence E, then the step of p still happens before the step of r even when some step of another process, which is not dependent with p, is inserted between p and r. This property holds in any reasonable computation model that we can think of. As examples, one situation is when p and q read a shared variable that is written by r. Another situation is that p sends a message that is received by r. If an intervening process q is independent with p, it cannot affect this message, and so r still receives the same message. Properties 4 and 5 together imply, as a special case, that if e and e' are two consecutive events in E with $e \not\rightarrow_E e'$, then they can be swapped and the (global) state after the two events remains the same.

2.8 Independence and Races

We now define independence between events of a computation. If $E.p$ and $E.w$ are both execution sequences, then $E \models p \diamondsuit w$ denotes that $E.p.w$ is an execution sequence such

that $next_{[E]}(p) \not\rightarrow_{E.p.w} e$ for any $e \in dom([E.p])(w)$. In other words, $E \models p \diamondsuit w$ states that the next event of p would not "happen before" any event in w in the execution sequence $E.p.w$. Intuitively, it means that p is independent with w after E. In the special case when w contains only one process q, then $E \models p \diamondsuit q$ denotes that the next steps of p and q are independent after E. We use $E' \models p \diamondsuit w$ to denote that $E \not\models p \diamondsuit w$ does not hold.
For a sequence w and $p \in w$, let $w \backslash p$ denote the sequence w with its first occurrence of p removed, and let $w \uparrow p$ denote the prefix of w up to but not including the first occurrence of p. For an execution sequence E and an event $e \in dom(E)$, let $pre(E, e)$ denote the prefix of E up to, but not including, the event e. For an execution sequence E and an event $e \in E$, let $notdep(e, E)$ be the sub-sequence of E consisting of the events that occur after e but do not "happen after" e (i.e., the events e' that occur after e such that $e \not\rightarrow_E e'$).
A central concept in most DPOR algorithms is that of a race. Intuitively, two events, e and e' in an execution sequence E, where e occurs before e' in E, are in a race if

- e happens-before e' in E, and
- e and e' are "concurrent", i.e., there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent.

Formally, let $e \lessdot_E e'$ denote that $proc(e) \neq proc(e')$, that $e \rightarrow_E e'$, and that there is no event $e'' \in dom(E)$, different from e' and e, such that $e \rightarrow_E e'' \rightarrow_E e'$.
Whenever a DPOR algorithm detects a race, then it will check whether the events in the race can be executed in the reverse order. Since the events are related by the happens-before relation, this may lead to a different global state: therefore the algorithm must try to explore a corresponding execution sequence. Let $e \lesssim_E e'$ denote that $e \lessdot_E e'$, and that the race can be reversed. Formally, if $E' \lesssim E$ and e occurs immediately before e' in E', then $proc(e')$ was not blocked before the occurrence of e.

2.9 Dynamic Partial Order Reduction

Before explaining the DPOR algorithm it is important to define sufficient sets.

Definition 2.3. (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.

Many techniques have been proposed in order to implement a DPOR algorithm. What most of these techniques share in common is the structure shown in Algorithm. 1.

Algorithm 1: General form of DPOR

```
1 Explore(∅);
2 Function Explore(E)
3     let T = Sufficient_set(final(E));
4     for all t ∈ T do
5         Explore(E.t) ;
```

where $final(E)$ represents the state that will be reached when the execution sequence E is executed.
The algorithm above describes a DFS search in the state space of all possible interleavings. As it can be inferred from the algorithm the most important step is that of the calculation of the set T.

Definition 2.4. (Enabled sets, $enabled(s)$) Given a state s, $enabled(s)$ represents the set of all the threads that can be scheduled immediately after s.

An obvious property the sufficient sets must hold is that Sufficient_set($final(E)$) $\subseteq$ $enabled(E)$.
Intuitively $enabled(s)$ represents the threads that are not blocked or have already finished their execution.
In bibliography many types of sufficient sets can be found [Gode96]. In this thesis we mainly focus on persistent sets and on source sets.

2.10 Persistent Sets

A persistent set in a state s is a sufficient set of transitions to explore from s while maintaining local state reachability for acyclic state spaces [Gode93]. A selective search using persistent sets explores a persistent set of transitions from each state s where $enabled(s) \neq \emptyset$ and prunes enabled transitions that are not persistent in s. In a more formal way:

Definition 2.5. (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 $t \in T$.

The above definition can be described as follows: If $t \in T$ and there is another thread t' that can be executed until a command which is in a race with t, then t' belongs in the persistent set.
Notice that the definition of persistent sets suggests a way to construct them.
In Figure 2.3 two different examples of persistent set construction are given. We denote the persistent set of branches the execution will take with BR. In the first, let a concurrent program contain 3 threads p, q, and r. Thread p changes the value of the variable (writer) and the other (q and r) just read this variable (readers). Let $p.q.q.r.r$ be an interleaving. According to the definition of the persistent sets, q and r are in a race with p, thus, q and r must also be on the persistent set of the first command of the interleaving. In Figure 2.3 we notice that both r and q threads are added to the persistent set of the first command of the trace since both conflict with the write operation. In the second example, let p and r be a readers and q be a writer. We notice that both r and q are added. However, there is no conflict between p and r since both p and r just read the variable x. The reason why the thread r is added is the conflict that will be produced by the q's write operation.

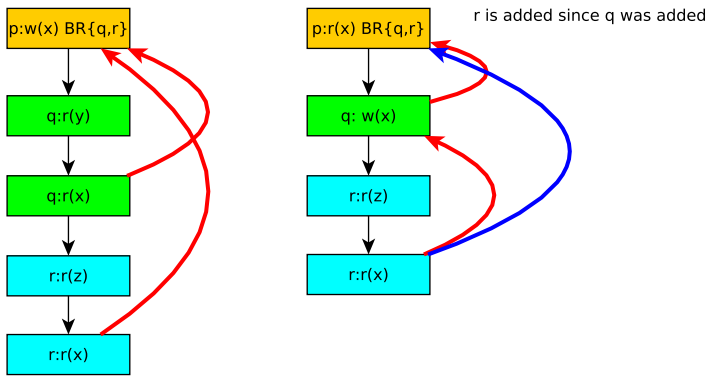

Figure 2.3: Construction of persistent sets

2.11 Source Sets

Before defining source sets, we have to give some more useful definitions.

Definition 2.6. ($dom(E)$) The set of events-transitions happening during the scheduling of E.

Definition 2.7. (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 2.8. (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 $p \in enabled(s_{[E]})$.

The point of these concepts is that for an execution sequence $E.w$:

- $p \in I_{[E]}(w)$ iff there is a sequence w' such that $E.w \simeq E.p.w'$, and
- $p \in WI_{[E]}(w)$ iff there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

Definition 2.9. (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 $WI_{[E]}(w) \cap P = \emptyset$.

A source set is a set of threads that guarantee that the whole state space will be explored. Notice that their is no requirement related to the races of the events. What the above definition implies is that source can be considered every set of threads that contains these threads that are able to cover the whole state-space. In fact, it suggests a property for the sufficient sets to hold.

2.12 Sleep Sets

Another technique complementary to the persistent or source sets aiming to reduce the number of interleavings is the sleep set technique. Sleep sets prohibit visited transitions from executing again until the search explores a dependent transition. 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.
A short example on sleep sets is the following: Let the concurrent program of one writer (w1) and two readers (r1,r2). let w1 <0.0>: w(x) r1 <0.1>: (local operations), r(x) and r2 <0.2>: (local operations), r(x).
The resulted traces are demonstrated in the Listing 2.5. Notice that whenever a process executes a command which is in race with some command of another process and the latter process is "sleeping", it wakes up. As we can infer from the execution of the DPOR algorithm the interleaving which started from r2 was blocked since it would lead to an interleaving which has already been explored. Notice that this is due to the fact that r1 cannot "wake up" since its first transition (local operations) does not conflict with any other transition in the program.
It can be proved [Gode96] that sleeps will eventually block all the redundant interleavings and thus the only interleavings that will be explored till their end (where all threads that

could be executed, have been executed). As a result an optimal algorithm should be able to not consider these interleavings whatsoever.

```
TSOTraceBuilder (debug print):
  (<0>,1-6)              [1]                    SLP:{}
  (<0>,7)                [7]                    SLP:{}
  (<0>,8)                [8]                    SLP:{}
  (<0>,9-13)             [9]                    SLP:{}
      (<0.0>,1-2)        [10, 0, 1]             SLP:{} branch: <0.1>(0)
          (<0.1>,1-2)    [11, 0, 0, 0, 1]       SLP:{}
          (<0.1>,3)      [11, 0, 0, 0, 3]       SLP:{}
          (<0.1>,4)      [11, 0, 1, 0, 4]       SLP:{}
          (<0.1>,5-6)    [11, 0, 1, 0, 5]       SLP:{}
              (<0.2>,1-2) [12, 0, 0, 0, 0, 0, 1] SLP:{}
              (<0.2>,3)   [12, 0, 0, 0, 0, 0, 3] SLP:{}
              (<0.2>,4)   [12, 0, 1, 0, 0, 0, 4] SLP:{}
              (<0.2>,5-6) [12, 0, 1, 0, 0, 0, 5] SLP:{}
 ============================
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)              [1]                    SLP:{}
  (<0>,7)                [7]                    SLP:{}
  (<0>,8)                [8]                    SLP:{}
  (<0>,9-13)             [9]                    SLP:{}
          (<0.1>,1-2)    [11, 0, 0, 0, 1]       SLP:{<0.0>}
          (<0.1>,3)      [11, 0, 0, 0, 3]       SLP:{<0.0>}
          (<0.1>,4)      [11, 0, 0, 0, 4]       SLP:{<0.0>}
      (<0.0>,1-2)        [11, 0, 1, 0, 4]       SLP:{} branch: <0.2>(0)
          (<0.1>,5-6)    [11, 0, 0, 0, 5]       SLP:{}
              (<0.2>,1-2) [12, 0, 0, 0, 0, 0, 1] SLP:{}
              (<0.2>,3)   [12, 0, 0, 0, 0, 0, 3] SLP:{}
              (<0.2>,4)   [12, 0, 1, 0, 4, 0, 4] SLP:{}
              (<0.2>,5-6) [12, 0, 1, 0, 4, 0, 5] SLP:{}
 ============================
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)              [1]                    SLP:{}
  (<0>,7)                [7]                    SLP:{}
  (<0>,8)                [8]                    SLP:{}
  (<0>,9-13)             [9]                    SLP:{}
          (<0.1>,1-2)    [11, 0, 0, 0, 1]       SLP:{<0.0>}
          (<0.1>,3)      [11, 0, 0, 0, 3]       SLP:{<0.0>}
          (<0.1>,4)      [11, 0, 0, 0, 4]       SLP:{<0.0>} branch: <0.2>(0)
              (<0.2>,1)   [12, 0, 0, 0, 0, 0, 1] SLP:{<0.0>}
          (<0.1>,5-6)    [11, 0, 0, 0, 5]       SLP:{<0.0>}
              (<0.2>,2)   [12, 0, 0, 0, 0, 0, 2] SLP:{<0.0>}
              (<0.2>,3)   [12, 0, 0, 0, 0, 0, 3] SLP:{<0.0>}
              (<0.2>,4)   [12, 0, 0, 0, 0, 0, 4] SLP:{<0.0>}
      (<0.0>,1-2)        [12, 0, 1, 0, 4, 0, 4] SLP:{}
              (<0.2>,5-6) [12, 0, 0, 0, 0, 0, 5] SLP:{}
 ============================
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)              [1]                    SLP:{}
  (<0>,7)                [7]                    SLP:{}
  (<0>,8)                [8]                    SLP:{}
  (<0>,9-13)             [9]                    SLP:{}
          (<0.1>,1-2)    [11, 0, 0, 0, 1]       SLP:{<0.0>}
          (<0.1>,3)      [11, 0, 0, 0, 3]       SLP:{<0.0>}
              (<0.2>,1-2) [12, 0, 0, 0, 0, 0, 1] SLP:{<0.0>, <0.1>}
              (<0.2>,3)   [12, 0, 0, 0, 0, 0, 3] SLP:{<0.0>, <0.1>}
              (<0.2>,4)   [12, 0, 0, 0, 0, 0, 4] SLP:{<0.0>, <0.1>}
      (<0.0>,1-2)        [12, 0, 1, 0, 0, 0, 4] SLP:{<0.1>}
          (<0.1>,4)      [12, 0, 1, 0, 4, 0, 4] SLP:{}
          (<0.1>,5-6)    [12, 0, 1, 0, 5, 0, 4] SLP:{}
              (<0.2>,5-6) [12, 0, 0, 0, 0, 0, 5] SLP:{}
 ============================
```

Listing 2.5: Sleep set example

```
Initially: x = y = z = 0


p:                          q:                          r:
m := x; (p1)                n := y; (q1)                o := z; (r1)
if (m = 0) then             if (n = 0) then             if (o = 0) then
    z := 1; (p2)                x := 1; (q2)                y := 1; (r2)
```

Figure 2.4: Program with non-minimal persistent sets

2.13 Comparing Persistent Sets with Source Sets

Note that the definition of source sets is much more relaxed than the definition of the persistent sets. This relaxation enables the source sets to be much more efficient than the persistent sets. In Figure 2.4 an example is given were source sets and persistent sets differ.
From the example, it is clear that the reason why source sets are an improvement over persistent sets is the fact that minimum source sets can eliminate sleep set blocked traces i.e., traces that would eventually be blocked by the sleep sets. An algorithm that would only calculate minimal source sets would be optimal [Abdu14], hence would never explore two equivalent interleavings.
It is obvious that a single transition cannot be a source set. For instance, the set $\{p_1\}$ does not contain the initials of execution $q_1.q_2.p_1.r_1.r_2$, since q2 and p1 perform conflicting accesses. On the other hand, any subset containing two enabled transitions is a source set. To see this, let us choose $\{p_1, q_1\}$ as the source set. Obviously, $\{p1, q1\}$ contains an initial of any execution that starts with either p_1 or q_1. Any execution sequence which starts with r_1 is equivalent to an execution obtained by moving the first step of either p_1 or q_1 to the beginning:

- If q_1 occurs before r_2, then q_1 is an initial, since it does not conflict with any other transition.

- If q_1 occurs after r_2, then p_1 is independent of all steps, so p_1 is an initial. We claim that $\{p_1, q_1\}$ cannot be a persistent set. The reason is that the execution sequence $\{r_1.r_2\}$ does not contain any transition in the persistent set, but its second step is dependent with q_1. By symmetry, it follows that no other two-transition set can be a persistent set.

In other words, persistent sets have the unpleasant property that adding a process may disturb the persistent set so that even more process may have to be added. This property is relevant in the context of DPOR, where the first member of the persistent set is often chosen rather arbitrarily (it is the next process in the first exploration after E), and where the persistent set is expanded by need.
Continuing the comparison between source sets and persistent sets, we first note some rather direct properties, including the following.

- Any persistent set is a source set.

- Any one-process source set is a persistent set.

2.14 Bounded Search and Preemption Bounding

Bounded search explores only executions that do not exceed a bound [Coon13, Thom16]. The bound may be any property of a sequence of transitions. A bound evaluation function $B_v(E)$ computes the bounded value for a sequence of transitions E. A bound evaluation function B_v and bound c are inputs to bounded search. Bounded search may not visit all relevant reachable states; it visits only those that are reachable within the bound. If a search explores all relevant states reachable within the bound, then it provides bounded coverage.
An algorithm that could describe a bounded search would be the following:

Algorithm 2: Bounded-DPOR

Result: Explore the whole statespace

```
1 Explore(∅);
2 Function Explore(E)
3     T = Sufficient_set(final(E)) for all t ∈ T do
4         if B_v(E.t) ≤ c then
5             Explore(E.t)
```

The only difference between the unbounded and the bounded version of the algorithm is the if-statement on line 4 which allows for an interleaving to be explored only if the bound has not been exceeded.
What is needed next is an appropriate definition of the function B_v that calculates a value that the bounded-DPOR tries to keep bounded, and the sufficient set.
In this thesis, we mainly focus on preemption-bounded search.
Preemption-bounded search limits the number of preemptive context switches that occur in an execution [Musu07]. The preemption bound is defined recursively as follows.

Definition 2.10. (Preemption bound)
$P_b(\emptyset) = 0$

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

The previous definition describes what a preemptive context switch is. A preemptive context switch happens when the previously running thread could execute its next step but it does not due to the scheduling of another thread. Hence, a preemptive switch will increase the preemption bound.

2.15 Preemption Bounded Persistent Sets

A set that has been proposed as a sufficient for preemption bounded search is the preemption bounded persistent set [Coon13].
An important observation is that the execution of a thread until it gets blocked or terminates will not increase the bound count.

Definition 2.11. ($ext(s,t)$) 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. $t.tid \notin enabled(final(E.\beta))$

Next, we need to define preemption bounded persistent sets. We denote with $A_G(P_b, c)$ the generic bounded state space with bound function P_b and bound c. $last(a)$ denotes the last execution step of an execution sequence a

Definition 2.12. (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 $ext(s,t) \leftrightarrow next(final(E.a), last(a).tid)$

When dealing with preemption bounded DPOR it is useful to introduce the idea of blocks in an execution sequence.

Definition 2.13. (Block of execution sequence) Block in an execution sequence is the maximal subsequence of execution steps that consists of execution steps of the same thread.

In the Figure 2.5 there are three blocks. The first block is coloured with yellow, the second with green and the third with blue.

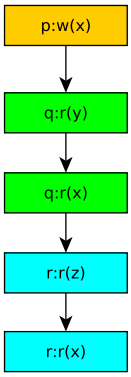

Figure 2.5: Example of blocks

Let P be persistent set. A preemption bounded persistent set is a set that contains all $p \in P$ with the addition of all the threads that would be added in a block that would be created when p was scheduled. These threads are called conservative threads and their goal is to allow the coverage of interleavings that would not exceed the bound. Notice that an interleaving can be both conservative and non-conservative. Preemption bounded persistent sets extend a persistent set by adding all the threads that will create a new block after the block that will be created by the persistent set.

Chapter 3

Unbounded DPOR

In this chapter the Classic-DPOR and Source-DPOR algorithms are discussed. These algorithms differ in terms of the sufficient sets they use in order to cover the state space. DPOR is based on persistent sets while Source-DPOR relies on source sets.

3.1 Source-DPOR

In this section Source-DPOR [Abdu14] is presented. Even though Source-DPOR is posterior to Classic-DPOR, as the name suggests, we choose to present it first since any algorithm presented in this thesis will be related to Source-DPOR. This is due to the fact that all the algorithms were implemented in Nidhugg which is based on Source-DPOR.

Algorithm 3: Source-DPOR

```
1  Explore(⟨⟩,∅);
2  Function Explore(E,Sleep)
3      if ∃p ∈ (enabled(s_[E])\Sleep) then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep) do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let E' = pre(E, e);
8                  let u = notdep(e, E).p;
9                  if I_{E'}(u) ∩ backtrack(E') = ∅ then
10                     add some q' ∈ I_[E'](u) to backtrack(E') ;
11             let Sleep' := {q ∈ Sleep | E ⊨ p◇q};
12             Explore(E.p, Sleep') ;
13             add p to Sleep ;
```

Initially an arbitrary enabled and not sleeping process is chosen and added to the $backtrack(E)$.
Each step of the algorithm consists of two separate phases. During the first phase of the algorithm the race detection takes place. The algorithm picks a process p which can execute its next step i.e, $p \in enabled(s_{[E]})$ and appends it to the explored trace E. Then the algorithm finds every event e which is already contained in the explored trace ($e \in dom(E)$) and can be reversed with the next step of p.
This happens in order to explore the execution sequence where p happens-before e. This execution sequence consists of:

- $E' \equiv pre(E, e)$: the subsequence E' which consists of events scheduled before e in E.

- $u \equiv notdep(e, E).p$: the concatination u of all the events that are scheduled after e in E but are independent with e and p.
- $proc(e)$ which is the id of the process that caused the event e.

Then the algorithm checks whether some process in $I_{[E']}(u)$ is already in $backtrack(E')$. If not, then a process in $I_{[E']}(u)$ is added to backtrack.
In the exploration phase, the exploration starts from $E.p$. The important part is the calculation of the new sleep set at that step since some processes may have woken up. If the next step of a process conflicts with the $next(p)$ then this process must wake up. As a result, the sleep set consists of the already sleeping processes whose next steps do not interfere with the $next(p)$, i.e, $Sleep' := \{q \in Sleep \mid E \models p \diamondsuit q\}$ After finishing the exploration of $E.p$, p is added to the sleep set because we want to refrain from executing an equivalent trace.

3.2 Classic-DPOR

Since Source-DPOR uses vector clocks to track events the DPOR using Clock Vectors [Flan05] variation will be used which is shown in Algorithm 4.

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

```
1  Function Explore(E,C)
2      let s := last(E);
3      for all process p do
4          if ∃i = max({i ∈ dom(E) | E_i is dependent and may be co-enabled with next(s,p)
             and i ⋠ C(p)(proc(E_i))}) then
5              if p ∈ enabled(pre(E,i))) then
6                  add p to backtrack(pre(E,i)) ;
7              else
8                  add enabled(pre(E,i)) to backtrack(pre(E,i)) ;
9      if ∃p ∈ enabled(s) then
10         backtrack(s) := p ;
11         let done = ∅;
12         while ∃p ∈ (backtrack(s)\done) do
13             add p to done ;
14             let t = next(s,p);
15             let E' = E.t;
16             let cu = max{C(i) | i ∈ 1..|S| and E_i dependent with t};
17             let cu2 = cu[p := |E'|];
18             let C' = C[p := cu2, |E'| := cu2];
19             Explore(E',C') ;
```

A clock vector $C(p)$ is maintained through out the algorithm for each process p. $C(p_i) = \langle c_1, c_2, ..., c_m \rangle$ represents the clock vector of process p_i. where c_j is the index of the last transition by process p_j such that $c_j \rightarrow_s p_i$. Intuitively, the clock vector of a process p_j informs the process p_j about the execution steps of the other processes that happen-before the execution steps of p_j. The clock vectors are based on the Lamport's algorithm [Lamp78] and more details of their usage in DPOR can be found here [Flan05]. We represent the initial state of all vector clocks as $\perp = \langle 0, ..., 0 \rangle$. With $C(p)(proc(E_i))$ we represent the value of clock vector of process p for the process in the i-th step of E.

The initial state of Classic-DPOR is an empty sequence of events with all vector clocks set to $\perp$.
The Classic-DPOR consists of two phases. The first phase is the race detection. During that phase, the next transition of all processes p is considered. For each such transition $next(s, p)$ (which may be enabled or disabled in s), the last transition dependent transition i in E is computed. The computation takes place in line 4. If there exists such a transition i, there might be a race condition or dependency between i and $next(s, p)$, and hence we might need to introduce a "backtracking point" in the state $pre(S, i)$, i.e., in the state just before executing the transition i. If p is enabled then it is added as backtrack point. Otherwise the whole set of the enabled transitions is added as backtracks.
During the exploration phase a process p in $enabled(s)$ is added to the backtrack point as in the Source-DPOR. Then the vector clocks are updated according to Lamport's algorithm.

3.3 Comparison of Classic-DPOR and Source-DPOR

As one can easily notice, both algorithms consist of the same two phases i.e., the race detection phase and the exploration phase. Moreover both algorithms are based on vector-clocks, in spite of the fact that their usage is just implied in the Source-DPOR in line 6 and line 8 of Algorithm 3.
The major difference lies in the race detection phase. During the race detection in Classic-DPOR all processes p are considered, before even be scheduled, and, thus, many of them may be backtracked. On the other hand, in Source-DPOR only the last scheduled process is considered. Moreover the two algorithms differ in the way they deal with the case of non-enabled process in the backtrack addition. In the case of Classic-DPOR the whole enabled set will be added whereas none or just one process will be added in the Source-DPOR. All in all, Source-DPOR takes advantage of Clock vectors more than Classic-DPOR does.

3.4 Combining Classic-DPOR and Source-DPOR

In order to take advantage of Nidhugg's infrastructure Algorithm 5 will be used. As a result there is no need to add all available threads when p is not enabled. If no sufficient candidate is found a candidate suggested by the Source-DPOR algorithm is added. This way of calculating persistent sets is considered to be more complex and thus a bad option [Gode05]. However, source sets algorithm is closer to this approach.
The Algorithm 5 differs from Source-DPOR (Alg. 3) in the calculation of the initials. It is clear that persistent algorithm implemented differs from Source-DPOR in the calculation of the initials. Specifically a subset of the initials (CI) that happen-before p is used. Intuitively in the case of a writer and two readers, both readers will be added to the branch since the first read does not happen before the second one. To generalize this idea: since Nidhugg does not enable us to create branches for $last(E)$, when at the scheduling phase, we add the branches later as in Source-DPOR (Alg. 3). When a race is considered usually only the thread that causes the race will be added since CI contains this thread only.
We will give an intuition why Nidhugg-DPOR calculates a persistent set and explain why when the algorithm finishes a persistent set will have been calculated in each step. The interesting part is to show for every branch that is a write command all the process that conflict with this write command happen concurrently with this command will be added to the branch set.
Let us assume two processes that are in race with $last(S)$.

- Case 1: at least one process contains a write command. We know that the two

Algorithm 5: Nidhugg-DPOR

```
1  Explore(⟨⟩,∅);
2  Function Explore(E,Sleep)
3      if ∃p ∈ (enabled(s_[E])\Sleep then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep) do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let E' = pre(E, e);
8                  let u = notdep(e, E).p;
9                  let CI = {i ∈ I_{E'}(u) | i → p};
10                 if CI ∩ backtrack(E') = ∅ then
11                     if CI ≠ ∅ then
12                         add some q' ∈ CI to backtrack(E') ;
13                     else
14                         add some q'I_{E'}(u) to backtrack(E')
15             let Sleep' := {q ∈ Sleep | E ⊨ p◇q} ;
16             Explore(E.p, Sleep) ;
17             add p to Sleep ;
```

processes will be inverted at some point. Since Nidhugg-DPOR ignores weak initials it will branch both processes. In Figure 3.1 we notice that processes q and r are inverted. 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.

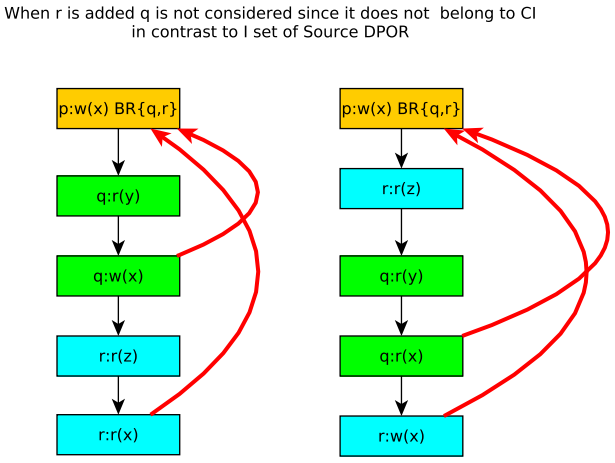

Figure 3.1: Construction of persistent sets in Nidhugg when there is a write process

- 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 $backtrack$. In Figure 3.2 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.

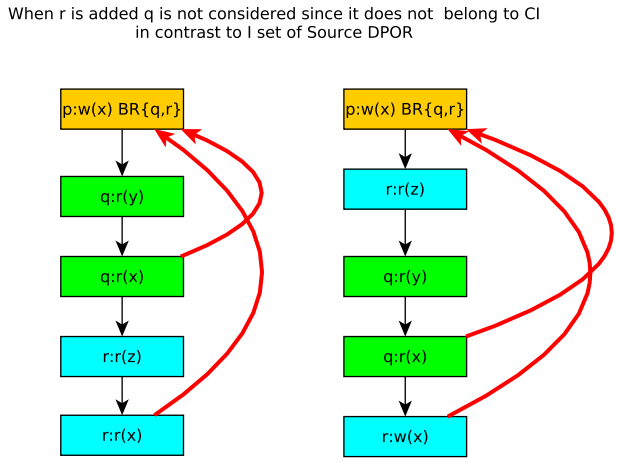

Figure 3.2: Construction of persistent sets in Nidhugg when both are read processes

It is clear that no process that does not belong to the $backtrack(S)$ has race with a process that belongs to $backtrack(S)$. The introduction of CI set allows as to ignore the initials that Source-DPOR takes advantage of in order to optimize the exploration.

Chapter 4

Bounding Techniques for DPOR

In this chapter Bounding Techniques for DPOR are discussed as well as the challenges emerging from these techniques. The challenges of bounding the DPOR have already been discussed.

4.1 Naive-BPOR

The first bounded technique to be presented is the Naive-BPOR (Algorithm 6). The purpose of the algorithm is to block traces that exceed the bound limit.

Algorithm 6: Naive-BPOR

```
1  Explore(⟨⟩,∅,b);
2  Function Explore(E,Sleep,b)
3      if ∃p ∈ (enabled(s_[E])\Sleep) such that B_v(E.p) ≤ b) then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) ≤ b do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let E' = pre(E, e);
8                  let u = notdep(e, E).p;
9                  if I_{E'}(u) ∩ backtrack(E') = ∅ then
10                     add some q' ∈ I_[E'](u)tobacktrack(E') ;
11             let Sleep' := {q ∈ Sleep | E ⊨ p◇q};
12             Explore(E.p, Sleep, b) ;
13             add p to Sleep ;
```

The Algorithm 6 is almost the same with Source-DPOR(Algorithm 3). The only additions made are related to the thread scheduling. When a a step of a process p added to E result the trace $E.p : B_v(E.p) > b$ then this process is not allowed to be scheduled. This algorithm is not sound i.e., it does not examine every trace that compensates with the bound
Let the writer-2-readers example with $b = 0$. An example of exploration for bound $b = 0$ is given in Figure 4.1.

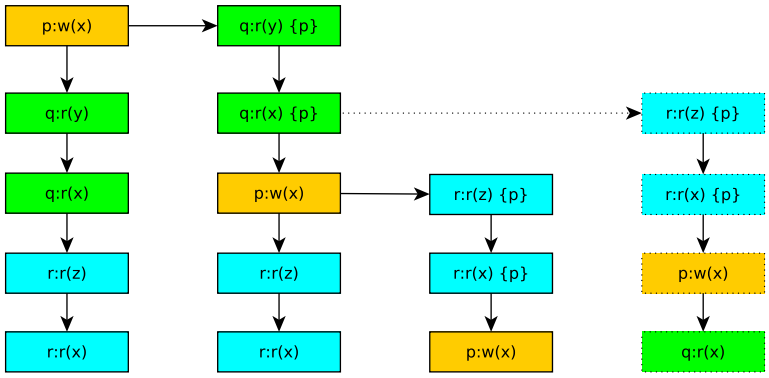

Figure 4.1: Naive-BPOR for bound=0

As we can see there are 4 traces that do not exceed the bound. These are: $p.q.q.r.r$, $q.q.p.r.r$, $r.r.p.q.q$, $q.q.r.r.p$. However the Naive-BPOR is not able to explore them all; $r.r.p.q.q$ is not explored. As it was shown in the comparison of persistent and source sets, r is never registered as the first event of the trace since this will lead to a sleep set blocked trace. The branch that would lead to an equivalent trace to $r.r.p.q.q$ is rejected since it would have higher bound count. Interestingly enough a Naive-BPOR based on persistent sets would explore a larger state-space since it would have explored would probably have explored the trace $r.r.p.q.q$.

4.2 BPOR

The next step is to implement a sound preemption bounded algorithm (BPOR) [Coon13]. This algorithm would rely on persistent sets. Since we have correctly implemented persistent sets for Nidhugg-DPOR we can easily implement a BPOR algorithm. The novelty of the BPOR is the introduction of conservative branches. These are branches that are introduced in order to guarantee the exploration of the whole state space. It is common for a trace to exceed the bound limit whereas there is an equivalent trace which does not. The conservative branches are used for this purpose.
To make this idea more clear we introduce the notion of Trace block.

Definition 4.1. (Trace block) For a trace T a sequence B of consecutive events is a trace block iff all events happen in the same thread i.e. all the events have the same thread id.

The idea behind conservative branches is quit simple. When a branch is added a conservative branch is added at the beginning of the corresponding block. Usually concurrent events take place inside a block. As a result when a branch is taken then the preemption count will most probably increased. However, had this branch been added at the beginning of the block the preemption count would not have been increased. BPOR is presented in detail in Algorithm 7. The BPOR algorithm differs from Classic-DPOR in the double nested loop which is introduced in line 3. This inner loop is introduced in order for the BPOR to compensate with the defintion of the preemption-bounded persistent sets which was introduced in Chapter 2. Moreover, as in Naive-BPOR, the exploration is also be terminated if a trace exceeds the bound limit.

4.3 Nidhugg-BPOR

In order to take advantage of Nidhuggs's infrastructure, a modification of this algorithm is used. The algorithm is presented in Algorithm 8.

Algorithm 7: BPOR

```
1  Function Explore(E)
2      let s := last(E);
3      for all process p do
4          for all process q ≠ p do
5              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
6                  if p ∈ enabled(pre(E, i))) then
7                      add p to backtrack(pre(E, i)) ;
8                  else
9                      add enabled(pre(E, i)) to backtrack(pre(E, i)) ;
10                 if j = max({j ∈ dom(E) | j = 0 or S_{j−1}.tid ≠ S_j.tid and j < i}) then
11                     if p ∈ enabled(pre(E, i))) then
12                         add p to backtrack(pre(E, i)) ;
13                     else
14                         add enabled(pre(E, i)) to backtrack(pre(E, i)) ;
15     if p ∈ enabled(s) then
16         add p to backtrack(s) ;
17     else
18         add any u ∈ enabled(s) to backtrack(s) ;
19     let visited = ∅;
20     while ∃u ∈ (enabled(s) ∩ backtrack(s)\visited) do
21         add u to visited ;
22         if (B_v(S.next(s, u)) ≤ c) then
23             Explore(S.next(s, u)) ;
```

Notice that persistent sets are calculated for both the conservative and the non-conservative branches.
A critical challenge arises when a DPOR algorithm is used in tandem with sleep sets. This stems from the fact that conservative branches are not added due to a concurrent event. By observing the sleep set algorithm we notice that if we follow the same strategy as with non-conservative branches many traces will end up being blocked.
Let us take the writer-2readers example as shown in Figure 4.2.

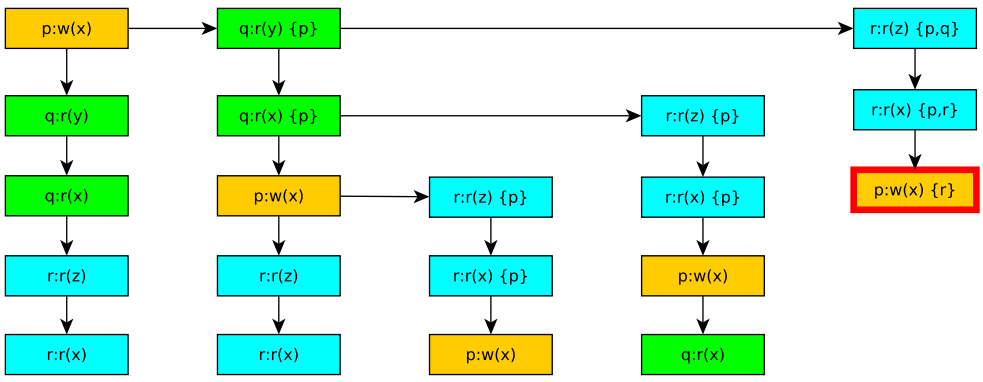

Figure 4.2: Usage of non-conservative branches

Algorithm 8: Nidhugg-BPOR

```
1  Explore(⟨⟩,∅,b);
2  Function Explore(E,Sleep,b)
3      if ∃p ∈ ((enabled(s_[E])\Sleep) and B_v(E.p) <= b then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) <= b do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let E' = pre(E, e);
8                  let u = notdep(e, E).p;
9                  let CI = {i ∈ I_{E'}(u) | i → p};
10                 if CI ∩ backtrack(E') = ∅ then
11                     if CI ≠ ∅ then
12                         add some q' ∈ CI to backtrack(E') ;
13                     else
14                         add some q' ∈ I_[E'](u) to backtrack(E') ;
15                 let E'' = pre_block(e, E);
16                 let u = notdep(e, E).p;
17                 let CI = {i ∈ I_{E''}(u) | i → p};
18                 if CI ∩ backtrack(E') = ∅ then
19                     if CI ≠ ∅ then
20                         add some q' ∈ CI to backtrack(E') ;
21                     else
22                         add some c(q') ∈ I_[E'](u) to backtrack(E'') ;
23             let Sleep' := {q ∈ Sleep | E ⊨ p◇q} ;
24             Explore(E.p, Sleep) ;
25             if p is not conservative then
26                 add p to Sleep ;
```

We notice that the last trace is sleep set blocked while it should be examined. The algorithm is unaware that the process r should be removed from the sleep set since there is no or will ever be found any conflict with the first command of the thread which is related with a non shared variable. In order to deal with this problem, when a conservative branch is chosen then it should not be added to the sleep set. However there must be a set recording all the branches that were added at this certain point of the trace so no thread is added twice. Without this set the exploration would continue indefinitely. The solution is based on the notion of the conservative sets where every thread that was added to the branch is recorded.

Intuitively the algorithm is the same with the Nidhugg-DPOR with the addition of the conservative branches. The algorithm is based on the idea of the conservative sets where every thread that was added to the branch is recorded. However many challenges arise which are discussed in the implementation section.

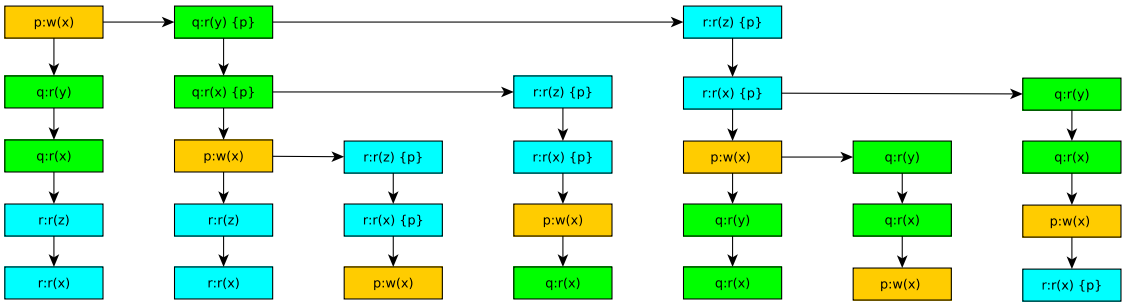

Figure 4.3: Example of BPOR execution

4.4 Source-BPOR

Having discussed BPOR algorithm and Nidhugg-BPOR, the next step is to try combine source sets with the algorithm. The first observation we have to make is that source sets and thus the algorithm for creating these sets is not suitable for adding conservative branches. A quick explanation is given in the next writer-2-readers example even though the problem will be further discussed later. Let us assume that we have followed the source set algorithm for adding conservative sets. The results are shown at Figure 4.4.

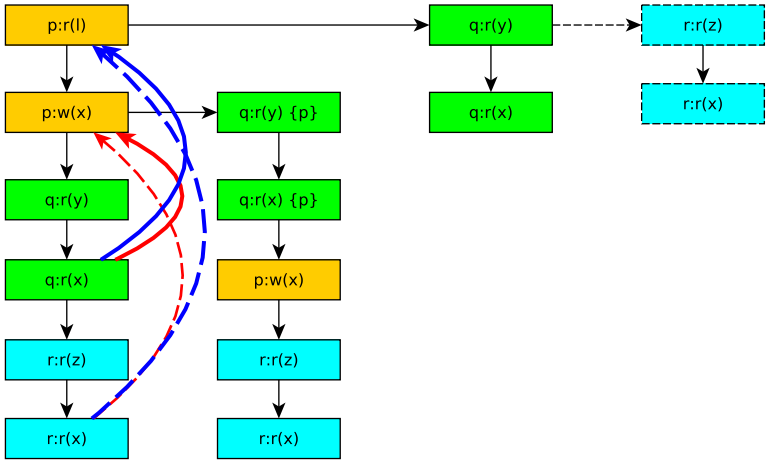

Figure 4.4: Following source sets for conservative branches

It is clear that some traces are not explored. Specifically, the trace which start with r has been rejected. The reason is that it shares the same initials with r1 even at the beginning of that block. As a result the algorithm must create to persistent sets when conservative threads are added.
Having made the preceding observations the algorithm used for Source-BPOR is shown in Algorithm 9.
We can notice that the Algorithm 9 works as Source-DPOR (Alg. 3) for non-conservative traces and as BPOR (Alg. 8) for conservative traces.

Algorithm 9: Source-BPOR

```
1  Explore(⟨⟩,∅,b);
2  Function Explore(E,Sleep,b)
3      if ∃p ∈ ((enabled(s_[E])\Sleep) and B_v(E.p) <= b then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) <= b do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let E' = pre(E, e);
8                  let u = notdep(e, E).p;
9                  if I_{E'}(u) ∩ backtrack(E') = ∅ then
10                     add some q' ∈ I_[E'](u) to backtrack(E') ;
11                 let E'' = pre_block(e, E);
12                 let u = notdep(e, E).p;
13                 let CI = {i ∈ I_{E''}(u) | i → p};
14                 if CI ∩ backtrack(E') = ∅ then
15                     if CI ≠ ∅ then
16                         add some q' ∈ CI to backtrack(E') ;
17                     else
18                         add some c(q') ∈ I_[E''](u) to backtrack(E'') ;
19             let Sleep' := {q ∈ Sleep | E ⊨ p◇q} ;
20             Explore(E.p, Sleep) ;
21             if p is not conservative then
22                 add p to Sleep ;
```

4.5 Challenges Arising from the addition of conservative branches

4.5.1 Conservative Branches

In a previous section we saw that conservative sets cannot utilize the sleep set optimization. This is due to the fact that these branches are not produced by conflicts and as a result it is impossible to "wake up" another process whose next step may be a local operation. The problem is getting even more complex considering that when a conservative branch is added the algorithm "forgets" what was previously in this place. This lack of memory leads to an explosion of the state space. This explosion is greater than the explosion happening when exploring the complete state space. In order to explain this better an example involving a writer and three readers is given in Figure 4.5. Specifically, the writer process just writes a value on a variable x while the reader processes read this variable after reading another variable unique for each process which can be regarded as a local operation.

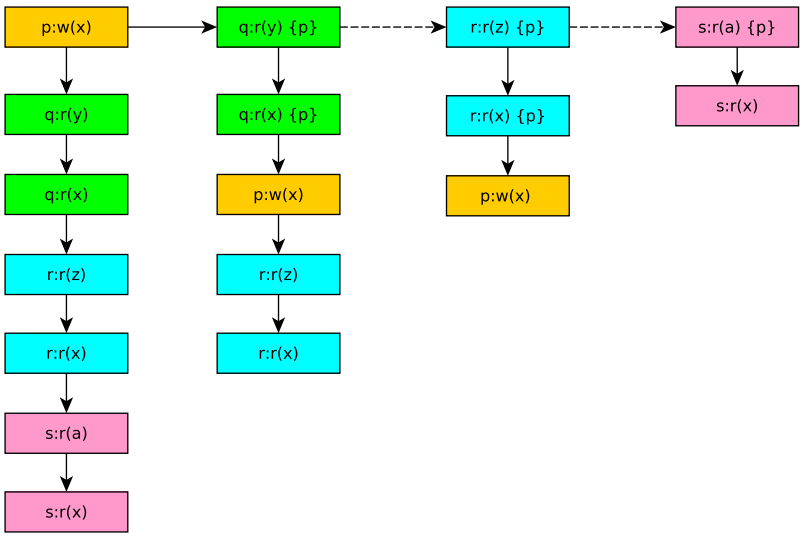

Figure 4.5: writer-3-readers explosion

As we can infer from the example there are more states to be explored than the expected ones. The expected number of traces would be 8 but $4!$ are explored. Notice that every reader is backtracked in the first command since it conflicts with $w(x)$. These branches should be blocked by the sleep sets in the unbounded search. However, these branches are also conservative ones. There are extra traces such as the $r2.r1.w$ trace which explores the already explored $r1.r2.w$ trace. The algorithm is not aware when adding the conservative trace whether an equivalent trace will be explored. The situation is worse when more readers are to read the same global variable. The total traces explored for a big bound approach the $N!$ where N the total number of threads. We can easily notice that execution seems to be aware of the previous executions since sleep sets cannot be used. This happens due to the redundant inversions of the reading operations.

4.5.2 Sleep Sets

The results of the various bounding algorithms suggest that the number of sleep set blocked traces is really small compared to the number of the explored traces. This is due to the conservative branches. It was made clear that when both a conservative and a non-conservative branch of the same thread is added then the conservative branch prevails. This trace would be redundant in an unbounded version of the algorithm but it is not in a bounded version since the non-conservative branch may had been rejected. Moreover, even if both threads had been accepted by the algorithm there may be a later scheduling which may be rejected when the trace was caused by the non-conservative branch and be accepted by the equivalent trace caused by the conservative branch.
Another "problem" with the sleep sets is that are "in favor" of the branches that increase the bound count while they block traces with lower bound count. In Figure 4.6 it is shown that the branch which would not have caused a increase of the bound is rejected.

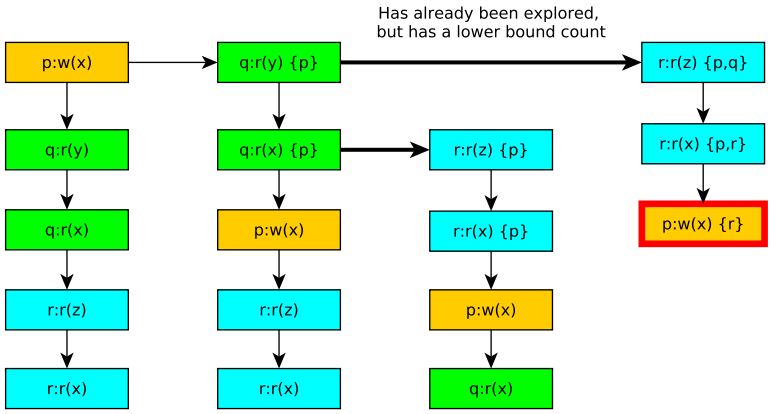

Figure 4.6: Sleep set contradiction

This behavior is quite reasonable if we consider the depth-first nature of the algorithm. As it was demonstrated in the first chapter, the DPOR algorithm performs a DFS-search. As a result it takes the branches which are located lower in the traces where the bound count is higher since more preemptive switches have taken place. The purpose of the sleep sets is to block redundant traces, i.e. traces that have already been explored. As a result the equivalent traces with lower bound count will be rejected. A method that would explore the state space in breadth first way would be unfeasible since it would entail a huge memory overhead which stems from the storage of the traces that have not be completely explored.

4.5.3 The Performance - Soundness Tradeoff

From the previous discussion it is clear that all the bounding algorithms have to deal with a tradeoff. Some algorithms may be faster but explore a smaller amount of traces, without even covering the full state space within the bound (Algorithm 6), whereas others explore the whole state space (Algorithm 8) within the bound but require more time.

Chapter 5

Algorithms' Implementations

Nidhugg is a bug-finding tool which targets bugs caused by scheduling non-determinism 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++ which can compile to llvm and implement shared-memory concurrency using the pthreads library.
At the time this thesis was written Nidhugg supported the SC, TSO, PSO and POWER memory models. On the other hand, Nidhugg does not handle data non-determinism and, thus, thread should be deterministic when run in isolation.

5.1 Nidhugg's Work Flow

Nidhugg works on the level of LLVM intermediate representation (IR). In order for Nidhugg to find a bug it creates an interpreter for the LLVM assembly. It then schedules and executes the different traces until an error is found such as the violation of an assertion. Traces play the most important role in Nidhugg as they represent different schedulings. These traces are represented as vectors of Events objects. The Event object maintains all the useful information about the event such as the pid of the thread that was executed. Branches which cause the exploration of different interleavings are also stored in the Event object. The scheduling is regulated by the Tracebuilder object which depends on the memory model used. Tracebuilder is also responsible for checking for races between different threads that access the same memory.
The execution follows in general the flow that is represented in Figure 5.1. As the flow chart suggests, Nidhugg maintains a TraceBuilder object. The trace builder tries to schedule new events according to the schedule() routine. After scheduling, the events are executed and the vector clocks are updated. Then it is checked whether this event is dependent with other events, i.e., accesses the same memory locations. After that, Nidhugg tries to add branches to the appropriate places of the branch and checks whether any errors were produced. In case of error the procedure stops and the error is reported. Notice that Nidhugg can be set so it can continue the exploration so more errors can be found. In absence of errors trace builder resets to the most recent branch. Then the whole trace is executed until that point and the next branch is scheduled. When no more resets are available, the execution terminates.
As far as the algorithm is concerned it is clear that the most important part of the flow chart is the detection of dependencies.

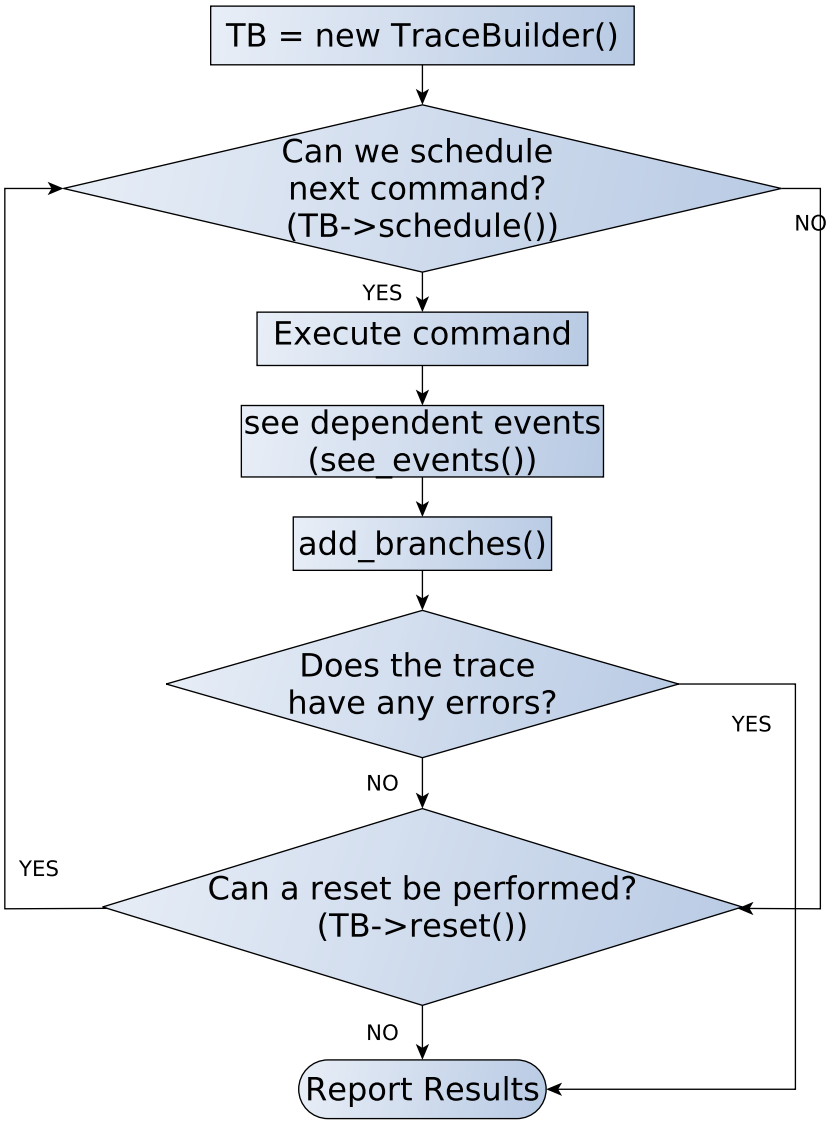

Figure 5.1: Nidhugg's Flow Chart

5.2 Branch addition in Nidhugg

Once a command, which is likely to cause a concurrent error, is scheduled the see_accesses vector is created which contains all the accesses that took place in the same memory location and calls the procedure `see_events()`.

Algorithm 10: see_events()

```
1 Function see_events(seen_access)
2     branches := seen_access − {a ∈ seen_access | a → last(E) or ∃a′ ∈ seen_access
        which happens after a} ;
3     update_clocks() ;
4     foreach b ∈ branches do
5         add_branch(b) ;
```

As it transparent from the algorithm the function's purpose is to filter out all the access that

are not in a race with the current access, i.e., the dependencies that cannot be represented as a trace with no other concurrent event occurring between them. It is important to notice that this does not suggest that these events cannot be concurrent in another scheduling. The events that were not discarded as not in race events are stored in the branch vector and checked by the `add_branch()` function.
Another task of `see_events()` is the update of the vector clocks. Two events that are in race will be concurrent for Nidhugg before the execution of this routine. At the end of the routine, however, the clocks will be updated so the last event happens after the `seen_access` events.
The `add_branch()` function shown in Algorithm 11 is the most crucial for the whole Nidhugg infrastructure.

Algorithm 11: add_branch()

```
1  Function add_branch(b)
2      candidates = ∅ ;
3      lc := null ;
4      E' := E starting from next(b) ;
5      foreach e ∈ dom(E') do
6          if b → e or ∃c ∈ candidates : c → e then
7              continue ;
8          lc := e.pid;
9          if lc ∈ candidates then
10             continue ;
11         if e.pid ∈ backtrack(b) or e.pid ∈ sleep_set then
12             return ;
13         candidates := candidates ∪ lc ;
14     backtrack(b) := backtrack(b) ∪ lc ;
```

Intuitively, add_branch does the following: Beginning from the event that conflicts with the most recently scheduled event, start traversing the vector that represents E until the closest to the end of the trace thread is found (if that is possible). In fact what really happens is the calculation of the I (initials) set. As suggested in the algorithm if there is an already added thread to the branch or the sleep set the procedure will be terminated. As a result `add_branch()` calculates a source set.

5.3 Implementation of Nidhugg-DPOR

The implementation of the persistent sets is based on the already implemented infrastructure of vector clocks. Specifically the `include()` function of vector clocks lets as determine whether $i \rightarrow p$. The algorithm of the `includes()` is given in Algorithm 12. Following the notation of Source-DPOR $\langle e, i \rangle$ is an already scheduled event which is the i th step of process e and p is the most recently backtracked event of process p.

Algorithm 12: includes() routine

```
1  Function includes(⟨e, i⟩, p)
2      return i ≤ p.clock[e]
```

To calculate the CI set we just need to prevent the `add\_branch()` function from rejecting

branches due to threads that belong to I but not to CI. An example of the modification is presented at Algorithm 13.

Algorithm 13: add_branch() for persistent sets

```
1  Function add_branch(b)
2      candidates = ∅ ;
3      lc = null  E' := Etextstartingfromnext(b) ;
4      foreach eindom(E') do
5          if b → e or ∃c ∈ candidates | c → e then
6              continue ;
7          if e → last(E) then
8              lpc := e.pid;
9          lc := e.pid;
10         if e.pid ∈ backtrack(b)ore.pid ∈ sleep_set(b) then
11             if lc → last(E) then
12                 return ;
13     if lpc then
14         backtrack(b) := backtrack(b) ∪ lpc
15     else
16         backtrack(b) := backtrack(b) ∪ lc
```

In case that E is empty we can just use a candidate that it is suggested from I set.

5.4 Implementation of Naive-BPOR

The first step in order to implement any bounding technique is the implementation of a bound counter. Since we are intrested in implementing a preemption bounded algorithm we should be able to know the bound count of each event i.e., how many preemptive switches happened until the current event.
The first observation we make is that preemption bound count is a property of each event, thus, apart from any other information the event object should maintain a counter attribute. Moreover the bound counter should be known to the TraceBuilder as well, since it is responsible for the scheduling. Such an attribute will prove pretty useful later when preemption-bounded algorithm will be implemented.The second step is to track where new events are added to the trace. There are two occasions when new events are added. New events are added during the scheduling. Here the implementation of the counter is rather straightforward. Taking advantage of an already implemented attribute that indicates the availability of the thread we can store whether the pid of the previous event corresponds to an available thread and thus, conclude if a preemptive switch happened. The other occasion when an event is added to the trace is during reset. Unfortunately, the availability attribute is not helpful here since it stores the latest state of thread and it is a property of the trace not the event. This results usually to all threads being marked unavailable when reset takes place.
There are two options on how to implement a bound counter in such a case. The first is to make thread availability a property of the event, hence, we should store all the threads' availability in each event. This option was rejected due to the overhead that would occur. The overhead would be caused by both the memory that would be required and by the fact that this vector should be constantly be copied throughout the DPOR execution. The

other solution is to infer the availability by the counter itself which as it was mentioned must be maintained in each event. Since the available attribute of a thread will be reset afterwards we can still use this attribute to store the availability of the threads. During the reset the event vector is traversed from the end to the beginning until a branch is found. We assume that each thread is available. We can make the following observation based on the bound counter of each event.
Given two consecutive events a, b , if $a.bound\_count < b.bound\_count$ then a was available. The pseudo code is given in Algorithm 14.

Algorithm 14: Should we increase the bound count?

```
1 let i = the most recent branching point;
2 bound_count := prefix[i].bound_count ;
3 if i > 0 then
4     if prefix[i].id == prefix[i − 1].id then
5         prefix[i].bound_count = ++bound_count;
6     else
7         prefix[i].bound_count = bound_count ;
```

In order to be able to verify the correct calculation of the bound count, the debug print during the reset was modified appropriately. The bound counter should work like Listing 5.1.

```
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)                BC:{0}
  (<0>,7)                  BC:{0}
  (<0>,8)                  BC:{0}
  (<0>,9-13)               BC:{0}
      (<0.0>,1-2)          BC:{0} branch: <0.1>(0)
          (<0.1>,1-2)      BC:{0}
          (<0.1>,3)        BC:{0}
          (<0.1>,4)        BC:{0}
          (<0.1>,5-6)      BC:{0}
              (<0.2>,1-2)  BC:{0}
              (<0.2>,3)    BC:{0}
              (<0.2>,4)    BC:{0}
              (<0.2>,5-6)  BC:{0}
 ============================
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)                BC:{0}
  (<0>,7)                  BC:{0}
  (<0>,8)                  BC:{0}
  (<0>,9-13)               BC:{0}
          (<0.1>,1-2)      BC:{0}
          (<0.1>,3)        BC:{0}
          (<0.1>,4)        BC:{0}
      (<0.0>,1-2)          BC:{1} branch: <0.2>(0)
          (<0.1>,5-6)      BC:{1}
              (<0.2>,1-2)  BC:{1}
              (<0.2>,3)    BC:{1}
              (<0.2>,4)    BC:{1}
              (<0.2>,5-6)  BC:{1}
 ============================
 === TSOTraceBuilder reset ===
TSOTraceBuilder (debug print):
  (<0>,1-6)                BC:{0}
  (<0>,7)                  BC:{0}
  (<0>,8)                  BC:{0}
  (<0>,9-13)               BC:{0}
          (<0.1>,1-2)      BC:{0}
          (<0.1>,3)        BC:{0}
          (<0.1>,4)        BC:{0} branch: <0.2>(0)
              (<0.2>,1)    BC:{1}
          (<0.1>,5-6)      BC:{2}
              (<0.2>,2)    BC:{2}
              (<0.2>,3)    BC:{2}
              (<0.2>,4)    BC:{2}
      (<0.0>,1-2)          BC:{3}
              (<0.2>,5-6)  BC:{3}
 ============================
```

Listing 5.1: Example of bound counter

Nidhugg expects only traces blocked due to sleep sets. Again the first step is to locate parts of the Nidhugg's code where bound block should take place. The best option for a bound block to occur is during the `schedule()` function. Before any new scheduling we just need to determine whether that bound was exceeded or not because of a reset. Moreover in order for the trace builder to know whether the trace was blocked due to the bound the bound_blocked flag was added. Finally modifications should be made in the DPORDriver so it can print correct messages about the reason why the trace was blocked. Running a random program will result the Listing 5.2.

```
Trace count: 15 (also 2 sleepset blocked, 4 schedulings and 1 branches were rejected due to the bound)
Total wall-clock time: 0.04 s
```

Listing 5.2: Naive-BPOR output

We notice that Nidhugg gives the number of scheduling that were rejected. Nidhugg schedules threads by giving priority to the older ones. As a result, as soon as an old thread becomes available it will be scheduled immediately. This will cause an increase of the bound count since it will probably stop the execution of another thread and maybe if the bound leading to the increase of the bound count which would cause the to exploration to stop if the bound count was exceeded. In order to explore as many interleavings as possible the priority of the threads was modified. Specifically, the thread executed most recently has the highest priority. If that thread is unavailable then the priority remains as it used to be with the oldest thread being prioritized.

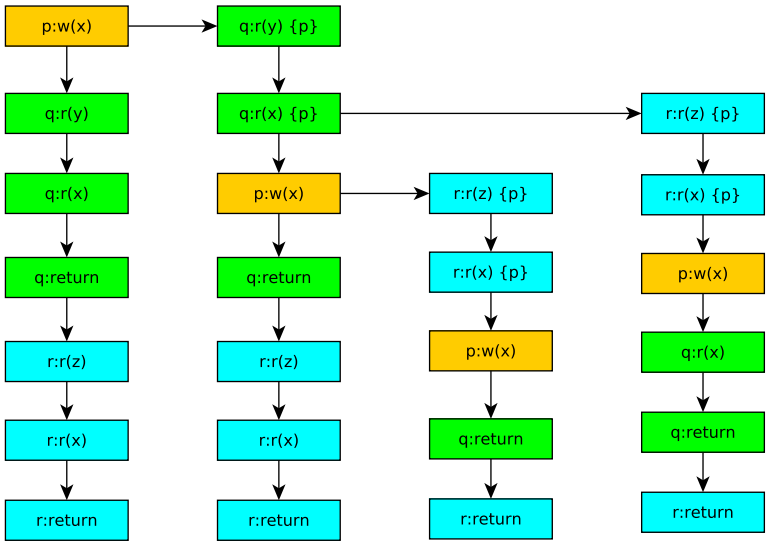

Figure 5.2: Execution without the scheduling optimization

In the Figure 5.2 an example of two of a program with two readers and a writer is demonstrated. In the second trace of the example we notice that when the read operation of q takes place p wakes up since $r(x)$ conflicts with $w(x)$. Since p is older than q, the execution of q stops and p is scheduled. Since q was enabled when p was scheduled the bound counter is increased. However, if q was scheduled again instead of p then the q would be blocked after the return command and the scheduling of p would not have increased the bound count. We can infer that, had the most recently running thread given the highest priority at least one more interleaving would have been explored and as a result more interleavings would have been explored.

5.5 Implementation of Nidhugg-BPOR

As it was made clear in previous sections for any algorithm to be implemented, the main modifications should take place in the see_events and add_branch procedures. The pseudo code for the see_events procedure is shown in Algorithm 15.
During see_events procedure we add another branch at the beginning of the block if that is possible. In order to do that we have to check whether the added thread is available or

not in that place. We use the available thread field for this purpose.

Algorithm 15: see_events() for BPOR

```
1 Explore(⟨⟩,∅);
2 Function see_events(seen_access)
3     branches := seen_access − {a ∈ seen_access | a happens before
        last(E) or ∃a' ∈ seen_access which happens after a} ;
4     update_clocks() ;
5     foreach b ∈ branches do
6         add_branch(b) ;
7         if last(E).id ∈ enabled( at the beggining of block b) then
8             add_branch( at the beginning of block b) ;
```

In case the `add_branch()` invokes directly, the `add_conservative_branch()` is called which works as the "conservative" part of the `see_events()`. The pseudo code for this procedure is shown in Algorithm 16. The `add_branch()` has already been described in Algorithm 13.

Algorithm 16: try_to_add_conservative_branches()

```
1 Explore(⟨⟩,∅);
2 Function try_to_add_conservative_branches(b)
3     if last(E).id ∈ enabled( at the beggining of block b) then
4         add_branch( at the beginning of block b) ;
```

During `add_branch()` procedure we add branches at the appropriate places using the algorithm for the persistent sets suggested in the previous section. As mentioned before two different types of branches are used. However, the Nidhugg's infrastructure takes into account only the non-conservative ones when it comes to searching for threads in set of threads such as sleep sets. As a result, at some points of the code we have to look for both the conservative and the non conservative branches in the set. Another important problem arises when both conservative and non conservative branches are added at the same point. In that case the conservative branch prevails. Looking at the writer-2readers example if we have chosen the non-conservative branch then the trace that begins with the r2 would have been blocked by the sleep sets.

5.6 Implementation of Source-BPOR

The implementation is again based on modifications on the procedure `add_branch()` since every change in the `see_events()` procedure is still necessary for this algorithm. We can infer from the algorithm that will choose the same candidate with the Source-DPOR when we are dealing with a non-conservative branch and the same candidate as in BPOR in case the branch is conservative. In order to differentiate for the two cases we add a second parameter at the routine so we can be aware of the nature of the branch (conservative or

non-conservative). The pseudo code for `add_branch()` is shown in Algorithm 17.

Algorithm 17: add_branch() routine for Source-BPOR

```
1  Function add_branch(b, is_conservative)
2      candidates = ∅ ;
3      lc = null ;
4      lpc = null ;
5      E' := E starting from next_E(b) ;
6      foreach e ∈ dom(E') do
7          if b → e or ∃c ∈ candidates | c → e then
8              continue ;
9          if e → last(E) then
10             lpc := e.pid;
11         lc := e.pid;
12         if e.pid ∈ backtrack(b) or e.pid ∈ sleep_set(b) then
13             if not is_conservative or lc → last(E) then
14                 return ;
15     if lpc and is_conservative then
16         backtrack(b) := backtrack(b) ∪ lpc
17     else
18         backtrack(b) := backtrack(b) ∪ lc
```

Chapter 6

Evaluation of Implemented Algorithms

In this chapter, the performance of each implemented technique will be discussed. Firstly, the performance of Nidhugg-DPOR is demonstrated in order to validate that its performance, indeed differs from Source-DPOR. The evaluation takes place in two parts. In the first part, short synthetic programs are used, while in the second part real world software is tested. Synthetic programs can be found in the appendix section. One area where Nidhugg is tested is the verification of the Read Copy Update technique of the Linux kernel.

6.1 Synthetic Tests

There are many tests provided from various sources. Most of these testcases are not complicated at all since their purpose is to demonstrate the performance difference of the Source-DPOR and Nidhugg-DPOR.

- The writer-N-readers test: In this test N threads read (readers) the same global variable and one thread (writer) writes that variable. It is important to notice that in this case there are some other local operations taking place before the read of the variable. As a result we must expect different results between source sets and persistent sets.

- Account: This test is a small bank account simulation which uses mutex locks to prevent simultaneous operations on the account. There are three possible operations: The deposit operation increases the balance by an amount. The withdraw operation decreases the balance by a certain amount. The check_result operation confirms final_balance $==$ initial_balance $+$ deposit $-$ withdraw and can only happen after both deposit and withdraw are completed.

- Micro: In this test three threads are spawned that perform the x++ operation twice. The x++ operation consists of two operations a read operation and a write operation.

- Last-zero test:The program consists of N+1 threads which operate on an array of N+1 elements which are all initially zero. In this program, thread 0 searches the array for the zero element with the highest index, while the other N threads read one of the array elements and update the next one. The final state of the program is uniquely defined by the values of i and array[1..N]. Last-zero does not produce more traces when DPOR is used for reasons that will be explained later. However a modification of the .ll file can expose the difference.

- Indexer.c: This benchmark uses a compare-and-swap(CAS) primitive instruction to check whether a specific entry in a matrix is 0 and set it to a new value.

- Indexermod.c: In this benchmark all the threads traverse and try to write the matrix at the same order and as a result many conflicts emerge.

6.2 RCU

Read-Copy-Update (RCU) is a synchronization mechanism, which was invented by McKenney and Slingwine [McKe98], based on mutual exclusion. It was added to the Linux kernel in October of 2002. RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers.
DPOR was used as an approach to systematically test the code of the main flavor of RCU used in the Linux kernel (Tree RCU) for concurrency errors, under sequential consistency. The modeling allows Nidhugg, a stateless model checking tool, to reproduce, within seconds, safety and liveness bugs that have been reported for RCU [Koko17b].
RCU provides an ideal testcase to evaluate the various DPOR and Bounded DPOR algorithms since it is:

- It is a real world software and not just a synthetic test.
- The number of traces (different schedulings) is large enough to demonstrate the differences in the performance.
- Previous work [Koko17b] enables us to evaluate the correctness of each algorithm's implementation.

6.3 Evaluation of Unbounded Algorithms

As it was established in the chapter 3 the implementation of the persistent sets is crucial since they are utilized in every bounding technique. In this section we demonstrate performance differences between Source-DPOR and Nidhugg-DPOR in both synthetic tests and RCU.

6.3.1 Evaluation of Persistent sets on Synthetic tests

The results are presented with two different ways. The writer-N-readers testcase result is given with a graph, in Figure 6.1, in order to demonstrate the escalation of the state space as well as the greater impact the source-DPOR has. The rest of the results are given in Table 6.1 so they can be easily compared. We deliberately do not show the duration of the execution since for most testcases the number of traces explored is really small. The execution of the synthetic test cases delineated that Source-DPOR indeed performs better than Nidhugg-DPOR. As it was expected Source-DPOR explores less traces than the Nidhugg-DPOR. It is important to notice that this difference is caused by the sleep set blocked traces that are produced by the DPOR algorithm that are omitted by the source DPOR. The reduction in the number of traces explored is not the same of all the testcases. For example in some testcases there is a significant decrease of the explored traces while in others the reduction is not so great. It varies due to the

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 6.1: Source-DPOR vs Nidhugg-DPOR for synthetic tests

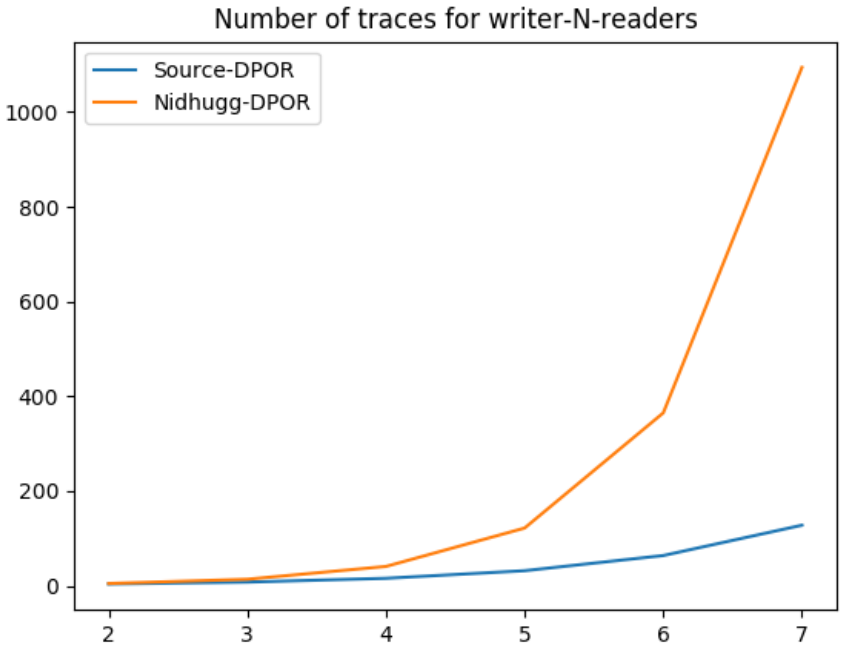

Figure 6.1: writer-N-readers

6.3.2 Evaluation of Persistent sets on RCU

We noticed that there is no difference between Source sets and persistent sets thus no results are presented since they coincide with [Koko17b]. The reason why the results of Nidhugg-DPOR and Source-DPOR are the same may be due to the operations that take place which not allow for the optimization of the Source-DPOR to be effective. Another reason is the LLVM IR which is used by Nidhugg and will be explained in the next section.

6.4 Comparison with Concuerror results

The are many cases where Concuerror's Source-DPOR explores less traces than Classic-DPOR while Nidhugg's Source-DPOR does not explore less traces than Nidhugg-DPOR. Moreover. Nidhugg-DPOR implementation seems to explore less number of traces than Concuerror's Classic-DPOR [Abdu14] which, in many cases, equals with the numbers of traces explored by Source-DPOR. Here we present the reasons that justify this behavior.

- The implementation of the persistent set calculation: In Concuerror this calclation is more relaxed than the in Nidhugg as a result Nidhugg calculates sometimes smaller persistent sets. In Figure 6.2 the example of lastzero testcase in Concuerror is given when a larger persistent set is calculated. In fact, the process q should never have been added to the persistent set since it does not conflict with any other process.

- The number of traces explored is closely related by scheduling of the events: Let a program consist of a process which writes on variable x and two process that read variable x. In Figure 6.3 the exploration is demonstrated when one reader is scheduled first. We notice that exactly 4 traces are explored. On the other hand, provided that the writer was scheduled first 6.4 5 traces with one sleep set blocked traces would have been explored.

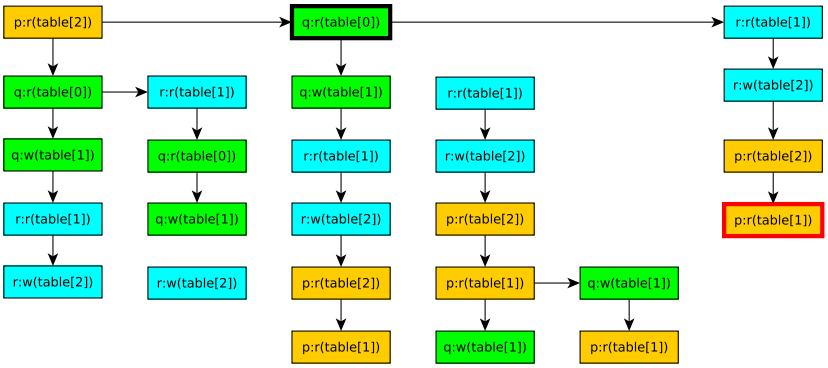

Figure 6.2: Lastzero Concuerror

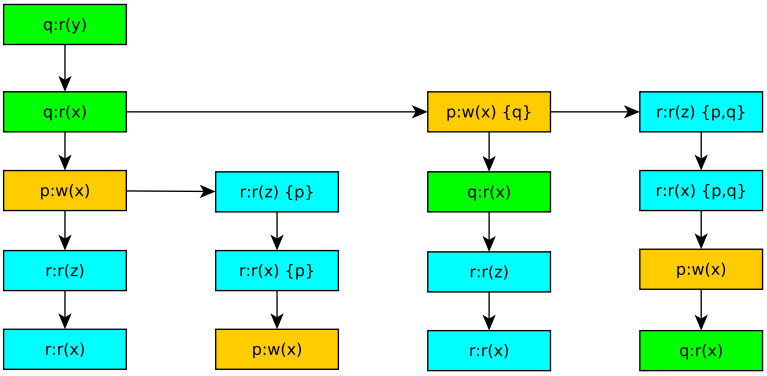

Figure 6.3: Scheduling Effect reader-writer-reader

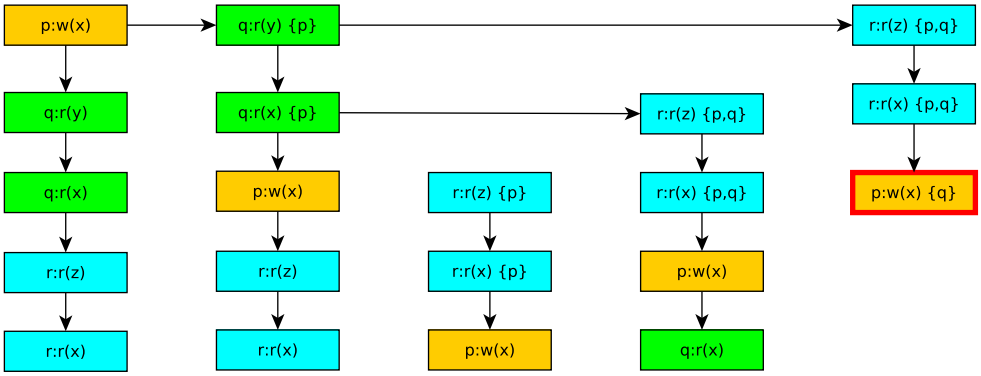

Figure 6.4: Scheduling Effect writer-reader-reader

6.5 Evaluation of Bounding Techniques

The evaluation of the bounding techniques takes into account two aspects. The number of traces explored and the soundness. The former is closely related with the amount of time required for a bug to be found or the whole state space to be explored. The second is important because it demonstrates the tradeoff between time and accuracy of the results discussed in chapter 4. It is intelligible that a faster algorithm may compromise the soundness of the state space.

6.5.1 Evaluation of Bounding Techniques on Synthetic tests

The results for the testcases are demonstrated in this section. Again they are presented in two different ways.

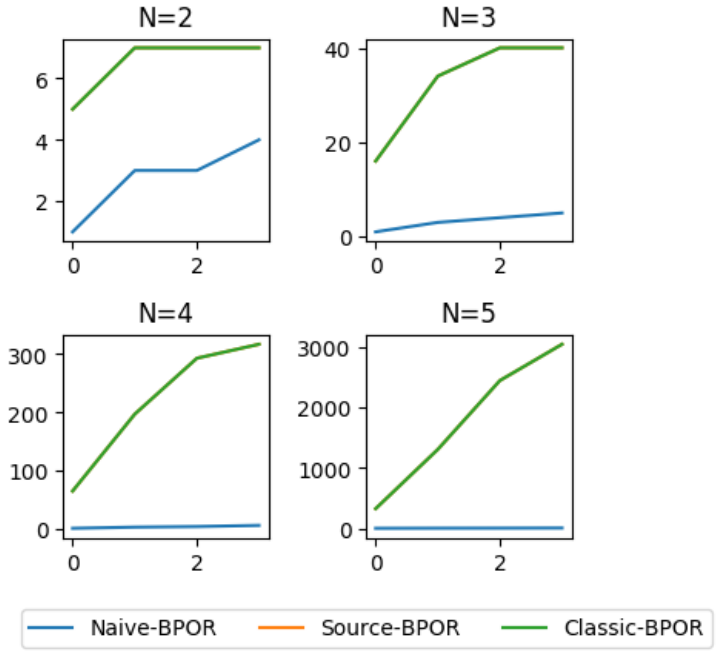

Figure 6.5: writer-N-readers bounded

As it was expected the Naive-BPOR explores significantly less traces than the Nidhugg-BPOR and the Source-DPOR. However, as it was previously discussed, the whole state space

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 6.2: Traces for various bound limits

ver:	3.0			3.19			4.3			4.7			4.9.6		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	19398	295.42	NF	24760	839.27	NF	28996	1365.18	NF	11076	546.84	NF	28996	1457.11	NF
-DASSERT_0	145	2.19	F	37	1.36	F	29	1.77	F	29	1.97	F	29	2.05	F
-DFORCE_FAILURE_1	146	2.19	F	41	1.48	F	33	1.94	F	33	2.16	F	33	2.23	F
-DFORCE_FAILURE_2	4	0.32	F	3	0.53	F	3	0.74	F	3	0.9	F	3	0.92	F
-DFORCE_FAILURE_3	2372	30.82	NF	13264	464.77	F	8114	408.74	F	8114	423.19	F	8114	440.16	F
-DFORCE_FAILURE_4	84	1.39	F	79	3.15	F	24	1.99	F	43	3.32	F	43	3.44	F
-DFORCE_FAILURE_5	4888	64.83	NF	9	0.85	F	9	1.21	F	9	1.43	F	9	1.46	F
-DFORCE_FAILURE_6	1	0.94	F	2	2.7	F	2	4.21	F	2	8.03	F	2	8.53	F
-DLIVENESS_CHECK_1	2024	26.33	NF	608	11.26	NF	488	13.38	NF	488	14.24	NF	488	14.92	NF
-DLIVENESS_CHECK_2	3888	53.82	NF	608	11.2	NF	516	14.84	NF	516	15.72	NF	516	16.56	NF
-DLIVENESS_CHECK_3	2184	27.62	NF	688	13.31	NF	488	13.5	NF	532	15.99	NF	532	16.76	NF

Table 6.3: RCU results without 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
-	1	0.2	NF	2	0.18	NF	1	0.29	NF	2	0.31	NF	1	0.57	NF	2	0.59	NF
-DFORCE_FAILURE_1	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DFORCE_FAILURE_3	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.32	NF	1	0.56	NF	2	0.61	NF
-DFORCE_FAILURE_5	1	0.17	NF	2	0.18	NF	1	0.29	NF	2	0.3	NF	1	0.56	NF	2	0.58	NF
-DLIVENESS_CHECK_1	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DLIVENESS_CHECK_2	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DLIVENESS_CHECK_3	1	0.17	NF	2	0.18	NF	1	0.29	NF	2	0.3	NF	1	0.56	NF	2	0.6	NF

Table 6.4: RCU results for bound $b = 0$

is not explored. The number of traces explored by the sound algorithms is significantly greater and it caused by the many conservative branches that are added in order to achieve soundness. Surprisingly, there is no difference between the other two bounding techniques. An explanation is given later.

6.5.2 Evaluation of Bounding Techniques on RCU

The results of the various implementetions of BPOR are given here. Notice that since the Source-DPOR did not resulted less traces than the DPOR we could not expect from the Source-BPOR and Nidhugg-BPOR to differentiate. Moreover tests did not reveal any difference. For these reasons only the performance of Naive-BPOR and Nidhugg-BPOR is examined. In each table the results with a given bound are demonstrated. Specifically the exploration time and the number of traces are shown. Moreover there is a cell indicating whether the assertion was found (We note F for found and NF for not found).
We notice that some assertions are found significantly faster. The most spectacular result is the `-DFORCE_FAILURE_3` which is found in only 6 seconds for bound $b = 3$ whereas it requires 464.77 seconds in the unbounded version. Moreover we notice for bound $b = 4$ all the errors that are found in the unbounded version are found. As a result the empirical observation that errors occur in low bound count seems to be confirmed. However, we have to underline that these are contrived errors aiming to verify the correctness of the

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 6.5: RCU results for bound $b = 1$

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
-	9	0.29	NF	353	5.16	NF	5	0.39	NF	153	3.8	NF	5	0.68	NF	153	6.51	NF
-DFORCE_FAILURE_1	9	0.3	NF	353	5.21	NF	5	0.37	NF	153	3.77	NF	5	0.68	NF	153	6.54	NF
-DFORCE_FAILURE_3	9	0.31	NF	188	2.69	NF	5	0.39	NF	201	7.03	F	5	0.72	NF	258	14.24	F
-DFORCE_FAILURE_5	9	0.29	NF	306	4.27	NF	5	0.36	NF	105	2.51	NF	5	0.67	NF	90	3.65	NF
-DLIVENESS_CHECK_1	9	0.31	NF	182	2.62	NF	5	0.37	NF	94	1.97	NF	5	0.69	NF	79	2.84	NF
-DLIVENESS_CHECK_2	10	0.34	NF	216	3.15	NF	5	0.37	NF	94	1.97	NF	5	0.68	NF	97	3.55	NF
-DLIVENESS_CHECK_3	9	0.3	NF	201	2.78	NF	5	0.36	NF	105	2.27	NF	5	0.68	NF	88	3.19	NF

Table 6.6: RCU results for bound $b = 2$

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
-	17	0.5	NF	1627	24.18	NF	8	0.42	NF	603	15.76	NF	8	0.77	NF	659	29.29	NF
-DFORCE_FAILURE_1	17	0.51	NF	1627	24.36	NF	8	0.42	NF	603	15.84	NF	8	0.77	NF	659	29.25	NF
-DFORCE_FAILURE_3	17	0.5	NF	634	8.78	NF	8	0.57	NF	1091	36.71	F	8	1.0	NF	1481	79.17	F
-DFORCE_FAILURE_5	17	0.5	NF	1157	16.0	NF	8	0.4	NF	386	9.56	NF	8	0.73	NF	324	13.01	NF
-DLIVENESS_CHECK_1	17	0.51	NF	597	8.28	NF	8	0.42	NF	251	5.18	NF	8	0.76	NF	198	6.67	NF
-DLIVENESS_CHECK_2	20	0.64	NF	767	10.93	NF	8	0.41	NF	251	5.19	NF	8	0.76	NF	258	8.99	NF
-DLIVENESS_CHECK_3	17	0.5	NF	665	9.0	NF	8	0.42	NF	292	6.24	NF	8	0.76	NF	232	7.91	NF

Table 6.7: RCU results for bound $b = 3$

rcu and as a result they cannot be regarded as substantial evidences. As it is expected for larger bounds ($b = 4$) the number of traces grows exponentially. An other impressive result is that when the bound grows larger the errors takes longer to be found. If we take a look at `-DFORCE_FAILURE_3` again we notice that the error takes significantly longer to be tracked even through it exposed for the first time at bound $b = 2$. For $b = 4$ the exploration will was stopped since it exceeded 100,000 traces. On the other hand many assertions are found faster with source-DPOR.

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 6.8: RCU results for bound $b = 4$

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
-	19398	316.78	NF	2	0.18	NF	24760	907.7	NF	2	0.31	NF	28996	1739.91	NF	2	0.59	NF
-DFORCE_FAILURE_1	146	2.41	F	275	4.2	4	41	1.6	F	182	5.51	4	33	2.42	F	300	15.42	4
-DFORCE_FAILURE_3	2372	33.9	NF	2	0.18	NF	13264	539.05	F	201	7.03	2	8114	492.84	F	258	14.24	2
-DFORCE_FAILURE_5	4888	71.29	NF	2	0.18	NF	9	0.92	F	60	2.34	4	9	1.52	F	60	3.92	4
-DLIVENESS_CHECK_1	2024	28.94	NF	2	0.18	NF	608	12.79	NF	2	0.31	NF	488	16.72	NF	2	0.59	NF
-DLIVENESS_CHECK_2	3888	56.71	NF	2	0.18	NF	608	12.76	NF	2	0.31	NF	516	18.34	NF	2	0.59	NF
-DLIVENESS_CHECK_3	2184	30.68	NF	2	0.18	NF	688	18.79	NF	2	0.3	NF	532	18.61	NF	2	0.6	NF

Table 6.9: Comparison between DPOR and BPOR

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 6.10: Comparison between DPOR and BPOR with the bug

6.5.3 A known bug

As it was discussed in previous section, the scheduling priorities of Nidhugg should be changed in order for the running thread to be prioritize since it does not increase the bound count. However this alternation in the priority causes Nidhugg to explore many more traces in unbounded search for an unknown reason. In order to deal with this problem alternation in priority occurs only when bound is applied. As a result the comparison between DPOR and BPOR is biased. Looking at table 6.10 we can clearly see that the minimum traces required for BPOR to track the bug for the first time are always less than DPOR

6.6 Equivalence between Classic-BPOR and Source-BPOR (Correctness of Source-BPOR)

Surprisingly the results of Classic-BPOR and Source-BPOR always coincide. However, further investigation of this behavior can reveal that these two techniques are actually equivalent.
It can be proved that a branch which was rejected by the Source-DPOR but accepted by the Classic-BPOR algorithm as a non-conservative one will be added as conservative by the source-bpor algorithm.
Let us assume a branch of the thread b that is added as a non-conservative by the BPOR algorithm. Let T be the persistent set at that point.
By the definition of persistent-sets this means that there is a $t \in T$ which conflicts with an execution step of b.
This non-conservative branch is rejected by the Source-BPOR. We know that there must be a trace such that thread b occurs before t. Since b was rejected there must be another branch s which shares the same initials with b,
When s is scheduled another block will be created.

- Case 1: s has an execution step which conflicts with b. Hence, there must be a trace where b happens before some step of s and s happens before t. Since b happens-before some execution step of s but share the same initials with s, b must be added as a conservative branch at the point where it was rejected at the initially. As shown in the Figure 6.6, the branch which seems to be initially rejected, will finally be added by the Source-BPOR and as a result belongs to the source-set.

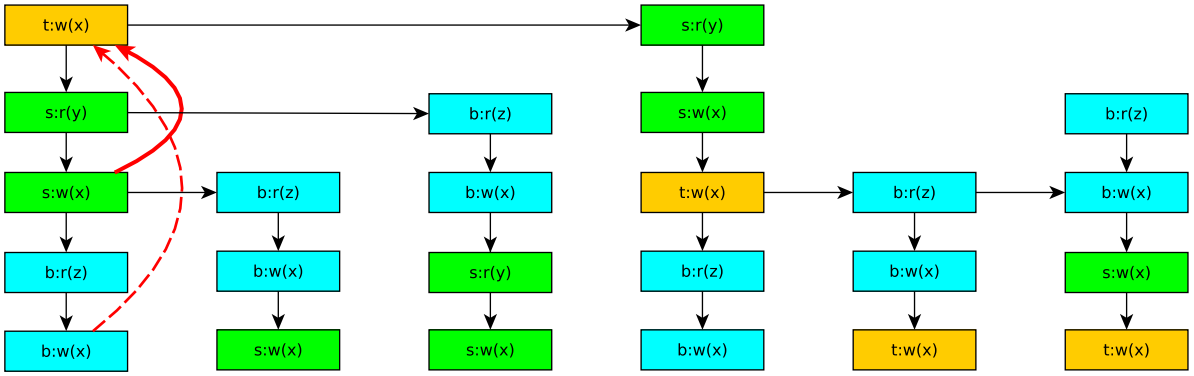

Figure 6.6: Source-BPOR and Nidhugg-BPOR equivalence Case 1

- Case 2: s doesn't conflict with b (both b and s are read operations). There must b a trace s.b.t (where s,b,t is the execution of all the steps of s,b,t). Since t conflicts with an execution step of s the first step of b is an initial for t and it will be added both as non-conservative branch and as conservative at the beginning of the block where it was rejected by the Source-DPOR. For Figure 6.7, both s and b belong to the persistent set. However, the b thread will be rejected since it shares the same initials with the s thread. However it will be added as a conservative set. Notice that it would be added as a non-conservative as well but we have already shown that when both conservative and non-conservative branches of the same thread are added we must keep the conservative one.

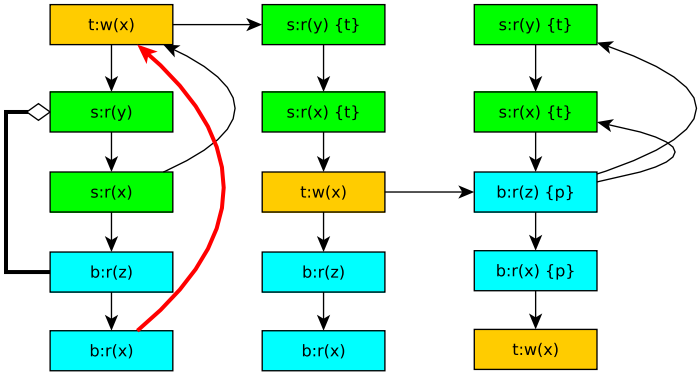

Figure 6.7: Source-BPOR and Nidhugg-BPOR equivalence Case 2

A more intuitive explanation of the equivalence of the two techniques would be based on the following two observation:

- Let B_v be a function that calculates the bound count then $B_v(pre(E, e)) \leq B_v(E)$ for every $e \in E$.
- The points in the trace where the bound count increases are those where branches are added.

As a result a Classic-BPOR algorithm would have to add conservative branches at points where the bound increases. The non-conservative branches that would have been rejected by the Source-DPOR are added as conservative ones to since the lead to already explored traces but with a smaller bound count.

We have proved that Source-BPOR is sound since the traces explored by the Classic-BPOR are subset of the traces explored by the Source-BPOR.

Chapter 7

Further Discussion on Bounding Problem

In this chapter we discuss alternative ideas of approaching the preemption bounding problem of the DPOR. We suggest a new approach which is shown to be equivalent to the addition of conservative branches but does not require the addition of conservative branches.

7.1 Techniques without the Addition of Conservative Branches

In previous chapters we discussed the challenges one have to deal with when designing a Bounded DPOR algorithm. We saw that there there is no apparent significant improvement can be made with the use of conservative branches and that other optimizations used for the unbouded versions of DPOR cannot have analogous results.

7.1.1 Motivation

The only algorithm that does not add any conservative branch is the Naive-BPOR. For a sufficient bound an erroneous trace would have still be found using this technique. The drawback of this algorithm is its unsoundness. In this algorithm a function which calculates the number of preemptive switches in the current thread is used. However many of the preemptive switches that are counted would be avoided.
An example is given in Figure 7.1 further explaining this idea. Let a program consisting of two processes p and q. Process p writes a shared variable x and process q reads the shared variable y (which is not modified at any point by any other process), and the shared variable x. There are only two possible schedulings i.e. one that the writing of x precedes the reading and one that these two commands are reversed. However, in order to make the reversion a preemptive switch must be introduced. Suppose that the first execution sequence is $q.q.p$, the other will be $q.p.q$ since the first step of q has no race with step of p.

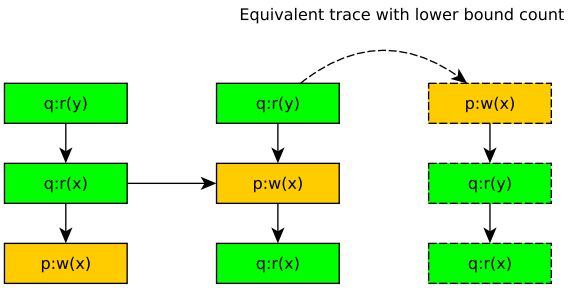

Figure 7.1: An example of avoidable preemption-switch

In Figure 7.1 the preemptive switch that takes places would have been easily avoided there is an obvious inversion of the two blocks $q : read(y)$ and p. But what allows such

an inversion?
The answer lies to the events of block (in this particular case the block consists of one step only). The first block reads a variable which is not used by any other block. It is fathomable that this block can be switched with the next block since there is no a happens before relationship with the two blocks.
This observation leads to the next question: Which of the preemption switches are compulsory? (Or equivalently, which traces cannot be produced without a preemption switch?) Moreover, is it possible for a given trace to calculate the minimum number of preemptive switches among all the equivalent traces?

7.1.2 An Algorithm without Conservative Branches

An algorithm that would preform such a bounded search would be different from the Naive-BPOR only concerning the function that calculates the bound count of the traces. This function f would have to be constantly ascending i.e. it would not be possible to calculate a lower bound later for the same traces. Given a suffix E, $f(E) \leq f(E.E')$ for any E'.
The general form of the algorithm is given in Algorithm 18.

Algorithm 18: General form of the BPOR without branch addition

```
   Result: Explore the whole state space within the bound
1  Explore(∅);
2  Function Explore(S)
3  |  T = Sufficient_set(final(S)) for all t ∈ T do
4  |  |  if min{B_v([S.t])} ≤ c then
5  |  |  |  Explore(S.t)
6  |  |  end
7  |  end
```

Comparing Algorithm 18 with Algorithm 1 we notice that instead of calculating the $B_v(S.t)$ i.e., the bound count of the current trace value, we calculate the $min(B_v[S.t])$ i.e., minimum of all B_v values of the traces that are equivalent with $S.t$.

7.1.3 Calculating Minimum Bound Count

The only thing left is the construction of this function f. For a given trace E which consists of blocks many happens-before relations hold. Each equivalent trace should also compensate with these relations. It is also possible for different instructions in one block different happens-before relations hold true. For this section only, we will consider that a happens-before relation is a relation that holds between blocks. This is done for two main reasons:

- The algorithm described later is simplified.
- We are not interested in further breaking each block and as a result we can regard is block as an entity.

These happens-before relations form a graph. This graph consists of nodes which are the blocks and edges which are these relations. Obviously blocks of the same thread have a happens-before relation. We can also move from one block to another as long as these blocks happen concurrently. We add weights to each edge. The edges that connect to blocks of the same thread weigh 0. Edges that start from a block that is blocked or the most recently added block of each thread weigh 0 since blocked blocks do not increase

the bound count and we do not know if the last block of each thread is indeed the last one. All the other edges which represent preemption points have weight 1.
In order to find the minimum bound count we have to traverse this all the blocks of this graph without breaking at any point the happens-before relations. Hence, the minimum bound count corresponds to the minimum hamiltonian path that compensates with all happens-before relations of the trace that is explored.
Since the calculation of the minimum hamiltonian path is demanding we have to create a graph that limits as much as possible traverses that break the happens-before relations. All traversals that cover the whole graph passing from each node only once are equivalent traces.
An algorithm on how to add a block to a given graph is given at Algorithm 19. The algorithm works using induction. Initially the graph consists of the first block. When a block of the trace is completed then we add it to the dependency graph. We connect the new block with each concurrent block with double edges with the new block. Moreover we connect the most recent block of each thread that happens before the new block with a directed edge ending to the new block.

Algorithm 19: Adding a new block to the dependencies' graph

```
1  Function AddBlock(block,graph)
2      if previous block of the same thread was not blocked then
3          increase the weigh of the edges coming from the previous block to 1 ;
4      for each thread t do
5          list:= preceding blocks t;
6          for l in reversed(list) do
7              if l ↔ block then
8                  add edge from block to l with weight 0 ;
9                  if l is not last then
10                     add edge from l to block with weight 1 ;
11                 else
12                     add edge from l to block with weight 0 ;
13             if l → block then
14                 if l is not last then
15                     add edge from l to block with weight 1 ;
16                 else
17                     add edge from l to block with weight 0 ;
18                 break ;
```

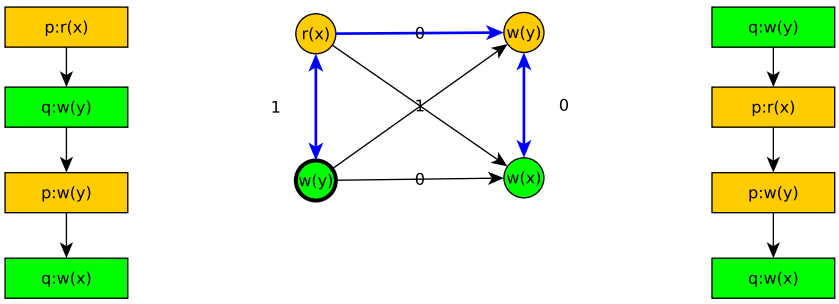

Figure 7.2: Graph example

In Figure 7.2 a simple example of such a graph is demonstrated. For this trace we notice that $w(y)$ of q thread is concurrent with $r(x)$ while it happens before $w(y)$ of the p thread. Each transition costs 1 preemption switch that is why the weight is 1. Moreover transitions between the same thread cost 0. The most important fact, however, is that if for any reason we try to violate the happen before relation (e.g. starting from $r(x)$ we jump to $w(x)$) there is no way to traverse all the nodes.
We can see that there is a hamiltonian path with weight 1 for the given trace. In fact, this is the minimum hamiltonian path of the graph. We can easily realize that there is no equivalent trace with the initial one that has bound count less than 1.
We can infer that the calculation of this bound count is reduced to the weight of the minimum hamiltonian path of this graph. This problem it is known to be NP-complete. As a result any algorithm that would calculate this weight would not be significantly better than a DFS-exploration.
This is an extremely interesting indication of the difficulty of the DPOR bounding problem since the addition of the conservative sets imply this DFS exploration at the state space. As a result this algorithm would not be better than the already proposed BPOR algorithm. Now that the difficulty of this approach is clear a new question arises. Is it possible to approximate the total weight of the minimum hamiltonian path? Such an algorithm would cover a greater state space than the Naive-BPOR without the explosion caused by the conservative branches.

7.1.4 Approximating Bound Count

There are two approaches examined in order to approximate the value of the bound count. The idea of both algorithms is based on this observation: A preemption switch is compulsory when for two blocks of the same process A a block of another process B must intervene in order for the happens-before relations to hold true. As a result, the execution of the first A block should stop so the execution of the B block takes place followed by the execution of the A block again. Hence it should hold $e_1(A) \rightarrow e(B) \rightarrow e_2(A)$. In case of $e_1(A) \not\rightarrow e(B)$ or $e(B) \not\rightarrow e_2(A)$ we could invert the blocks without affecting the happens-before relations and, thus construct an equivalent trace with lower bound count. The algorithm is presented here:

Algorithm 20: First Approximation Algorithm

```
1 Function BoundCount(E,current_bound)
2     for i = 0 to len(E) − 1 do
3         if E[i].pid = last(E).pid then
4             higher_block = i ;
5             break ;
6     for i = higher_block + 1 to len(E) − 1 do
7         if E[higher_block] → E[i] → last(E) then
8             current_bound++ ;
9             return ;
```

In Algorithm 20 we find the most recent block with the same pid as with the last block. We, then try to find if there is an event that happens before the first and after the last event. If exists such an event we increase the counter. Notice that for establishing the happens-before relation vector clocks can be used. Moreover, it is obvious that more happens-before relations can be counted.
Th second algorithm explores more state space than it is required.

Algorithm 21: Second Approximation Algorithm

```
1 Function BoundCount(E,current_bound)
2     for i = len(E) − 1 to 0 do
3         if E[i].pid = last(E).pid then
4             lower_block = i ;
5             break ;
6     for i = lower_block + 1 to len(E) − 1 do
7         if E[lower_block] → E[i] → last(E) then
8             current_bound++;
9             return ;
```

This algorithm starts the search for an event that intervenes the two events of the same id from the immediately previous block with the same id.

7.1.5 Evaluation of Approximating Algorithms

The previous discussed approaches were tested and produced some interesting results. The both estimation algorithms seem to be "more sound" than the BPOR and may explore traces that exceed the bound. This stems from the fact that they tend to underestimate the bound count since there are more complex relations between blocks that result traces with higher bound count than the one estimated. We notice that in writer-N-readers example the number of traces explored is stable for every bound. In fact, each trace of this test has an equivalent trace with zero bound count since in each thread only a command related to a shared variable is executed.

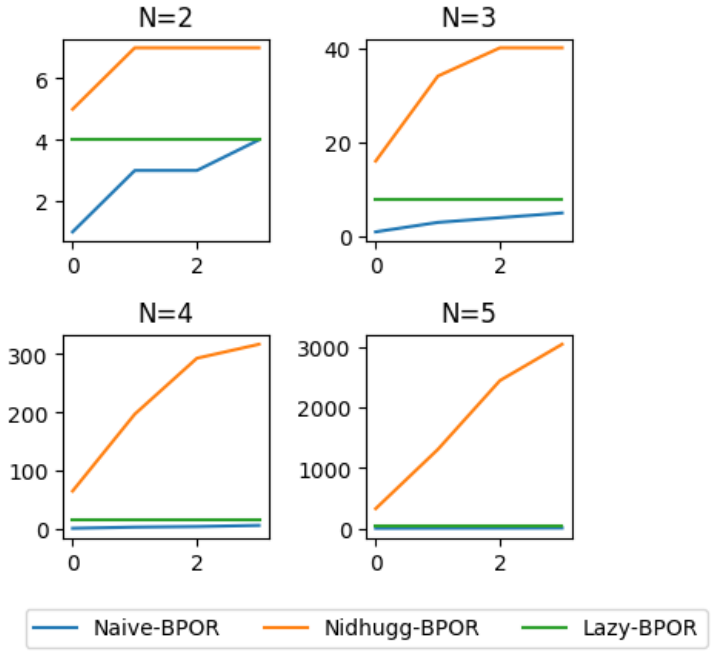

Figure 7.3: writer-N-readers bounded by the first estimation algorithm

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

7.1.6 Implementation of Lazy-BPOR

Some of the testcases such as lastzero or writer-N-readers made clear that an implementation of a bound count function which does not simply counts the preemptive switches in traces can prevent the state space explosion caused by the conservative branches added. The next step is to implement the Lazy-BPOR, an algorithm that calculates the number of compulsory preemptive switches (preemptive switches that cannot be avoided in any equivalent trace with the one examined). The main difference from the Naive-BPOR is that the Lazy-BPOR maintains throughout the execution of the DPOR a graph of the blocks that are contained in the traces. When a new block is created, it is added by the algorithm previously described. When it comes to the calculation of the bound count, the minimum hamiltonian path is calculated. The weight of this path corresponds to the bound count taken into consideration.

Algorithm 22: Lazy-BPOR

```
1  let G =: ∅;
2  Explore(⟨⟩,∅,G,b);
3  Function Explore(E,Sleep,G,b)
4      if ∃p ∈ (enabled(s_[E])\Sleep) such that B_v(E.p) ≤ b then
5          backtrack(E) := p ;
6          while ∃p ∈ (backtrack(E)\Sleep do
7              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
8                  let E' = pre(E, e);
9                  let u = notdep(e, E).p;
10                 if I_{E'}(u) ∩ backtrack(E') = ∅ then
11                     add some q' ∈ I_[E'](u)tobacktrack(E') ;
12             let Sleep' := {q ∈ Sleep | E ⊨ p◇q};
13             if p creates a new block then
14                 let block = last_block(E);
15                 let G' = add_block(block,G);
16             if
                   min{Ham_path(G') which compensate with all happens-before relations of E} ≤
                   b then
17                 Explore(E.p, Sleep, G', b) ;
18                 add p to Sleep ;
```

7.1.7 Lazy-BPOR - RCU Evaluation

The results are demonstrated below. Since we have to compare Lazy-BPOR with BPOR the bugged versions of the DPOR must be used. The bugged version of DPOR is that where the last running thread is prioritized.
At Figures 7.2 and 7.4 we present the results for the various testcases of RCU test suite.

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR		
	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound
-DASSERT_0	145	2.19	F	67	1.88	3	37	1.36	F	27	1.39	2	29	1.77	F	23	1.84	2	29	1.97	F	23	2.06	2	29	2.05	F	23	2.13	2
-DFORCE_FAILURE_1	146	2.19	F	105	2.05	4	41	1.48	F	41	1.57	4	33	1.94	F	33	2.03	4	33	2.16	F	33	2.27	4	33	2.23	F	33	2.34	4
-DFORCE_FAILURE_2	4	0.32	F	4	0.34	1	3	0.53	F	3	0.55	1	3	0.74	F	3	0.77	1	3	0.9	F	3	0.93	1	3	0.92	F	3	0.95	1
-DFORCE_FAILURE_3	2372	30.82	NF	9	0.38	NF	13264	464.77	F	128	30.03	3	8114	408.74	F	109	36.8	3	8114	423.19	F	109	38.23	3	8114	440.16	F	109	39.79	3
-DFORCE_FAILURE_4	84	1.39	F	46	1.37	2	79	3.15	F	27	3.12	2	24	1.99	F	15	2.53	2	43	3.32	F	17	3.46	2	43	3.44	F	17	3.6	2
-DFORCE_FAILURE_5	4888	64.83	NF	9	0.38	NF	9	0.85	F	9	0.88	4	9	1.21	F	9	1.24	4	9	1.43	F	9	1.44	4	9	1.46	F	9	1.48	4
-DFORCE_FAILURE_6	1	0.94	F	1	0.95	0	2	2.7	F	2	2.78	0	2	4.21	F	2	5.31	0	2	8.03	F	2	11.21	0	2	8.53	F	2	9.86	0

Table 7.2: Comparison between DPOR and Lazy-BPOR

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR		
	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	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	858	37.31	F	85	8.57	2	338	15.94	F	75	6.28	2	858	40.42	F	85	9.44	2
-DFORCE_FAILURE_1	247	3.55	F	141	3.45	3	515	18.21	F	121	8.68	3	861	37.8	F	163	21.73	3	341	15.9	F	123	11.28	3	861	40.52	F	163	23.54	3
-DFORCE_FAILURE_2	4	0.34	F	4	0.35	1	3	0.55	F	3	0.52	0	3	0.7	F	3	0.71	0	3	0.86	F	3	0.87	0	3	0.88	F	3	0.9	0
-DFORCE_FAILURE_3	2372	32.1	NF	38	1.87	NF	17094	636.25	F	200	54.62	1	15349	736.84	F	233	103.89	1	15349	714.01	F	233	107.1	1	15349	793.75	F	233	111.37	1
-DFORCE_FAILURE_4	78	1.43	F	51	1.38	2	61	2.74	F	24	2.1	1	16	1.67	F	14	1.79	1	27	2.48	F	17	2.27	1	27	2.6	F	17	2.34	1
-DFORCE_FAILURE_5	12426	185.57	NF	38	3.96	NF	118	4.1	F	52	3.58	3	112	5.12	F	52	5.26	3	112	5.51	F	52	5.66	3	112	5.8	F	52	5.92	3
-DFORCE_FAILURE_6	1	0.98	F	1	0.94	0	2	2.93	F	2	2.77	0	2	4.21	F	2	4.33	0	2	8.13	F	2	8.45	0	2	8.62	F	2	8.56	0

Table 7.3: Comparison between DPOR and Lazy-BPOR without the bug

We compare the algorithm with BPOR. We notice that Lazy-BPOR examines less traces but requires longer time since the cost of the lazy bound count is significantly increased. This is due to the calculation of the minimum hamiltonian path.

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 7.4: Comparison between BPOR and Lazy-BPOR

7.2 Conclusion

Even though the Lazy-BPOR is not proved to be a more efficient way than all the other sound BPOR algorithms examined in this thesis it provides some interesting results.

- 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.

Chapter 8

Concluding Remarks

In this thesis we have implemeted persistent set based DPOR on Nidhugg and used it to implement BPOR. We combined source-DPOR and BPOR and showed that both approaches are equivalent. We used this approach verify RCU and count the minimum preemptive-switches of each injection and showed that bounded DPOR can find all these injections in a shorter period of time exploring less traces. Moreover we explored other techniques that could reduce the number of traces explored showing that are not feasible or are equivalent to the already proposed techniques.
However, this exploration is far from over. Our tasks for the future include:

- The examination and implementation of other bounding techniques and their evaluation compared to the premption bounded dynamic partial order reduction.
- The implementation of optimal DPOR for Nidhugg and the implications of such an implementation to the bounded DPOR.
- The examination of novel techniques such Observers can reduce the state space of the exploration.
- The further useage of Nidhugg in the verification of concurrent software.
- The parallelization of Nidhugg and its effects on performance on both unbounded and bounded search.

Bibliography

[Abdu14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction", in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pp. 373–384, New York, NY, USA, 2014, ACM.

[Abdu15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson and Konstantinos Sagonas, "Stateless Model Checking for TSO and PSO", in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pp. 353–367, New York, NY, USA, 2015, Springer-Verlag New York, Inc.

[Abdu17a] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, *Comparing Source Sets and Persistent Sets for Partial Order Reduction*, pp. 516–536, Springer International Publishing, Cham, 2017.

[Abdu17b] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction", *J. ACM*, vol. 64, no. 4, pp. 25:1–25:49, August 2017.

[Ahme15] Iftekhar Ahmed, Alex Groce, Carlos Jensen and Paul E. McKenney, "How Verified is My Code? Falsification-Driven Verification", in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 737–748, November 2015.

[Algl13] Jade Alglave, Daniel Kroening and Michael Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software", in *Proceedings of the 25th International Conference on Computer Aided Verification*, pp. 141–157, 2013.

[AMDC] "Cool'n'Quiet". Available: https://en.wikipedia.org/wiki/Cool%27n%27Quiet.

[Beye15] Dirk Beyer, "Rules for 4th Intl. Competition on Software Verification", in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2015. Available: https://sv-comp.sosy-lab.org/2015/rules.php.

[Bier03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman and Yunshan Zhu, "Bounded Model Checking", *Advances in Computers*, vol. 58, 2003.

[Chri13] Maria Christakis, Alkis Gotovos and Konstantinos Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pp. 154–163, Los Angeles, CA, USA, 2013, IEEE Computer Society.

[Clan] "LLVM Atomic Instructions and Concurrency Guide". Available: http://llvm.org/docs/Atomics.html#libcalls-atomic.

[Clar99] E.M. Clarke, O. Grumberg, M. Minea and D. Peled, "State space reduction using partial order techniques", *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, Nov 1999.

[Clar04] Edmund Clarke, Daniel Kroening and Flavio Lerda, "A tool for checking ANSI-C programs", in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.

[Coon13] Katherine E. Coons, Madan Musuvathi and Kathryn S McKinley, "Bounded Partial-Order Reduction", ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2013.

[Desn09] Mathieu Desnoyers, *Low-Impact Operating System Tracing*, Ph.D. thesis, Ecole Polytechnique de Montréal, December 2009. Available: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf.

[Desn12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais and Jonathan Walpole, "User-Level Implementations of Read-Copy Update", *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, February 2012.

[Desn13] Mathieu Desnoyers, Paul E. McKenney and Michel R. Dagenais, "Multi-core Systems Modeling for Formal Verification of Parallel Algorithms", *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 51–65, July 2013.

[Dijk] Edsger W. Dijkstra, "Over de sequentialiteit van procesbeschrijvingen". circulated privately.

[Dugg10] Abhinav Duggal, *Stopping Data Races Using Redflag*, Ph.D. thesis, Stony Brook University, 2010.

[Emmi10] Michael Emmi, Shaz Qadeer and Zvonimir Rakamaric, "Delay-Bounded Scheduling", Technical report, September 2010.

[Flan05] Cormac Flanagan and Patrice Godefroid, "Dynamic Partial-order Reduction for Model Checking Software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 110–121, New York, NY, USA, 2005, ACM.

[GCCA] "Built-in Functions for Memory Model Aware Atomic Operations". Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html.

[Gode93] Patrice Godefroid and Didier Pirottin, "Refining dependencies improves partial-order verification methods (extended abstract)", in Costas Courcoubetis, editor, *Computer Aided Verification*, pp. 438–449, Berlin, Heidelberg, 1993, Springer Berlin Heidelberg.

[Gode96] Patrice Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gode97] Patrice Godefroid, "Model Checking for Programming Languages Using VeriSoft", in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pp. 174–186, New York, NY, USA, 1997, ACM.

[Gode05] Patrice Godefroid, "Software Model Checking: The VeriSoft Approach", *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, Mar 2005.

[Gode15] Patrice Godefroide, "Between Testing and Verification: Software Model Checking via Systematic Testing". HVC 2015, 2015.

[Gots13] Alexey Gotsman, Noam Rinetzky and Hongseok Yang, "Verifying Concurrent Memory Reclamation Algorithms with Grace", in *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pp. 249–269, Berlin, Heidelberg, 2013, Springer-Verlag.

[Inte] "Power Management States: P-States, C-States, and Package C-States". Available: https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states.

[Kerna] "NO_HZ: Reducing Scheduling-Clock Ticks". Available: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[Kernb] "RCU Concepts". Available: https://www.kernel.org/doc/Documentation/RCU/rcu.txt.

[Koko17a] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas and Viktor Vafeiadis, "Effective Stateless Model Checking for C/C++ Concurrency", *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 17:1–17:32, December 2017.

[Koko17b] Michalis Kokologiannakis and Konstantinos Sagonas, "Stateless Model Checking of the Linux Kernel's Hierarchical Read-copy-update (Tree RCU)", in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pp. 172–181, New York, NY, USA, 2017, ACM.

[Kurs98] R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigün, "Static partial order reduction", in Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 345–357, Berlin, Heidelberg, 1998, Springer Berlin Heidelberg.

[Lamp78] Leslie Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", pp. 558–565, July 1978.

[Linu] "The Linux kernel". https://www.kernel.org/.

[LKMLa] "rcu: clean up locking for `->completed` and `->gpnum` fields". https://lkml.org/lkml/2009/10/30/212.

[LKMLb] "rcu: fix long-grace-period race between forcing and initialization". https://lkml.org/lkml/2009/10/28/196.

[LKMLc] "rcu: Fix synchronization for `rcu_process_gp_end()` uses of `->completed` counter". https://lkml.org/lkml/2009/11/4/69.

[Love10] Robert Love, *Linux Kernel Development*, Addison-Wesley, 3rd edition, 2010.

[McKe] Paul E. McKenney, "RCU Linux Usage". Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html.

[McKe98] Paul E. McKenney and John D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems", in *Parallel and Distributed Computing and Systems*, pp. 509–518, Las Vegas, NV, October 1998.

[McKe07a] Paul E. McKenney, "The design of preemptible read-copy-update". Available: http://lwn.net/Articles/253651/, October 2007.

[McKe07b] Paul E. McKenney and Jonathan Walpole, "What is RCU, Fundamentally?". Available: http://lwn.net/Articles/262464/, December 2007.

[McKe08a] Paul E. McKenney, "Hierarchical RCU". Available: http://lwn.net/Articles/305782/, November 2008.

[McKe08b] Paul E. McKenney, "RCU part 3: the RCU API". Available: http://lwn.net/Articles/264090/, January 2008.

[McKe08c] Paul E. McKenney, "What is RCU? Part 2: Usage". Available:, http://lwn.net/Articles/263130/, January 2008.

[McKe09] Paul E. McKenney, "Hunting Heisenbugs". Available: http://paulmck.livejournal.com/14639.html, 11 2009.

[McKe10] Paul E. McKenney, "The RCU API, 2010 Edition". Available: http://lwn.net/Articles/418853/, December 2010.

[McKe14] Paul E. McKenney, "The RCU API, 2014 Edition". Available: http://lwn.net/Articles/609904/, September 2014.

[McKe15] Paul E. McKenney, "Verification Challenge 4: Tiny RCU". Available: http://paulmck.livejournal.com/39343.html, 3 2015.

[Musu07] Madanlal Musuvathi and Shaz Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs", *SIGPLAN Not.*, vol. 42, no. 6, pp. 446–455, June 2007.

[Musu08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar and Iulian Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs", in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 267–280, Berkeley, CA, USA, 2008, USENIX Association.

[Pele93] Doron Peled, "All from One, One for All: On Model Checking Using Representatives", in *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pp. 409–423, London, UK, UK, 1993, Springer-Verlag.

[Rela] "Relaxed-Memory Concurrency". Available: http://www.cl.cam.ac.uk/~pes20/weakmemory/.

[Seys12] Justin Seyster, *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*, Ph.D. thesis, Stony Brook University, 2012.

[Spar] "Sparse - a Semantic Parser for C". Available: https://sparse.wiki.kernel.org/index.php/Main_Page.

[Tass15] Joseph Tassarotti, Derek Dreyer and Viktor Vafeiadis, "Verifying Read-copy-update in a Logic for Weak Memory", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pp. 110–120, New York, NY, USA, 2015, ACM.

[Thom16] Paul Thomson, Alastair F. Donaldson and Adam Betts, "Concurrency Testing Using Controlled Schedulers: An Empirical Study", *ACM Trans. Parallel Comput.*, vol. 2, no. 4, pp. 23:1–23:37, February 2016.

[Valm91] Antti Valmari, "Stubborn Sets for Reduced State Space Generation", in *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pp. 491–515, London, UK, UK, 1991, Springer-Verlag.

[Wikia] "Memory ordering". Available: https://en.wikipedia.org/wiki/Memory_ordering.

[Wikib] "Model checking". Available: https://en.wikipedia.org/wiki/Model_checking.

[Wikic] "Read-copy-update". Available: https://en.wikipedia.org/wiki/Read-copy-update.

Appendix A

A.1 Modifications in test suite

For any implementation to be verified the test suit already available with Nidhugg was used. However in the test suit there are many limitations related to the source-DPOR that do not hold true in the BPOR and Source-DPOR. For example the test suit driver would report equivalent traces as errors even though that these traces cannot be eliminated when bounded DPOR takes place. The reasons of this behavior have already been explained. In the section the changes on the test driver are reported. The modification took place was rather straightforward since we just had to mute warnings when the number of traces exceeded the anticipated or equivalent traces were explored more than once. However, in two cases (Atomic_9,Intrinsic_2) the only check that takes place concerns the number of the traces. In these cases only the test suit will report an error. The report of the test suit when bounded DPOR is executed is shown below.
Below are listed some testcases examined throughout.

```
// 1writer-2readers.c
#include <pthread.h>
#include <assert.h>

volatile int c = 0;
void *writer(){
  c = 2;
  return NULL;
}

void *reader(void * arg){
  int local;
  local = c;
  return NULL;
}

int main(int argc, char *argv[]){
  pthread_t t,t2,t3;
  pthread_create(&t,NULL, writer,NULL);
  pthread_create(&t2, NULL, reader, NULL);
  pthread_create(&t3, NULL, reader, NULL);
  return 0;
}

//acount.c
#include <pthread.h>
#include <stdio.h>
#include <assert.h>

pthread_mutex_t m;
//int nondet_int();
int x, y, z, balance;
_Bool deposit_done=0, withdraw_done=0;
```

```
void *deposit(void *arg)
{
  pthread_mutex_lock(&m);
  balance = balance + y;
  deposit_done=1;
  pthread_mutex_unlock(&m);
}

void *withdraw(void *arg)
{
  pthread_mutex_lock(&m);
  balance = balance - z;
  withdraw_done=1;
  pthread_mutex_unlock(&m);
}

void *check_result(void *arg)
{
  pthread_mutex_lock(&m);
  if (deposit_done && withdraw_done)
    assert(balance == (x + y) - z);
  pthread_mutex_unlock(&m);
}

int main()
{
  pthread_t t1, t2, t3;

  pthread_mutex_init(&m, 0);

  x = 1;
  y = 2;
  z = 4;
  balance = x;

  pthread_create(&t3, 0, check_result, 0);
  pthread_create(&t1, 0, deposit, 0);
  pthread_create(&t2, 0, withdraw, 0);

  return 0;
}

//indexer0.c
#include <assert.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include  <stdatomic.h>
#include <stdio.h>

#define SIZE 128
#define MAX  4

atomic_int table[SIZE];

void *thread_n(void *arg)
{
  int tid = *((int *) arg);
  int zero = 0;
  int w, h;

  for (int i = 0; i < MAX; i++) {
    w = i * 11 + tid;
```

```
      h = (w * 7) % SIZE;

      if (h < 0)
        assert(0);

      while (!atomic_compare_exchange_strong_explicit(&table[h], &zero, w,
                  memory_order_relaxed,
                  memory_order_relaxed)) {
  //            printf("%d: %d\n",tid,h);
        h = (h+1) % SIZE;
        zero = 0;
      }
    }
    return NULL;
  }

  int idx[N];

  int main()
  {
    pthread_t t[N];

    for (int i = 0; i < N; i++) {
      idx[i] = i;
      pthread_create(&t[i], NULL, thread_n, &idx[i]);
    }
    for(int i = 0; i<N; i++){
      pthread_join(t[i],NULL);
    }
    return 0;
  }

  #include <assert.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include  <stdatomic.h>

#define SIZE 128
#define MAX  1

atomic_int table[SIZE];

void *thread_n()
{
    int h = 0, zero = 0;
    while (!atomic_compare_exchange_strong_explicit(&table[h], &zero, 1,
                                                    memory_order_relaxed,
                                                    memory_order_relaxed))
    {
        h = (h + 1) % SIZE;
        zero = 0;
    }
    return NULL;
}

int idx[N];

int main()
{
        pthread_t t[N];
```

```
        for (int i = 0; i < N; i++) {
                pthread_create(&t[i], NULL, thread_n, NULL);
        }

        return 0;
}

//lastzero.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include ”stdatomic.h”

int array[N+1];
int idx[N+1];

void *thread_reader(void *unused)
{
        for (int i = N; array[i] != 0; i--);

        return NULL;
}

void *thread_writer(void *arg)
{
        int j = *((int *) arg);

        array[j] = array[j - 1] + 1;
        return NULL;
}

int main()
{
        pthread_t t[N+1];

        for (int i = 0; i <= N; i++) {
                idx[i] = i;
                if (i == 0) {
                        if (pthread_create(&t[i], NULL, thread_reader, &idx[i]))
                                abort();
                } else {
                        if (pthread_create(&t[i], NULL, thread_writer, &idx[i]))
                                abort();
                }
        }

        return 0;
}

//lazy.c

#include <pthread.h>
#include <assert.h>

pthread_mutex_t  mutex;
int data = 0;

void *thread1(void *arg)
{
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  return NULL;
```

```
}


void *thread2(void *arg)
{
  pthread_mutex_lock(&mutex);
  data+=2;
  pthread_mutex_unlock(&mutex);
}


void *thread3(void *arg)
{
  pthread_mutex_lock(&mutex);
  if (data >= 3){
    //assert(0);
  }
  pthread_mutex_unlock(&mutex);
}


int main()
{
  pthread_mutex_init(&mutex, 0);

  pthread_t t1, t2, t3;

  pthread_create(&t3, 0, thread3, 0);
  pthread_create(&t1, 0, thread1, 0);
  pthread_create(&t2, 0, thread2, 0);

  pthread_join(t1, 0);
  pthread_join(t2, 0);
  pthread_join(t3, 0);

  return 0;
}

//micro.c

#include <assert.h>
#include <pthread.h>

int x=0;

void* t1(void* arg)
{
  x++;
  x++;
  assert(0<x);
}

void* t2(void* arg)
{
  x++;
  x++;
  assert(0<x);
}

void* t3(void* arg)
{
  x++;
  x++;
```

```
  assert(0<x);
}

int main(void)
{
  pthread_t id[3];

  pthread_create(&id[0], NULL, &t1, NULL);
  pthread_create(&id[1], NULL, &t2, NULL);
  pthread_create(&id[2], NULL, &t3, NULL);

  return 0;
}
```